OS-9
TECHNICAL I/O
MANUAL

ACKNOWLEDGEMENTS

Many thanks to Warren Brown, Larry Crane, and Peter Dibble for their wisdom, patience, and perseverance.

COPYRIGHT AND REVISION HISTORY

Copyright © 1990 Microware Systems Corporation. All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual or otherwise is prohibited, without written permission from Microware Systems Corporation.

This manual reflects Version 2.4 of the OS-9 Operating System.

Publication Editor:	Walden Miller, Kathleen Flood, Debbie Baier
Revision:	C
Publication date:	October 1990
Product Number:	oio68na68mo

DISCLAIMER

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

REPRODUCTION NOTICE

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

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

TRADEMARKS

OS-9 is a trademark of Microware Systems Corporation.

Microware Systems Corporation • 1900 N.W. 114th Street
Des Moines, Iowa 50325-7077 • Phone: 515/224-1929

Table of Contents

Introduction vii

The OS-9 Input/Output System

The OS-9 Unified Input/Output System 1-1
The Kernel and I/O 1-4
    Kernel I/O Service Requests 1-5
Device Descriptor Modules 1-7
Path Descriptors 1-13
File Managers 1-16
    File Manager Organization 1-17
    File Manager I/O Service Requests 1-18
Device Driver Modules 1-21
    Driver Module Format 1-21
Device Drivers that Control Multiple Devices 1-27
    Simple Devices 1-27
    Multi-Port Devices 1-28
    Multi-Class Devices 1-31
    Examples of Multi-Class Devices Using SCSI System Concept 1-31
Interrupt Driven I/O 1-37
DMA I/O and System Caches 1-39
    Syscache Module 1-39
    Init Module 1-39
    Avoiding Stale Data Problems 1-40
Address Translation and DMA Transfers 1-42

Random Block File Manager (RBF)

RBF General Description 2-1
RBF I/O Service Requests 2-2
RBF Device Descriptor Modules 2-7
RBF Path Descriptor Definitions 2-16
RBF Device Drivers 2-19
Main Driver Types 2-21
RBF Device Driver Storage Definitions 2-22
Device Driver Tables 2-24
Linking RBF Drivers 2-28
RBF Device Driver Subroutines 2-30
INIT 2-31
READ 2-33
WRITE 2-37
GETSTAT/SETSTAT 2-40
TERM 2-46
IRQ Service Routine 2-47

Sequential Character File Manager (SCF)

SCF General Description 3-1
SCF Line Editing 3-2
SCF I/O Service Requests 3-3
SCF Device Descriptor Modules 3-6
SCF Path Descriptor Definitions 3-11
SCF Device Drivers 3-13
Special Characters and NULLs 3-14
Parity Stripping 3-14
Data Flow Control 3-15
SCF Device Driver Storage Definitions 3-17
Linking SCF Drivers 3-20
SCF Device Driver Subroutines 3-22
INIT 3-23
READ 3-24
WRITE 3-26
GETSTAT/SETSTAT 3-28
TERM 3-32
IRQ Service Routine 3-33

Sequential Block File Manager (SBF)

SBF General Description 4-1
Unbuffered I/O 4-2
Buffered I/O 4-2
Considerations When Writing to Tapes 4-2
End-of-Tape Processing 4-3
SBF I/O Service Requests 4-3
SBF Device Descriptor Modules 4-6
SBF Path Descriptor Definitions 4-9
SBF Device Drivers 4-10
Sensing the End-of-Tape 4-10
Tape Positioning Operations 4-12
Tape Streaming 4-13
SBF Device Driver Storage Definitions 4-14
Device Driver Tables 4-16
Linking SBF Drivers 4-18
SBF Device Driver Subroutines 4-20
INIT 4-21
READ 4-23
WRITE 4-24
GETSTAT/SETSTAT 4-26
TERM 4-30
IRQ Service Routine 4-31

End of Table of Contents

NOTES

Introduction

You can use the ***OS-9 Technical I/O Manual*** as a supplement to the ***OS-9 Technical Manual***, which descibes in detail how the I/O system operates. The ***OS-9 Technical I/O Manual*** provides further information to help you create new file managers and device drivers, and supplies examples which you can adapt to your specific system needs. A basic understanding of the ***OS-9 Technical Manual*** is assumed.

This manual contains the following chapters:

- ***Chapter 1 - The OS-9 Input/Output System***
  Explains the relationships between the kernel, device descriptors, path descriptors, and file managers, and how each of these components operates within OS-9.

- ***Chapter 2 - Random Block File Manager (RBF)***
  Explains how to use the RBF manager to process I/O service requests to random access devices, and the parameters that drive it.

- ***Chapter 3 - Sequential Character File Manager (SCF)***
  Explains how to use the SCF manager to process I/O service requests to devices which operate on a character by character basis, and the I/O editing functions available for line-oriented operations.

- ***Chapter 4 - Sequential Block File Manager (SBF)***
  Explains how to use the SBF manager to process I/O service requests to sequential block-oriented mass storage devices

In addition, chapters 2, 3, and 4 each contain a description of how device driver routines for the respective class should operate. These descriptions are based on existing Microware drivers.

If this manual accompanies a media package that contains driver source code (for example, Port Pak, Driver Pak), we recommend that you study the source code in conjunction with this manual.

The OS-9 Input/Output System

The OS-9 Unified Input/Output System

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

- The kernel.
- File managers.
- Device drivers.
- The device descriptor.

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

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

- Maintains the I/O modules by managing various data structures. It ensures that the appropriate file manager and device driver modules process each I/O request.
- Establishes paths. These are the connections between the kernel, the application, the file manager, and the device driver.

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

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

The device descriptor contains the information required to assemble the various components of an I/O subsystem (that is, a device). It contains the names of the file manager and device driver associated with the device, as well as the device's operating parameters. Parameters in device descriptors can be fixed, such as interrupt level and port address, or variable, such as terminal editing settings and disk physical parameters. The variable parameters in device descriptors provide the initial default values when a path is opened, but applications can change these values. The device descriptor name is the name of a device as known by the user. For example, the device /d0 is described by the device descriptor d0.

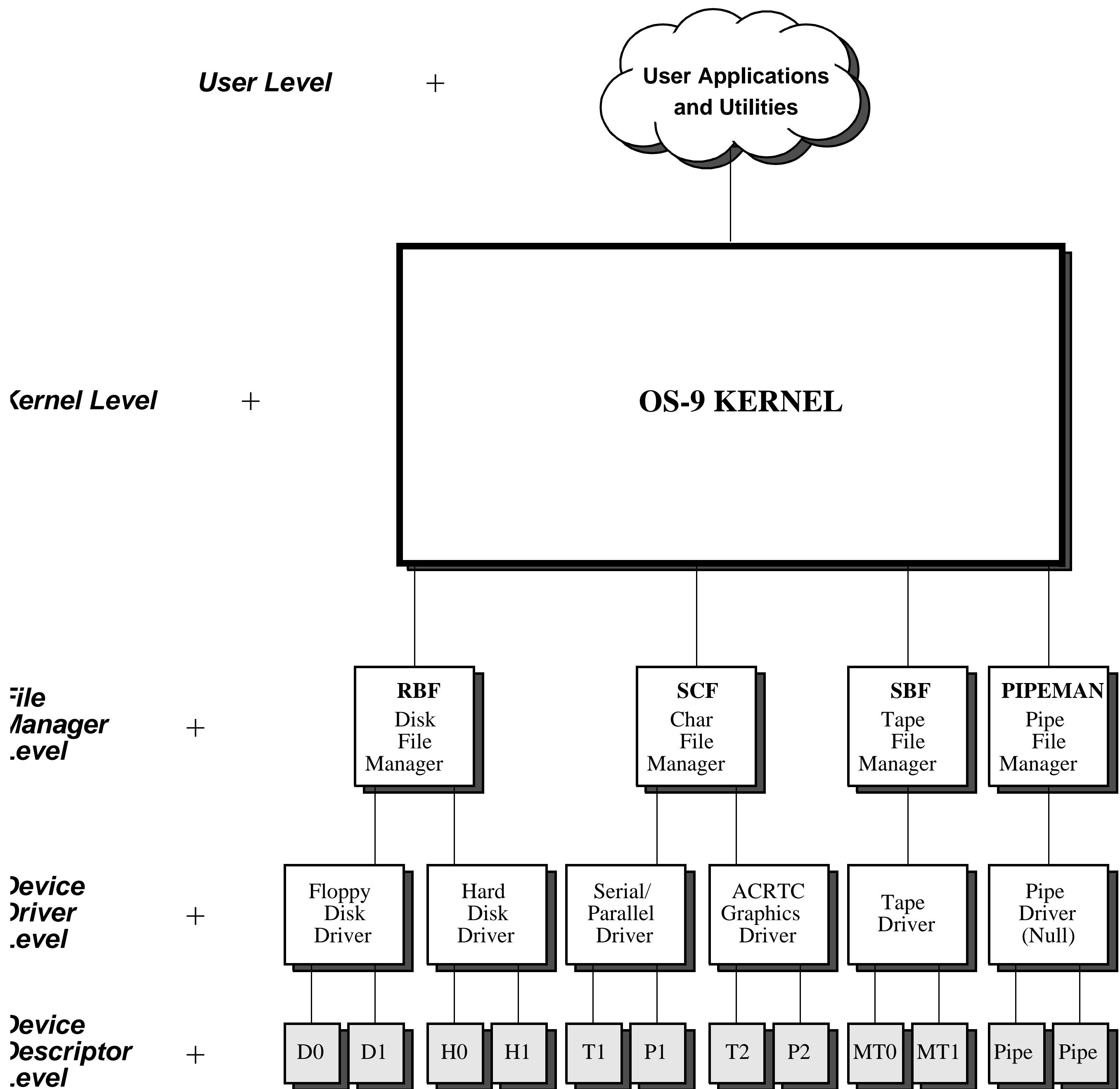


Figure 1-1: OS-9 I/O System Module Organization

The Kernel and I/O

The kernel maintains the I/O system for OS-9. It provides the first level of I/O service by routing system call requests between processes and the appropriate file managers and device drivers. The kernel also allocates and initializes static storage for device drivers.

The kernel maintains two important internal data structures: the device table and the path table. The device table is a list of all devices currently attached (loaded and initialized). The path table is a list of all I/O paths currently open. These tables reflect two other structures respectively: the device descriptor and the path descriptor.

Whenever a path is opened (I$Open), the kernel's attach routine (I$Attach) is called, and it links to the device descriptor of the specified (or implied) device name in the pathlist. The device descriptor contains the port address of the device, the file manager's name, and the device driver's name. The attach routine then links to the specified file manager and device driver. After these components are located, the I$Attach routine inspects the current device table entries, and compares the new device specification with the current entries in the device table.

The I$Attach routine proceeds as follows:

¨ If the device port address, file manager, device driver, and device descriptor match an existing entry in the device table, the device is known to the system. The use count for that device table entry is incremented and the kernel returns to the caller.

¦ If the device port address, file manager, and device driver match an existing device table entry, but the device descriptor does not, this is a new, or synonymous device on the port. A new device table entry is created, its use count is set to one, and the kernel returns to the caller.

Æ If neither of the above situations occur (no match on port address, file manager, and device driver) or this is the first time the path is opened, then the device is unknown to the system. In this case, the kernel allocates static storage for the driver and calls the driver's INIT routine. If INIT does not return an error, then a new device table entry is created, its use count is set to one, and the kernel returns to the caller. If INIT returns an error, the kernel calls the device driver's TERM routine before performing any necessary clean-up and returning the original error.

Whenever a path is closed, its use count is decremented. If the use count becomes zero, the kernel attempts to detach the device (I$Detach) associated with the path from the I/O system. The use count in the device's device table entry is decremented. If the use count becomes zero, the following actions take place:

¨ The device table is searched to determine if another device table entry is using the same static storage as the device being deleted.

¦ If no other device is using the static storage, the driver's TERM routine is called to de-initialize the device. The driver's static storage is then returned to the system.

Æ The device's entry is removed from the device table.

The file manager, device driver, and device descriptor are then unlinked.

Path descriptors maintain the status of I/O operations to devices and files. The kernel maintains pointers to these path descriptors in the path table. Each time a path is created (I$Open, I$Create), a new path descriptor is created and an entry is added to the path table. If I$Dup is used to open a path, only the use count of an existing path descriptor is incremented. When a path is closed and its use count becomes zero, the path descriptor is de-allocated, and the appropriate entry is deleted from the path table.

Kernel I/O Service Requests

File managers are not called for I$Attach, I$Detach, and I$Dup. The kernel performs the necessary system functions for these requests.

I$Attach The kernel performs the following functions:

- **Links to component modules (file manager, device driver, device descriptor)**
- **Determines if a device table entry matches an existing entry for the device**
  If the device port address, file manager, device driver, and device descriptor match, the kernel:
  - **Increments the use count for the device.**
  - **Returns to the caller.**

If the device port address, file manager, and device driver match an existing device table entry, but the device descriptor does not, this is a new (or synonymous) device on the port. I$Attach:

- **Creates a new device table entry.**
- **Sets the use count to one.**
- **The kernel returns to the caller.**

If there is no match on port address, file manager, and device driver, the kernel:

- **Allocates and clears the driver's static storage**
- **Sets V_PORT to the hardware address given in the descriptor**
- **Calls the driver's INIT routine to initialize the hardware**
  If INIT returns an error, the kernel calls the driver's TERM routine, deallocates any resources, and returns the error.
- **Adds the device to the device table**

I$Detach

The kernel decrements the use count for the device. If the use count becomes zero, the kernel searches the device table for other devices using the same static storage. If any are found, the original device table entry is removed from the table. Otherwise, the kernel performs the following actions:

- **Calls the driver's TERM routine**
- **Returns the driver's static storage to the system's free memory pool**
- **Removes the device entry from the device table**

The kernel then unlinks the file manager, device driver, and device descriptor.

I$Dup

The kernel increments the use count (PD_COUNT, PD_CNT) of the path.

Device Descriptor Modules

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

File managers operate on a class of *logical* devices. Device drivers operate on a class of *physical* devices. A device descriptor module tailors a device driver or file manager to a specific I/O port. At least one device descriptor module must exist for each I/O device in the system. An I/O device may have several device descriptors with different initialization parameters and names. For example, a serial/parallel driver could have two device descriptors, one for terminal operation (/T1) and one for printer operation (/P1).

If a suitable device driver exists, adding devices to the system consists of adding the new hardware and another device descriptor. Device descriptors can be in ROM, in the boot file, or loaded into RAM while the system is running.

The module name is used as the logical device name by the system and user (it is the device name given in pathlists). A device descriptor module header consists of the standard module header fields with a type code of device descriptor (DEVIC). The standard device descriptor header is followed by a device-type specific initialization table (see Figure 1-2).

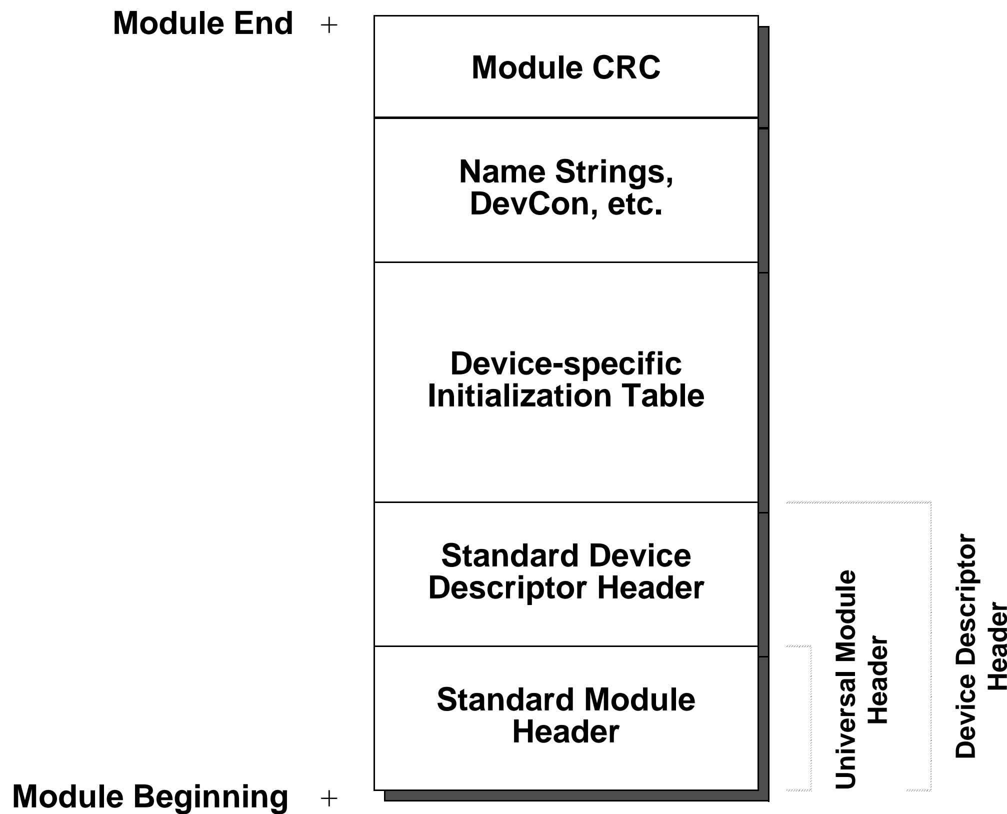


Figure 1-2: Device Descriptor Layout

The standard device descriptor fields are listed below and described in the following pages. Refer to the appropriate chapter of this manual for the specific device-type for the device descriptor initialization table fields.

Offset	Name	Description
$30	M$Port	Port Address
$34	M$Vector	Interrupt Vector Number
$35	M$IRQLvl	Interrupt Level
$36	M$Prior	Interrupt Polling Priority
$37	M$Mode	Device Mode Capabilities
$38	M$FMgr	File Manager Name Offset
$3A	M$PDev	Device Driver Name Offset
$3C	M$DevCon	Device Configuration Offset
$3E		Reserved
$46	M$Opt	Initialization Table Size
$48	M$DTyp	Device Type (first field of initialization table)

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

Name	Description
M$Port	**Port address** M$Port usually contains the absolute physical address of the hardware controller. However, it can be another address (for example, R0/R1). Before the kernel attaches a device (calls its INIT routine), this value is copied into the V_PORT field of the driver's static storage.
M$Vector	**Interrupt Vector Number** The interrupt vector associated with the port, used to initialize hardware and for installation on the IRQ poll table: 25-31 for an auto-vectored interrupt. Levels 1 - 7. 57-63 for 68070 on-chip auto-vectored interrupts. Levels 1 - 7. 64-255 for a vectored interrupt.

Name	Description
M$IRQLvl	**Interrupt Level** The device's physical interrupt level. It is *not* used by the kernel or file manager. The device driver may use it to mask off interrupts for the device when critical hardware manipulation occurs. **NOTE:** Level 7 is a non-maskable interrupt. It should not be used by OS-9 I/O devices. A device set at this level can interrupt the kernel during critical system operations. Level 7 may be used, however, for hardware operations *unknown* to the system (for example, dynamic RAM refreshing).
M$Prior	**Interrupt Polling Priority** Indicates the priority of the device on its vector. Smaller numbers are polled first if more than one device is on the same vector. A priority of zero indicates the device requires exclusive use of the vector.
M$Mode	**Device Mode Capabilities** This byte is used to validate a caller's access mode byte in I$Create or I$Open calls. It may be any combination of the following: bit 0: Set if read access bit 1: Set if write access bit 2: Set if executable access bit 6: Set if single-user access (non-sharable) bit 7: Set if directory file access All other bits are reserved.
M$FMgr	**File Manager Name offset** The offset to the name string of the file manager module for this device.
M$PDev	**Device Driver Name offset** The offset to the name string of the device driver module for this device.

Name	Description
M$DevCon	**Device Configuration** This is the offset to an optional device configuration table. You can use it to specify parameters or flags that the device driver needs and are not part of the normal initialization table values. This table is located after the standard initialization table. The kernel or file manager never references it. As the pointer to the device descriptor is passed in INIT and TERM, M$DevCon is generally available to the driver only during the driver's INIT and TERM routines. Other routines in the driver (for example, Read) must first search the device table to locate the device descriptor before they can access this field. Typically, this table is used for name string pointers, OEM global allocation pointers, or device-specific constants/flags. **NOTE:** These values, unlike the standard options, are not copied into the path descriptors options section.
M$Opt	**Table Size** This contains the size of the device's standard initialization table. Each file manager defines a ceiling on M$Opt.
M$DTyp	**Device Type (First Field of Initialization Table)** The device's standard initialization table is defined by the file manager associated with the device, with the exception of the first byte (M$DTyp). The first byte indicates the class of the device (RBF, SCF, etc.).

Name	Value	Description
DT_SCF	0	Sequential Character File Manager (SCF)
DT_RBF	1	Random Block File Manager (RBF)
DT_Pipe	2	PIPE File Manager (PIPEMAN)
DT_SBF	3	Sequential Block File Manager (SBF)
DT_NFM	4	Network File Manager (NFM)
DT_CDFM	5	Compact Disc File Manager (CDFM)
DT_UCM	6	User Communications Manager (UCM)
DT_SOCK	7	Socket Communications Manager (SOCKMAN)
DT_PTTY	8	Pseudo-keyboard Manager (PKMAN)
DT_INET	9	Internet Interface Manager (IFMAN)
DT_NRF	10	Non-volatile RAM File Manager (NVRAM)
DT_GFM	11	Graphics File Manager (GFM)

The initialization table (M$DTyp through M$DTyp + M$Opt) is copied into the option section of the path descriptor when a path to the device is opened. Typically, this table is used for the default initialization parameters such as the delete and backspace characters for a terminal. Applications may examine all of the values in this table using $GetStt (SS_Opt). Some of the values may be changed using I$SetStt; some are protected by the file manager to prevent inappropriate changes.

The theoretical maximum initialization table size is 128 bytes. However, a file manager may restrict this to a smaller value.

Path Descriptors

Every open path is represented by a data structure called a path descriptor. It contains path-related information required by file managers and device drivers. Path descriptors are dynamically allocated and de-allocated as paths are opened and closed.

A path descriptor is 256 bytes long. It has three sections:

- The first 42 bytes are defined universally for all file managers and device drivers.
- The next 86 bytes are reserved for and defined by each type of file manager for file pointers, permanent variables, etc.
- The last 128 bytes constitute the option area used for the path's operating parameters. This area can be inspected or changed by the user. The variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module. The file manager may also initialize certain variables at the end of the initialization table section so that they may be inspected. The values in this table may be examined using I$GetStt or changed using I$SetStt by applications using the SS_Opt code. The file manager protects some values to prevent inappropriate changes.

The universal path descriptor fields are described below. Each file manager chapter contains definitions of the option area specific to that manager.

Offset	Name	Maintained By	Description
$00	PD_PD	Kernel	Path Number
$02	PD_MOD	Kernel	Access Mode (R W E S D)
$03	PD_CNT	Kernel	Number of Paths using this PD (obsolete)
$04	PD_DEV	Kernel	Address of Related Device Table Entry
$08	PD_CPR	Kernel	Requester's Process ID
$0A	PD_RGS	Kernel	Address of Caller's MPU Register Stack
$0E	PD_BUF	File Manager	Address of Data Buffer
$12	PD_USER	Kernel	Group/User ID of Original Path Owner
$16	PD_PATHS	Kernel	List of Open Paths on Device
$1A	PD_COUNT	Kernel	Number of Paths using this PD
$1C	PD_LProc	Kernel	Last Active Process ID
$20	PD_ErrNo	File Manager	Global "errno" for C language file managers
$24	PD_SysGlob	File Manager	System global pointer for C language file managers
$2A	PD_FST	File Manager	File Manager Working Storage
$80	PD_OPT	Driver/File Man.	Option Table

Name	Description
PD_PD	**Path Number** The path number assigned by the kernel to the open path associated with this descriptor.
PD_MOD	**Access Mode (R W E S D)** The file access mode specified by the I/O request. It may be any combination of the following: bit 0: Set if read access. bit 1: Set if write access. bit 2: Set if executable access. bit 6: Set if single-user access (non-sharable). bit 7: Set if directory file access. All other bits are reserved.
PD_CNT	**Number of Paths using this PD (obsolete)**
PD_DEV	**Address of Related Device Table Entry** The address of the device table entry associated with this path.
PD_CPR	**Requester's Process ID** The process ID of the process originating the I/O request.
PD_RGS	**Address of Caller's MPU Register Stack** The address of the originating process's MPU register stack. This pointer can be used to read or write the registers of the calling process.
PD_BUF	**Address of Data Buffer** This is the address of the data buffer associated with the current I/O operation. It may be a buffer created by the file manager or a pointer directly to an application's buffer.
PD_USER	**Group/User ID of Original Path Owner** The group/user ID of the process which created this path.
PD_PATHS	**List of Open Paths on Device** This field is used to link this descriptor into a circular, singly-linked list of paths open to this device.

Name	Description
PD_COUNT	**Number of Paths using this PD** The number of open paths using this path descriptor. This is set to one when the first path is opened. Using I$Dup to open paths increments this counter.
PD_LProc	**Last Active Process ID** The process ID of the most recent process to perform I/O on this path.
PD_ErrNo	**Global "errno" for C language file managers** This field is available for C language file managers to implement as they see fit.
PD_SysGlob	**System global pointer for C language file managers** This field is available for C language file managers to implement as they see fit.
PD_FST	**File Manager Working Storage** Reserved for and defined by the file manager.
PD_OPT	**Option Table** A 128-byte option area used for the path's operating parameters that you can inspect or change. These variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module. The file manager may also initialize certain variables at the end of the initialization table so that they may be inspected. The values in this table may be examined using I$GetStt or changed using I$SetStt by applications using the SS_Opt code. The file manager protects some values to prevent inappropriate changes.

File Managers

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

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

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

NOTE: I/O system modules must have the following module attributes:

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

Four file managers are usually included in an OS-9 system:

RBF (Random Block File Manager)
Operates random-access, block-structured devices such as disk systems.

SCF (Sequential Character File Manager)
Used with single-character-oriented devices such as CRT or hard-copy terminals, printers, and modems.

SBF (Sequential Block File Manager)
Used with sequential block-structured devices such as tape systems.

PIPEMAN (Pipe File Manager)
Supports interprocess communication through memory buffers called pipes.

File Manager Organization

A file manager is a collection of major subroutines accessed through an offset table. The table contains the starting address of each subroutine relative to the beginning of the table. The location of the table is specified by the execution entry point offset in the module header. A sample listing of the beginning of a file manager module is shown below.

```
* Sample File Manager
* Module Header declaration
        Type_Lang equ (FlMgr<<8)+Objct
        Attr_Revs equ ((ReEnt+Supstat)<<8)+0

        psect FileMgr,Type_Lang,Attr_Revs,Edition,0,Entry_pt

* Entry Offset Table
Entry_pt dc.w       Create-Entry_pt
     dc.w           Open-Entry_pt
     dc.w           MakDir-Entry_pt
     dc.w           ChgDir-Entry_pt
     dc.w           Delete-Entry_pt
     dc.w           Seek-Entry_pt
     dc.w           Read-Entry_pt
     dc.w           Write-Entry_pt
     dc.w           ReadLn-Entry_pt
     dc.w           WriteLn-Entry_pt
     dc.w           GetStat-Entry_pt
     dc.w           SetStat-Entry_pt
     dc.w           Close-Entry_pt
* Individual Routines Start Here
```

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

(a1) Pointer to Path Descriptor.
(a4) Pointer to current Process Descriptor.
(a5) Pointer to User’s Register Stack; user registers pass/receive parameters as shown in the system call description section.
(a6) Pointer to system Global Data area.

These routines are called in system state.

File Manager I/O Service Requests

The general I/O responsibilities for file managers are described in the following pages. Each file manager chapter contains a description of the specific I/O functions for that manager.

Name	Description
I$ChgDir	On multi-file devices, I$ChgDir searches for a directory file. (The kernel allocates a path descriptor so that I$ChgDir may use I$Open when searching for the directory.) If the directory is located, the file manager saves its address in the caller's process descriptor at P$DIO. I$Open and I$Create begin searching in this directory when the caller's pathlist does not begin with a slash (/) character. File managers that do not support directories return with the carry bit set and an appropriate error code in (d1.w).
I$Close	I$Close ensures that any output to a device is completed (writing out the last buffer if necessary), and releases any buffer space allocated when the path was opened. If required, it may do specific end-of-file processing, such as writing end-of-file records on tapes.
I$Create	I$Create performs the same function as I$Open. If the file manager controls multi-file devices, a new file is created. File managers that do not support multi-file devices usually consider I$Create synonymous with I$Open.
I$Delete	Multi-file device managers usually perform a directory search that is similar to I$Open. Once found, the file name is removed from the directory. Any media space that was in use by the file is returned to the free media pool.
I$GetStt	I$GetStt is a wild-card call designed to determine the status of various features of a device (or file manager) that are not generally device independent. The file manager may perform some specific function such as obtaining the size of a file. Status calls that are unknown to the file manager are passed to the driver to provide a further means of device independence.
I$MakDir	I$MakDir creates a directory file on multi-file devices. File managers that are incapable of supporting directories return with the carry bit set and an unknown service error code in (d1.w).
I$Open	I$Open opens a file on a particular device. This typically involves allocating required buffers, initializing path descriptor variables, and parsing the path name. If the file manager controls multi-file devices, directory searching is performed to locate the specified file.

Name	Description
I$Read	I$Read returns the number of bytes requested to the user's data buffer. If no further data is available, an EOF error is returned. I$Read generally performs no editing on data. Usually, a file manager calls the device driver to read the data into a buffer. The buffer may be an internal buffer maintained by the file manager or it may be the application's buffer. The file manager chooses the appropriate buffer for the driver to use. If an internal buffer is used, the data is then copied into the user's data area.
I$ReadLn	I$ReadLn differs from I$Read in two respects. First, I$ReadLn is expected to terminate when the first end-of-record character (carriage return) is encountered. Second, I$ReadLn performs any input editing that is appropriate for the device. Typically, I$ReadLn uses an internal buffer when calling the driver and copies the data from the buffer into the user's data area.
I$Seek	File managers that support random access devices use I$Seek to position file pointers of the already open path to the specified byte. This is a logical movement and does not necessarily affect the physical device. If the position is beyond the current end-of-file, no error is produced at the time of the I$Seek. File managers that do not support random access usually do nothing during the I$Seek operation, and do not return an error.
I$SetStt	I$SetStt is the same as the I$GetStt function except that it is generally used to set the status of various features of a device (or file manager). The file manager may perform some specific function such as setting the size of a file to a given value. Status calls that are unknown to the file manager are passed to the driver to provide a further means of device independence. For example, an I$SetStt call to format a disk track may behave differently on different types of disk controllers.
I$Write	The I$Write request, like I$Read, generally performs no editing on data. Usually, the I$Read and I$Write routines are nearly identical. The most notable difference is that I$Write uses the device driver's output routine instead of the input routine. Writing past the end-of-file on a device expands the file with new data. RBF and similar random access devices that use fixed-length records (sectors) must often pre-read a sector before writing it unless the entire sector is being written.

Name	Description
I$Writln	I$Writln is the counterpart of I$ReadLn. It calls the device driver to transfer data up to and including the first (if any) end-of-record (carriage return) encountered. Appropriate output editing is also performed. For example, after a carriage return, SCF usually outputs a line-feed character and nulls (if appropriate).

Device Driver Modules

Device driver modules perform basic low-level physical input/output functions. For example, a disk driver's basic function is to read or write a physical sector. The driver is not concerned about files, directories, etc., which are handled at a higher level by the OS-9 file manager.

When written properly, a single physical driver module can support multiple identical hardware interfaces simultaneously. The specific information for each physical interface (port address, initialization constants, etc.) is provided in the device descriptor module.

Driver Module Format

All drivers must conform to the standard OS-9 memory module format. The module type code is Drivr. Drivers should have the system-state bit set in the attribute byte of the module header.

NOTE: I/O system modules must have the following module attributes:

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

A sample assembly language header is shown below:

```
* Module Header

     Type_Lang equ (Drivr<<8)+Objct
     Attr_Revs equ ((ReEnt+Supstat)<<8)+0

     psect Acia,Typ_Lang,Attr_Rev,Edition,0,AciaEnt

* Entry Point Offset Table
     AciaEnt   dc.w    Init    Initialization routine offset
               dc.w    Read    Read routine offset
               dc.w    Write   Write routine offset
               dc.w    GetStat Get dev status routine offset
               dc.w    SetStat Set dev status routine offset
               dc.w    TrmNat  Terminate dev routine offset
               dc.w    Trap    Error handler routine offset (0=none)
```

The M$Exec module header field is the offset to the address of an ***offset table***. This table specifies the starting address of each of the seven driver subroutines relative to the base address of the module.

The M$Mem module header field specifies the amount of local static storage required by the driver. This is the sum of the global I/O storage, the storage required by the file manager, and any variables and tables declared in the driver.

The driver subroutines are called by the associated file manager and the kernel through the offset table, with the exception of the device driver's IRQ routine (if any) which is called directly by the kernel's IRQ polling routines. The driver routines are always executed in system state. Regardless of the device type, the standard parameters listed below are passed to the driver in the corresponding registers. Other parameters may also be passed, depending on the device type and subroutine called. These are described in individual file manager chapters.

INIT and TERM *(called by the kernel):*

(a1) The address of the device descriptor module.
(a2) The address of the driver's static variable storage.
(a4) The address of the process descriptor requesting the I/O function.
(a6) The address of the system global variable storage area.

INIT initializes the device controller hardware and related driver variables as required. INIT also enables device interrupts and adds the device to the system's IRQ polling table, if necessary.

TERM de-initializes the device. It is assumed that the device will not be used again unless re-initialized. TERM also deletes the device from the IRQ polling table and disables interrupts, if necessary.

Refer to Figure 1-3 for a diagram of the I/O system layout during the INIT and TERM routines.

READ, WRITE, GETSTAT and SETSTAT *(called by the file manager):*

(a1) The address of the path descriptor storage.
(a2) The address of the driver's static variable storage.
(a4) The address of the process descriptor requesting the I/O function.
(a5) The address of the caller's register stack image.
(a6) The address of the system global variable storage area.

READ reads one or more standard physical units (a character or sector, depending on the device type). WRITE writes one or more standard physical units (a character or sector, depending on the device type).

GETSTAT returns a specified device status. SETSTAT sets a specified device status.

CAVEAT: The register conventions shown above apply to RBF and SCF. For SBF's READ and WRITE routines, the contents of registers a1 and a5 are undefined. For SBF's GETSTAT and SETSTAT routines, the contents of register a5 are undefined. Other file managers may adopt whatever register conventions are desired.

Refer to Figure 1-4 for a diagram of the I/O system layout during the READ, WRITE, GETSTAT, and SETSTAT routines.

TRAP ***(also known as ERROR; not currently called):***

This entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

IRQ ***(called by the kernel's IRQ polling table handler):***

(a2) The address of the driver's static variable storage.
(a3) The address of the device port.
(a6) The address of the system global variable storage area.

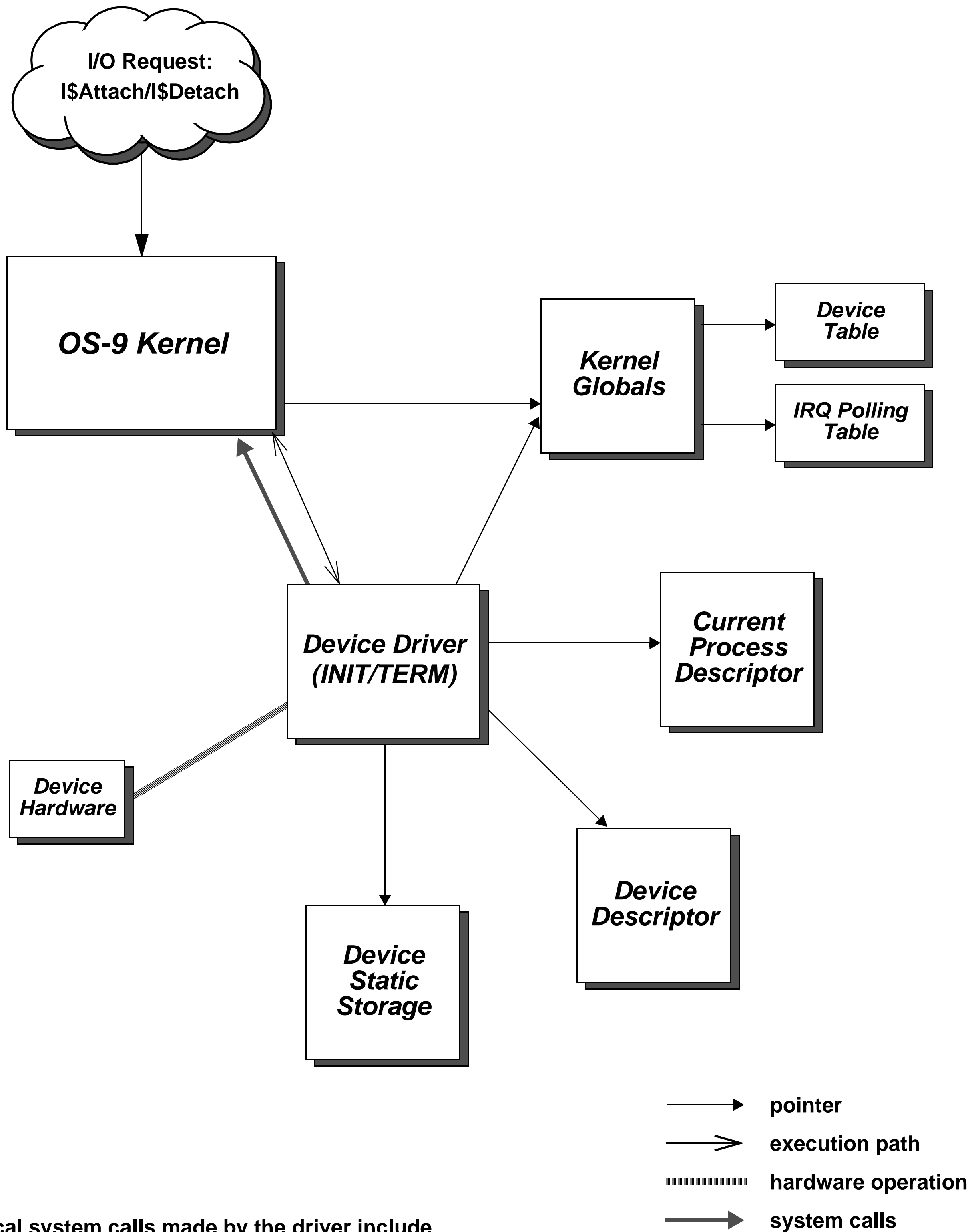
The IRQ subroutine is not called by the file manager, but by the kernel's interrupt polling routine. It communicates with the driver's main section through the static storage and certain system calls.

NOTE: The values passed in a2 and a3 are, by convention, as described above. The values are those that existed in the respective registers when the device was installed on the IRQ polling table (F$IRQ). Register a2 is usually passed to enable the IRQ service routine to access the driver's static storage. Register a3 can have any value desired, because the hardware is never accessed by the kernel's IRQ polling routine.

IRQ may only destroy values in the following registers: d0, d1, a0, a2, a3, and a6. If the interrupt was serviced, IRQ returns the carry bit clear. If not serviced, IRQ returns the carry bit set. This provides the kernel's IRQ polling routine with an indication that it should call the IRQ service routine associated with the next lowest priority device on the vector.

Refer to Figure 1-5 for a diagram of the I/O system layout during the IRQ service routine.

Each subroutine is terminated by a RTS instruction. Error status is returned using the CCR carry bit with an error code returned in register d1.w. For the IRQ service routine, only the CCR carry status is meaningful.



Typical system calls made by the driver include (if any): FIRQ, FSRqMem, F$SRtMem

Figure 1-3: I/O System Layout for INIT/TERM Routines

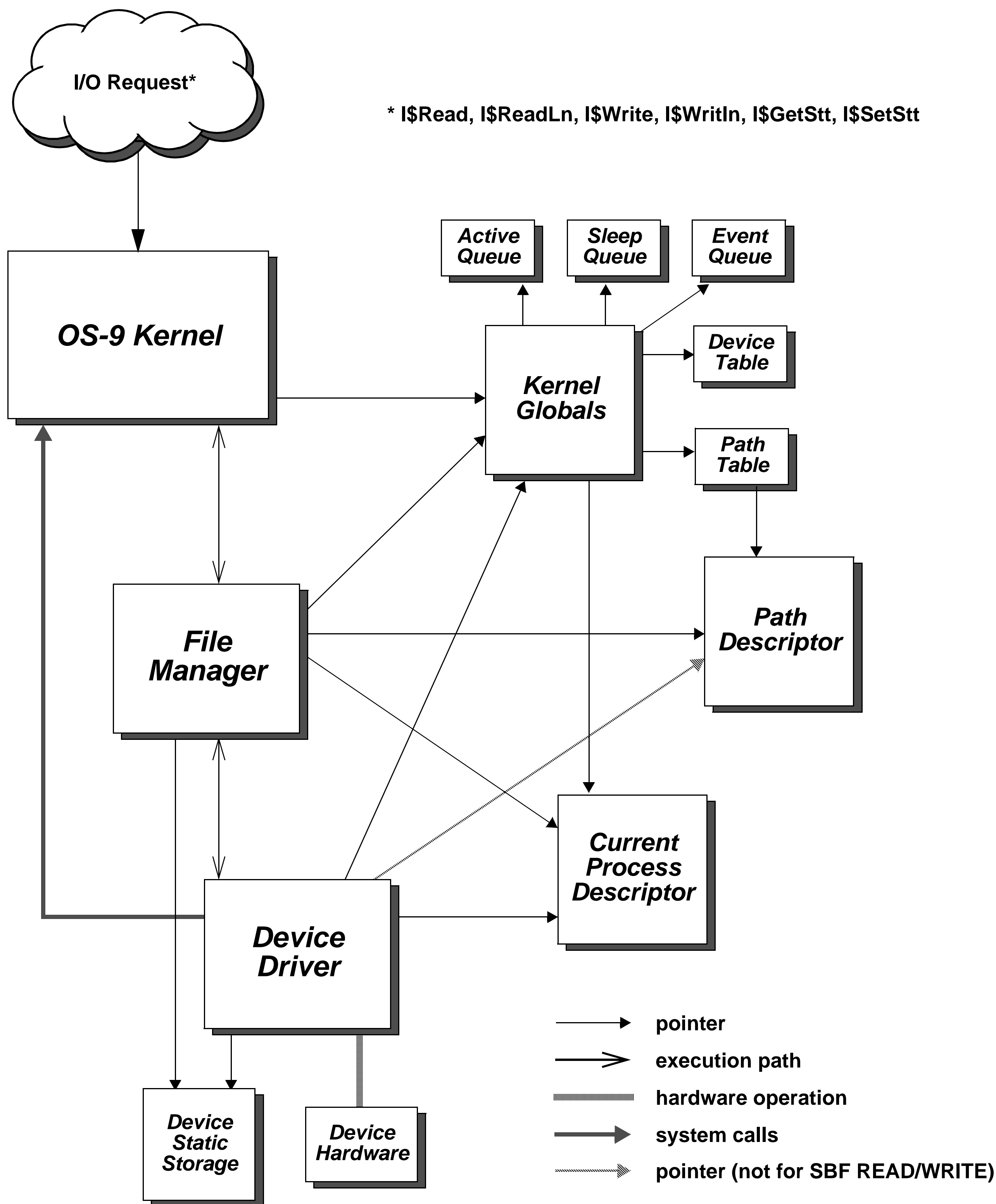


Figure 1-4: I/O System Layout for READ/WRITE/GETSTAT/SETSTAT Routines

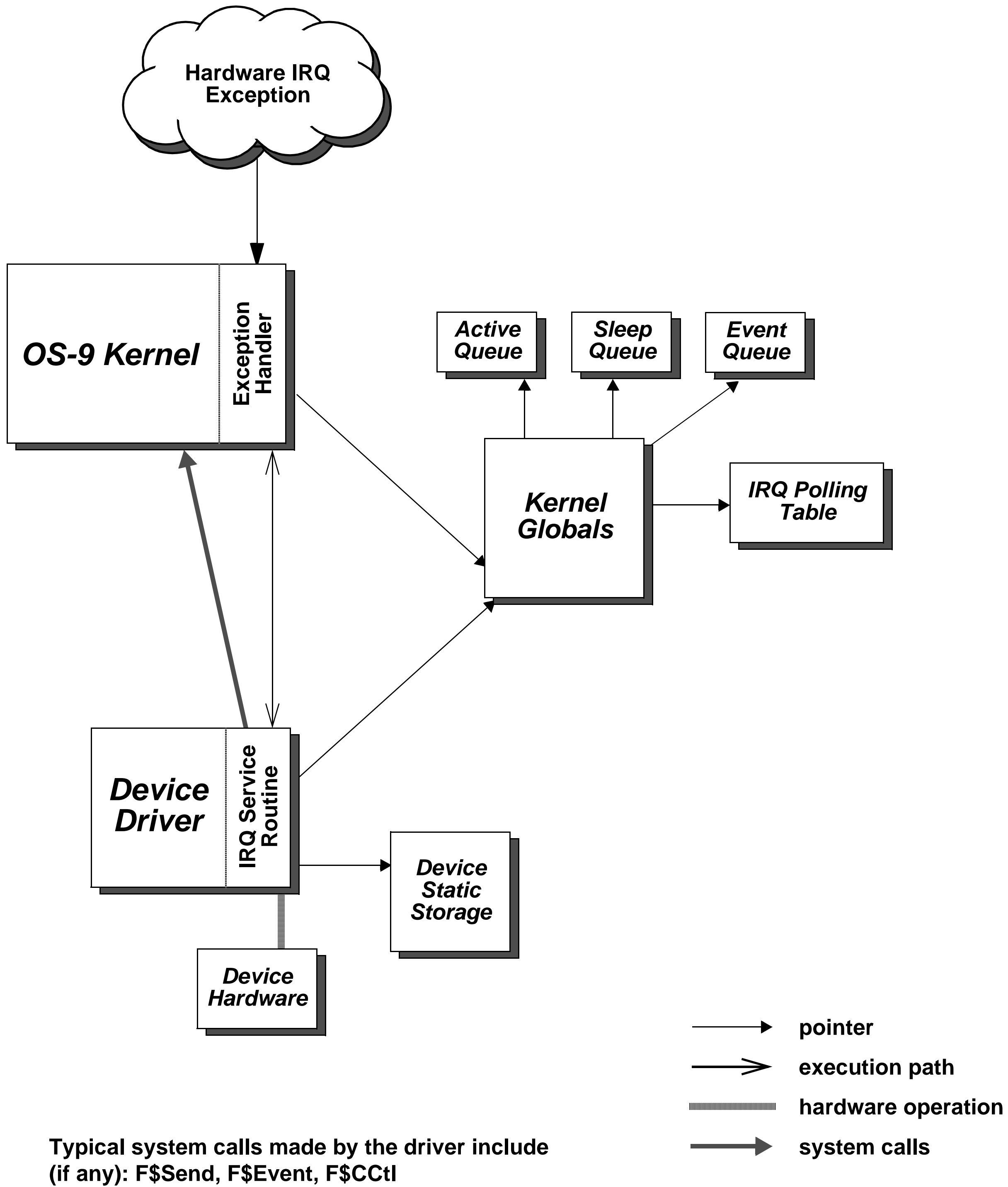


Figure 1-5: I/O System Layout for IRQ Service Routine

Device Drivers That Control Multiple Devices

Properly written re-entrant device drivers can handle more than one physical hardware device. The driver is responsible for isolating the file manager from the specifics of the device interface. The device descriptor tailors the device driver to the actual physical parameters of the hardware in use (for example, port address, interrupt level, etc.). Consequently, adding hardware ports to a system is generally a matter of creating new device descriptors for the new ports.

This section highlights some of the issues that arise when dealing with multi-port/multi-device hardware. It discusses three general types of hardware devices:

- Simple Devices
- Multi-Port Devices
- Multi-Class Devices

Simple Devices

Simple devices provide a single discrete I/O interface, such as a UART (Universal Asynchronous Receiver Transmitter) or a disk controller. If a system has a driver for a specific simple device, instances of that device can be created by building new device descriptors. This can usually be accomplished by editing an existing descriptor and installing the new hardware and descriptor on the system.

The I/O system creates a new ***incarnation*** of the device driver when each device is installed in the system. Each incarnation of the driver has its own static storage area; therefore, the operating parameters for each device are separated from those of similar devices.

The I/O system considers a device a ***new device*** when its device table entry (port address, device descriptor, driver, and file manager) differs from all existing device table entries. When this condition is detected, the new device is added to the I/O system and the device's INIT routine is called.

NOTE: If the new device differs only in that its device descriptor is different (same port address, device driver, and file manager), a new entry is made into the device table, but the INIT routine is *not* called. This is how multi-device, single-controller devices are handled. An example of this is a disk controller supporting more than one drive. The INIT routine is called only once for these devices - at the first I$Attach to any device on this port. In this case, no new incarnation of the driver will occur. The device driver usually discriminates between the devices on the port by means of "logical" devices. For example, a RBF disk controller controlling four drives uses the PD_DRV field of the device descriptor to discriminate between each drive.

Generally, most OS-9 device drivers are expected to handle only one request from a file manager at a time. The mechanism that ensures proper handling of access requests is called I/O Blocking. It is usually performed by the file manager associated with the device, using the V_BUSY variable of the driver's static storage. RBF, SCF, SBF, and PIPEMAN implement I/O Blocking in this manner. Consequently, a driver written to work with one of these file managers need handle only one request at a time. For example, the disk access request to drive 0 of a controller must be completed before RBF makes an access request to drive 1.

I/O blocking *does not* affect *different* devices that use the same driver. This is because the I/O blocking function is performed on a port address basis; V_BUSY is unique to each static storage area. Drivers written for other file managers (for example, NFM) may have to deal with more than request at a time, depending upon how the file manager operates.

Multi-Port Devices

Multi-port devices provide more than one physical I/O channel. If the hardware implementation totally separates the physical I/O channels, the device can be treated as multiple simple hardware devices. An example of this would be a DUART (Dual Universal Asynchronous Receiver Transmitter), a device that provides two separate channels, each with an independent register set. Typically, the only difference between the two device descriptors is the port address. This allows separate incarnations of the driver to control each relevant part of the device.

If, however, the device contains registers that are common between the physical I/O channels, problems can arise with interaction between the incarnations of the driver running on the different ports.

A common example of this situation is the MC68681 DUART. This device contains register sets that are associated with each individual channel and register sets that are common to both channels. The common registers present a problem, in this case, because they are ***write-only*** registers. Each incarnation of the driver needs to manipulate these registers, but has no knowledge of the current state of the ***other-side*** values.

Without a mechanism for sharing these values, manipulation of the common registers can cause a driver to produce inadvertent side effects on the “other” channel. However, you can easily overcome this situation by using one of the following techniques:

OEM Global Storage

The OEM global storage area is a 256-byte area in the system globals of the kernel. This area is provided for system-specific, custom storage allocation. In the case of the common write-only registers, the system can be configured so that memory images of these registers are stored in the OEM global area. When an incarnation of the driver wishes to modify a common register, it must locate the appropriate image stored in RAM, modify it, store the new image back in RAM, and update the hardware. Using this scheme, multiple incarnations of the driver can operate without affecting other incarnations.

The allocation of storage within the OEM global area is system-specific and is usually defined by the individual system designer (OEM). For these types of devices, the device descriptor’s DevCon section is often used to store a pointer to the area allocated for the particular device in the OEM globals.

Using the OEM global area to overcome the problems with multi-port device drivers has the following advantages:

- For the system boot-ROM’s console and communications ports, it allows high-level interrupt-driven drivers to communicate current register values to low-level polled I/O routines in the boot-ROM code. Consequently, correct system operation results when switching the console port between the operating system and the boot ROMs.
- It allows multiple-function devices that share different types of device drivers to communicate current register values between the drivers. The MC68681 DUART is a prime example of this type of device: it has two serial channels and a tick-timer device.

Data Modules

For drivers that only need to communicate between themselves (they do not need to communicate to low-level boot-ROM routines), the use of data modules to store common register values may also be an option. The driver’s INIT routine would dynamically determine the storage area to be used by attempting to create/link the data module. Once the storage has been created/found, then the driver can manipulate the required images in the same way that the OEM global storage variables are accessed.

NOTE: This technique often does not require DevCon values to indicate the storage to be used. Incarnations of the driver only have to agree on the naming convention to adopt when forming the data module’s name. For example, you could use a common part of the port address as part of the name.

Depending upon the system's requirements, other techniques may also be appropriate for managing these situations, such as using the OS-9 event system.

Multi-Class Devices

Creating drivers for I/O systems that support more than one class of I/O device (for example, disk and tape devices on a SCSI bus) presents a different set of problems. However, these problems are generally easy to solve. The most common problems for these devices involve I/O blocking and sensitivity to device class.

Because I/O blocking is usually performed at the file manager level, a common driver supporting two classes of devices (for example, RBF and SBF) may be called by one file manager while running on behalf of another file manager. Therefore, the driver must be written to handle this case or at least provide I/O blocking.

In addition, the layout of the path descriptor options and device static storage is different for each device class. Because the device driver has to be continually sensitive to the device class, the driver is somewhat cumbersome to write. The net effect is attempting to *merge* two separate drivers into a single piece of code.

To simplify these problems, the technique that is usually adopted is to split the driver into ***high-level*** and ***low-level*** functions. The high-level portion of the driver is the actual “device driver,” as it is the module called directly by the file manager. This module deals with all issues related to the device class (for example, static storage allocations, operational characteristics) and the target hardware (for example, command protocols). Once the request has been prepared by the driver, it calls the low-level subroutine module, which is designed to manage the physical interface. The low-level module has no knowledge of the device class or type of operation required. Its function is to manage the I/O requests (with I/O blocking, if necessary) from multiple drivers through the physical interface.

When this technique is adopted, the DevCon section of the device descriptor is usually used as a name string for the low-level module to be used. The individual high-level device drivers can link/unlink to the module and call it, if necessary, during its INIT/TERM routines.

Examples of Multi-Class Devices Using SCSI System Concept

The basic premise of this system is to break the OS-9 driver into separate ***high-level*** and ***low-level*** areas of functionality. This allows different file managers and drivers to talk to their respective devices on the SCSI bus.

The device driver handles the high-level functionality. The device driver is the module that is called directly by the appropriate file manager. Drivers deal with all controller-specific/device-class issues (for example, disk drives on an OMTI5400). They should be written so that they are “portable” code (no MPU/CPU specific code). The high-level drivers prepare the command packets for the SCSI target device and then pass this packet to the low-level subroutine module.

This low-level module passes the command packet (and data if necessary) to the target device on the SCSI bus. The low-level code does NOT concern itself with the contents of the commands/data, it simply performs requests on behalf of the high-level driver. The low-level module is also responsible for co-ordinating all communication requests between the various high-level drivers and itself. The low-level module is often an MPU/CPU specific module, and thus can often be written as an optimized module for the target system.

The device descriptor module contains the name strings for linking the modules together. The file manager and device driver names are specified in the normal way. The low-level module name associated with the device is indicated via the DevCon offset in the device descriptor. This offset pointer points to a string containing the name of the low-level module.

An example system setup shows how drivers for disk and tape devices can be mixed on the SCSI bus without interference:

Hardware Configuration

OMTI5400 Controller:

- Addressed as SCSI ID 6.
- Hard disk addressed as controller's LUN 0.
- Floppy disk addressed as controller's LUN 2.
- Tape drive addressed as controller's LUN 3.

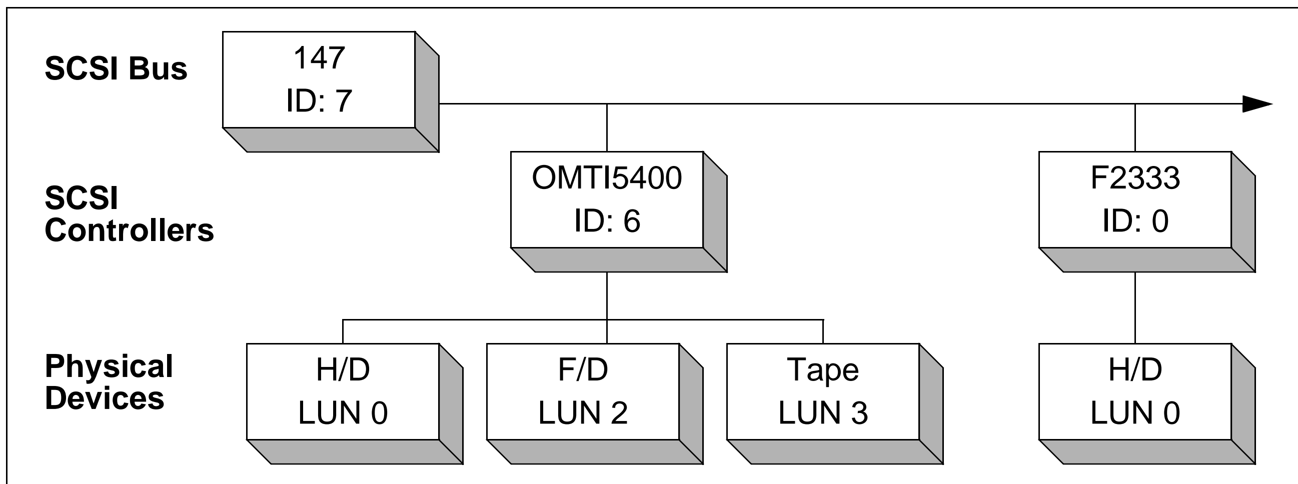
Fujitsu 2333 Hard Disk with Embedded SCSI Controller:

- Addressed as SCSI ID 0.

Host CPU: MVME147

- Uses WD33C93 SBIC Interface chip.
- "Own ID" of chip is SCSI ID 7.

The hardware setup would look like this:



Software Configuration:

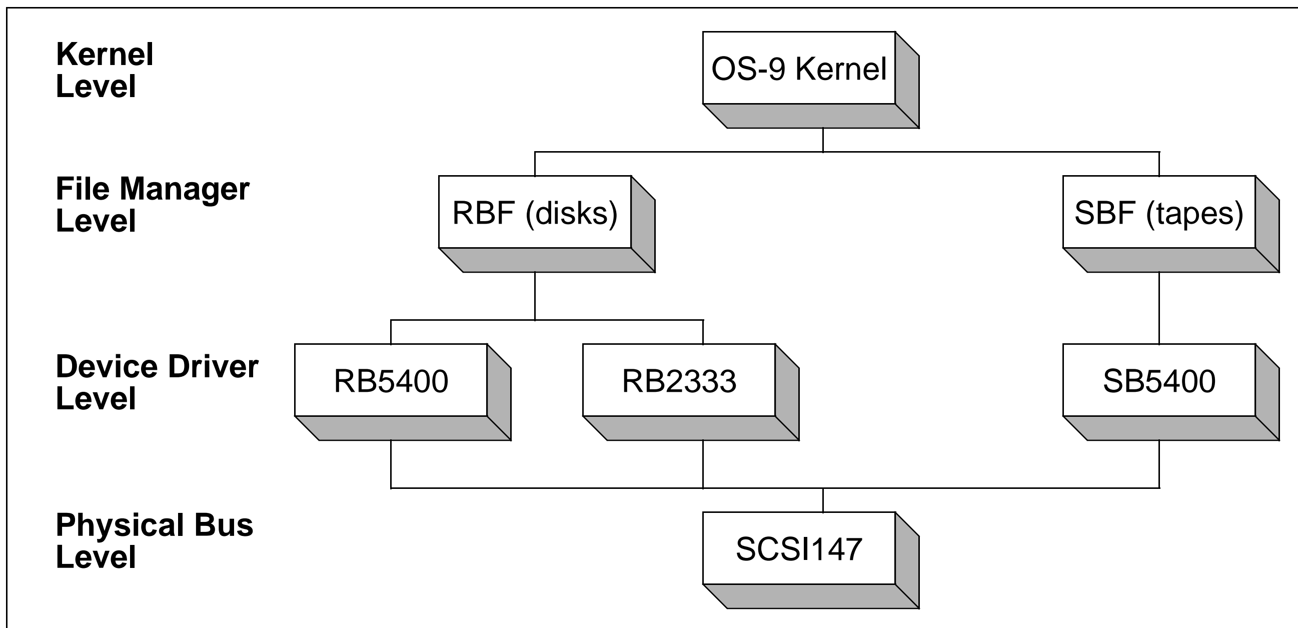
The high-level drivers associated with this configuration are:

Name	Description
RB5400	Handles hard and floppy disk devices on the OMTI5400.
SB5400	Handles tape device on the OMTI5400.
RB2333	Handles hard disk device.

The low-level module associated with this configuration is:

Name	Description
SCSI147	Handles WD33C93 Interface on the MVME147 CPU.

A conceptual map of the OS-9 modules for this system would look like this:



If the guidelines previously given are adhered to, expansion and reconfiguration of the SCSI devices (both in hardware and software) can be easily accomplished. Three examples show how this could be achieved:

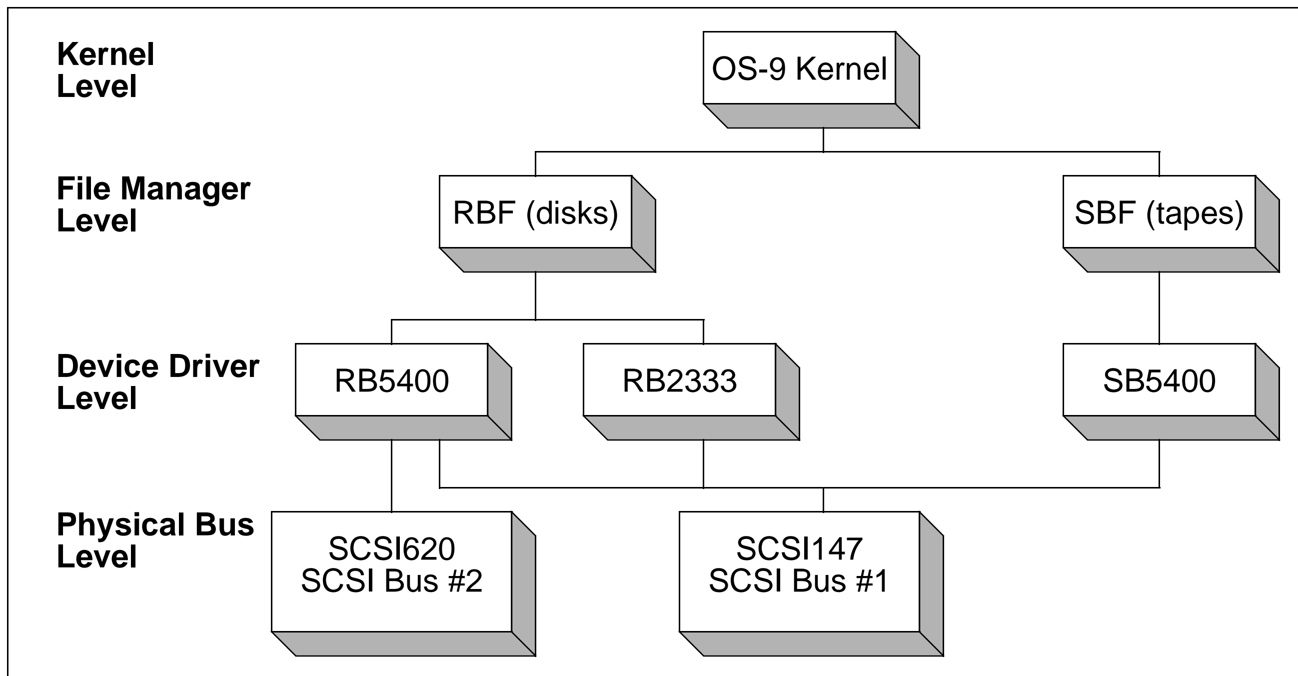
Example One

This example describes the addition of a second SCSI bus using the VME620 SCSI controller. This second bus will have an OMTI5400 controller and associated hard disk.

The VME620 module uses the WD33C93 chip as the SCSI interface controller, but it uses a NEC DMA controller chip. Thus, a new low-level module needs to be created for the VME620 (we will call the module SCSI620). You can create this module by editing the existing files in the SCSI33C93 directory to add the VME620 specific code. This new code would typically be “conditionalized.” A new makefile (such as make.vme620) could then be created to allow production of the final SCSI620 low-level module.

The high-level driver for the new OMTI5400 is already written (RB5400), so you only have to create a new device descriptor for the new hard disk. Apart from any disk parameter changes pertaining to the actual hard disk itself (such as the number of cylinders, etc), you could take one of the existing RB5400 descriptors and modify it so that the DevCon offset pointer points to a string containing SCSI620 (the new low-level module).

The conceptual map of the OS-9 modules for the system would now look like this:

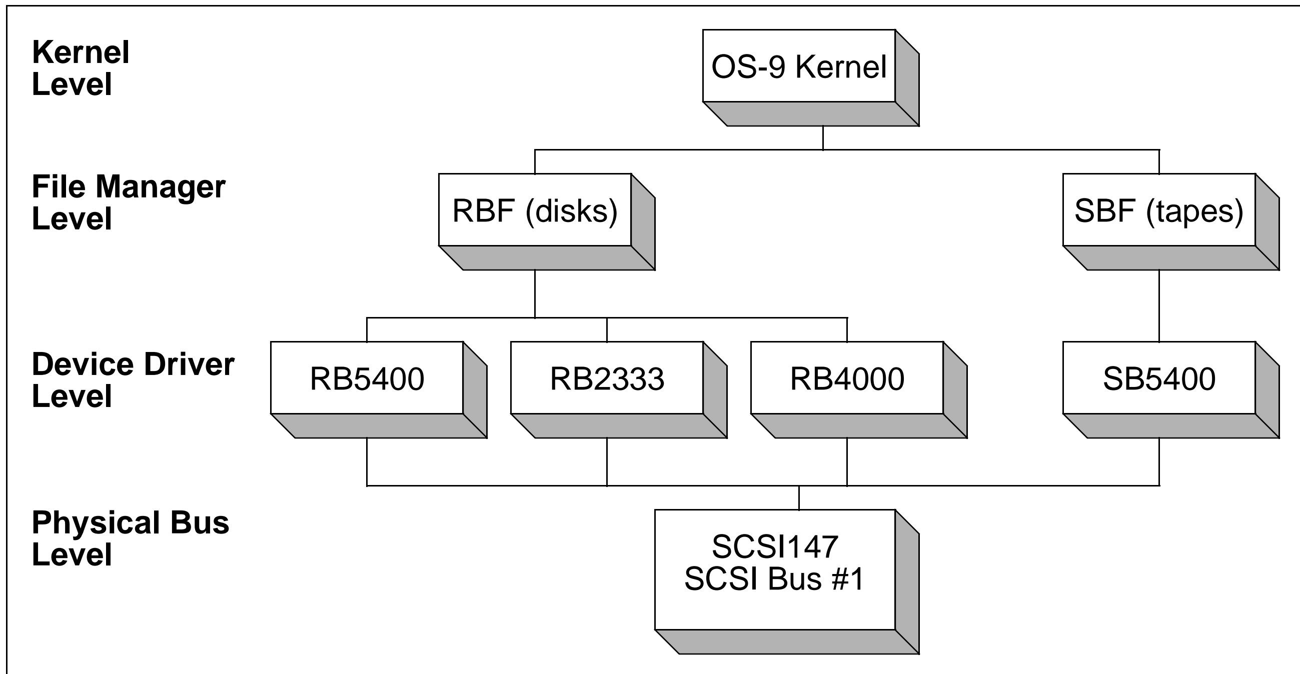


Example Two

This example describes the addition of an Adaptec ACB4000 Disk Controller to the SCSI bus on the MVME147 CPU.

To add a new, different controller to an existing bus, you need to write a new high-level device driver. You would create a new directory (such as RB4000) and write the high-level driver based upon an existing example (such as RB5400). You do not need to write a low-level module, as this already exists. You then need to create your device descriptors for the new devices, with the module name being rb4000 and the low-level module name being scsi147.

The conceptual map of the OS-9 modules for the system would now look like this:



Example Three

Perhaps the most common reconfiguration will occur when adding additional devices of the same type as the existing device. For example, adding an additional Fujitsu 2333 disk to the SCSI bus on the MVME147. To add a similar controller to the bus, all you need to do is create a new device descriptor. There are no drivers to write or modify, as these already exist (RB2333 and SCSI147). The only modifications required would be to take the existing descriptor for the RB2333 device and modify it to reflect the second devices physical parameters (e.g., SCSI ID) and change the actual name of the descriptor itself.

Interrupt Driven I/O

OS-9 is a multi-tasking, real-time operating system. To support these capabilities, I/O devices should be, whenever possible, set up to provide fully interrupt-driven operation. Non-interrupt-driven operation (polled I/O) should only be used for I/O devices that are always ready to read/write data (for example, output to a memory-mapped video RAM). If a driver has to wait for the device to read/write data, then real-time system operation may be affected.

For character-oriented devices (for example, SCF), the controller should be set up to generate an interrupt upon the receipt of an incoming character and at the completion of transmission of an outgoing character. Both the input data and the output data should be buffered in the driver. In the case of block-type devices (for example, RBF, SBF), the controller should be set up to generate an interrupt upon the completion of a block read or write operation. It is usually not necessary for the driver to buffer data because the driver is passed the address of a complete buffer.

Devices are usually added to the system's IRQ polling table when the device is attached (INIT routine) and removed from the IRQ polling table when the device is detached (TERM routine). The device is added and deleted by the driver using the F$IRQ service request. Device drivers for devices that generate multiple vectors (for example, separate receive and transmit interrupts) or hardware ports that have multiple devices (for example, disk controllers with associated DMA device) may have to make multiple F$IRQ calls to add and delete each device in the polling table.

NOTE: The maximum number of devices (device table entries) and interrupting devices (polling table entries) are defined in the initialization module ("init"). These fields (M$DevCnt and M$PollSz) are user adjustable.

The kernel does not place any restrictions on which vectors (M$Vector of the device descriptor) may be used by devices or how many devices may share a vector. If devices share a vector, the priority of the device on the vector is determined by the IRQ polling priority (M$Prior) specified for the device. As a general rule, the system integrator should attempt to allocate one device per vector so that the kernel's IRQ polling table will "vector" to the correct device immediately.

Interrupt-driven drivers generally consist of two separate execution threads: the driver mainline and the interrupt service routine. A typical I/O operation by the driver consists of the following:

¨ Driver mainline (called by file manager) initiates I/O operation and suspends itself.

¦ Device interrupt occurs and IRQ service routine initiates wake-up of driver mainline.

Æ Driver mainline is reactivated and returns to caller.

The synchronization of the driver mainline and IRQ service routine is usually accomplished by one of the following mechanisms:

SIGNALS — The driver suspends itself by sleeping (F$Sleep) and is reactivated when the IRQ service routine sends the driver a signal (F$Send, signal S$Wake). This is the most common method used by interrupt-driven drivers. The interlock between the execution threads is usually done using the static storage variable V_WAKE.

EVENTS — The driver suspends itself by waiting on an event (F$Event), and is reactivated when the IRQ service routine signals the event. The interlock between the execution threads is done via the event values.

The decision whether to use signals or events for interrupt operation should be based on the complexity of the driver. If the driver is simple, (only needs to communicate interrupt occurrences) either method is suitable. If the driver is complicated, (needs to communicate more than one state) the event system is usually preferred. For example, the event system would be more suitable for a SCSI driver that supports multiple devices that can disconnect.

The assignment of a device's physical interrupt level(s) can have a significant impact on system operation. Generally, the smarter the device, the lower its interrupt level can be set. For example, a disk controller that buffers sectors can wait longer for service than a single-character buffered serial port. Usually, the interrupt levels can be assigned according to the system's requirements, but it is recommended that you assign the clock tick device the highest possible level to keep interference with system time-keeping at a minimum.

The following table shows how interrupt levels can be assigned in a typical system:

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

CAVEAT: Level 7 is a non-maskable interrupt. It should not be used by OS-9 I/O devices. A device set at this level can interrupt the kernel during critical system operations. However, level 7 can be used for hardware operations *unknown* to the system (for example, dynamic RAM refreshing).

CAVEAT: Exception conditions (such as a Bus Error) should be avoided when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

DMA I/O and System Caches

Direct Memory Access (DMA) support, if available, significantly improves data transfer speed and general system performance, because the MPU does not have to explicitly transfer the data between the I/O device and memory. Enabling these hardware capabilities is generally a desirable goal, although systems that include cache (particularly data cache) mechanisms need to be aware of DMA activity occurring in the system, so as to ensure that ***stale*** data problems do not arise.

Stale data occurs when another bus master writes to (alters) the memory of the local processor. The bus cycles executed by the other master may not be seen by the local cache/processor. Therefore, the local cache copy of the memory is inconsistant with the contents of main memory.

The system's caching algorithms are controlled by two components of OS-9:

- The Syscache module.
- The Init module.

Syscache Module

The Syscache module is the global mechanism to invoke caching. If this module is present in the bootstrap file, caching will occur in the system. If the module is not found during system startup, all cache functions are disabled.

Default Syscache modules are provided for each class of MPU (for example, the 68020 provides instruction caching, while the 68030 provides instruction and data caching) so as to support the on-chip cache capabilities of the system.

You can integrate off-chip (system specific) caches into the system by having the OEM customize the Syscache module for the CPU module in use.

Init Module

The Init module's Compat variables also play a role in the cache control for the system. You can set flags in these variables to fine-tune the kernel's cache control.

The flags available in the Init module are:

Variable	Bit #	Function		
M$Compat	3	0	=	enable burst mode (68030 systems only)
		1	=	disable burst mode
M$Compat2	0	0	=	external instruction cache is NOT snoopy*
		1	=	external instruction cache is snoopy or absent
	1	0	=	external data cache is NOT snoopy
		1	=	external data cache is snoopy or absent
	2	0	=	on-chip instruction cache is NOT snoopy
		1	=	on-chip instruction cache is snoopy or absent
	3	0	=	on-chip data cache is NOT snoopy
		1	=	on-chip data cache is snoopy or absent
	7	0	=	kernel disables data caches when in I/O
		1	=	kernel DOES NOT disable data caches when in I/O

* snoopy = cache that maintains its integrity without software intervention

Avoiding Stale Data Problems

To ensure that stale data problems do not arise, use the following set of guidelines when writing system code (file managers and device drivers) and setting up the Init module cache flags:

Data-Cache disabling when calling the I/O system

The Init module's M$Compat2 byte controls whether or not the kernel disables the data cache(s) when calling the I/O system. The flag setting are defined as follows:

Bit 7	1	Data caching is on. The kernel does NOT disable data caching when calling the I/O system.
	0	Data caching is off. The kernel disables the data caches while any process is in the I/O system.

The decision to turn the flag ON (and thus keep data caching ON for I/O calls) is made depending upon the following factors. Set the flag ON if one of the following conditons is true:

- If no DMA activity occurs in the I/O system.
- If the system cache hardware keeps the caches coherent when DMA activity occurs. **NOTE:** The hardware coherency of the caches is indicated to the kernel via other flags in M$Compat2.

- If the caches do not maintain coherency, and DMA drivers exist in the system, and they ensure that data cache flushes occur (the driver's perform F$CCtl calls).

If none of the above situations can be guaranteed, stale data situations may arise (often at unexpected times) and system behavior may be affected. In these cases, leave the flag OFF so that data cache disabling will occur.

Indication of Cache Coherency

The M$Compat2 variable also has flags that indicate whether or not a particular cache is coherent. Flagging a cache as coherent (when it is) allows the kernel to ignore specific cache flush requests, using F$CCtl. This provides a speed improvement to the system, as unneccessary system calls are avoided and the caches are only explicitly flushed when absolutely necessary.

NOTE: An absent cache is inherently coherent, so it is important to indicate absent (as well as coherent) caches.

Device Drivers that use DMA can determine the need to flush the data caches using the kernel's system global variable, D_SnoopD. This variable is set to a non-zero value if BOTH the on-chip and external data caches are flagged as snoopy (or absent). Thus a driver can inspect this variable, and determine whether a call to F$CCtl is required or not.

Address Translation and DMA Transfers

In some systems, the local address of memory is not the same as the address of the block as seen by other bus masters. This causes a problem for DMA I/O drivers, in that the driver is passed the local address of a buffer, but the DMA device itself requires a different address.

The Init module's "colored memory" lists provide a means to setup the local/external addressing map for the system. This mapping can be determined by device drivers in a generic manner using the F$Trans system call. Thus, you should write drivers that have to deal with DMA devices in a manner that ensures the code will run on any address mapping situation. You can do this using the following algorithm:

> If a pointer must be passed to an external bus master, a call should be made to the kernel's F$Trans system call.
>
> If F$Trans returns an "unknown service request" error, no address translation is in effect for the system and the driver can pass the unmodified address to the other master.
>
> If F$Trans returns any other error, something is seriously wrong. The driver should return the error to the file manager.
>
> If F$Trans returns no error, the driver should check that the size returned for the translated block is the same as the size requested. If so, the address can be passed to the other master. If not, the driver can adopt one of two strategies:
>
> - Refuse to deal with "split blocks", and return an error to the file manager.
> - Break up the transfer request into multiple calls to the other master, using multiple calls to F$Trans until the original block has been fully translated.
>
> The first method proposed above (refuse split blocks) is the usual method adopted by drivers, as the current version of the kernel does allocate memory blocks that span address translation factors.

If drivers adopt these methods, the driver will function irrespective of the address translation issues. Boot drivers can also deal with this issue in a similar manner by using the TransFact global label in the bootstrap ROM.

End of Chapter 1

Random Block File Manager (RBF)

RBF General Description

The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service requests to random-access devices. RBF can handle any number or type of such devices simultaneously (for example, large hard disk systems, small floppy systems, RAM disk systems, etc.) and is responsible for maintaining the logical file structure.

Because RBF is designed to support a wide range of devices with different performance and storage capacities, it is highly parameter-driven.

Some of the physical parameters RBF uses are stored on the media itself. On disk systems, this information is written on the first few sectors of track number zero. The device drivers also use this information, particularly the media format parameters stored on sector 0. These parameters are written by the format program when it initializes and tests the media. Storage systems that initialize themselves without using format are responsible for establishing the initial file structure of the media themselves (for example, RAM disk systems).

The following I/O service requests are handled by RBF:

I$ChgDir	I$Close	I$Create	I$Delete	I$GetStt
I$MakDir	I$Open	I$Read	I$ReadLn	I$Seek
I$SetStt	I$Write	I$Writln		

The following I/O service requests do not call RBF:

I$Attach	I$Detach	I$Dup

RBF I/O Service Requests

When a process makes one of the following system calls to an RBF device, RBF executes the file manager functions described for that call.

I$ChgDir RBF performs the following functions:

- **Sets the directory bit in the access mode**
- **Calls RBF's Open routine to search the specified pathlist**
- **If accessible, updates the appropriate default P$DIO pointer in the process descriptor**
- **Closes the path opened by the Open routine**

I$Close RBF performs the following functions:

- **Flushes any data that has not yet been written to the disk**
  (any partial block of data left from a previous write call)
- **Checks the use count in the path descriptor**
  If the use count is non-zero, no further action is taken. Otherwise, RBF:
  - **Updates the file descriptor, if necessary**
  - **Trims the file size, if necessary**
  - **Calls the device driver with the SS_Close SetStat**
    (ignores any returned errors)

I$Create RBF performs the following functions:

- **Initializes the path descriptor to the default option values**
- **Searches directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **If the file is found, RBF will return an error**
- **Creates a directory entry for the new file**
  If there is no free space in the directory, it is expanded to make room for the new entry.
- **Creates and initializes a file descriptor for the file**
  If an initial size allocation has been specified, RBF attempts to allocate the specified amount of disk space for the file. If not specified, the first I$Write expands the file.
- **Calls the device driver with an SS_Open SetStat**
  RBF ignores E$UnkSvc errors, but aborts I$Create on any other error.

I$Delete RBF performs the following functions:

- **Initializes the path descriptor to the default option values**
- **Searches any directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **Checks the permission attributes of the file**
  The file's directory bit (dirbit) must be turned off using the SS_Attr SetStat call before I$Delete is called. To delete the file, the user must have permission to write to the file and there cannot be other open paths to the file. An error is returned if these conditions are not met.
- **Decrements link count in file descriptor**
  If the link count becomes zero, all disk space associated with the file is returned. This includes the file's file descriptor block. If the link count is non-zero, no disk space is returned.
- **Removes directory entry for the file**

I$GetStt

Refer to the I$GetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the RBF supported I$GetStt functions:

SS_EOF	Check for end-of-file condition.
SS_FD	Get a copy of the file descriptor.
SS_FDInf	Get a copy of a specified file descriptor.
SS_Opt	Read path descriptor options.
SS_Pos	Determine file position.
SS_Ready	Test for data ready.
SS_Size	Determine file's size.

All other GetStat calls are passed to the driver.

I$MakDir

RBF performs a Create operation with the directory bit set for the file access mode. If the Create succeeds, RBF creates directory entries for "." and ".." in the new directory file and then closes the path opened by Create.

I$Open

RBF performs the following functions:

- **Initializes the path descriptor with the default option values**
- **Searches any directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **Checks the permission attributes of the file**
  If the user does not have permission to open the file in the access mode requested, an error is returned.
- **Updates the last modified date in the file descriptor, if open for writing**

- **Calls the device driver with the SS_Open SetStat**
  RBF ignores E$UnkSvc errors, but aborts the I$Open on any other error.

I$Read — RBF performs the following functions:

- **Attempts to acquire a record lock of the section of the file**
  If the record is in use, RBF waits for the time specified by the SS_Ticks SetStat call. This value defaults to zero, resulting in an indefinite sleep until the record becomes available.
- **Determines if there is data left to read in the file**
  If there is none, an end-of-file error (E$EOF) is returned.
- **Calls the driver to read the data, as needed by RBF**
  Complete blocks of data are transferred directly into the process's buffer. Partial blocks are read into a buffer maintained by RBF after which the portion of data requested from those blocks are copied into the calling process's buffer. If the requested data was found in a buffer from a previous read, RBF copies the data to the calling process's buffer without calling the driver.

If the file is open only for reading, the record lock on the requested section is released immediately. If the file is open for update, the record remains locked. A read of 0 bytes, a read of a different section, or an I$Write releases the current section's record lock.

I$ReadLn — I$ReadLn is similar to I$Read, except that RBF maintains a buffer to read data into using single sector reads. It searches the data until it locates the first end-of-record character (carriage return), or reads the number of bytes requested, whichever comes first. It copies the read buffer into the process's buffer as necessary.

If the file is open only for reading, the record lock on the requested section is released immediately. If the file is open for update, the record remains locked. A read of 0 bytes, a read of a different section, or an I$Write releases the current section's record lock.

NOTE: The portion of the file that is record locked begins at the file position from where the I$ReadLn call was made and continues through the number of bytes requested, regardless of whether the EOR is found earlier.

I$Seek — RBF sets the current position in the path descriptor to the specified position. If RBF's internal buffer contains a sector which contains modified data, and the new position is not in that sector, the driver is called to write that sector before the current position in the path descriptor is updated.

I$SetStt — Refer to the I$SetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the RBF supported I$SetStt functions:

SS_Attr	Set file's permission attributes.
SS_FD	Write some file descriptor information.
SS_Lock	Record lock a portion of the file.
SS_Opt	Write the path descriptor options.
SS_RsBit	Reserve bitmap sector.
SS_Size	Set the file's size.
SS_Ticks	Set the record locking time-out value.

All other SetStat calls are passed to the driver.

NOTE: SS_Opt is passed to the driver after processing by RBF. If an unknown service request error (E$UnkSvc) is returned by the driver, it is ignored.

I$Write

RBF performs the following functions:

- **Attempts to acquire a record lock of the section of the file**
  If the record is in use, RBF waits for the time specified by the SS_Ticks SetStat call. This value defaults to zero which results in an indefinite sleep until the record becomes available.
- **Expands the file, if necessary**
- **Calls the driver to write the data, as needed**
  Complete blocks of data are transferred directly from the process's buffer. Partial blocks are copied into a buffer maintained by RBF. This data is written after a subsequent write fills the buffer, or a seek, read, or write is done to another portion of the file, or when the file is closed.

Any active record lock is released once the section has been written. A write of zero bytes also releases the record lock.

I$WritIn

I$WritIn is similar to I$Write, except that RBF searches the calling process's data buffer for an end-of-record character (carriage return). If one is found, only the data up to that end-of-record character is written. If no end-of-record character is found, RBF writes the number of bytes specified by the caller.

Any active record lock is released once the section has been written. A write of 0 bytes also releases the record lock.

RBF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for RBF devices. The table immediately follows the standard device descriptor module header fields. The size of the table is defined in the M$Opt field.

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Class
$49	PD_DRV	Drive Number
$4A	PD_STP	Step Rate
$4B	PD_TYP	Device Type
$4C	PD_DNS	Density
$4D		Reserved
$4E	PD_CYL	Number of Cylinders
$50	PD_SID	Number of Heads/Sides
$51	PD_VFY	Disk Write Verification
$52	PD_SCT	Default Sectors/Track
$54	PD_T0S	Default Sectors/Track 0
$56	PD_SAS	Segment Allocation Size
$58	PD_ILV	Sector Interleave Factor
$59	PD_TFM	DMA Transfer Mode
$5A	PD_TOffs	Track Base Offset
$5B	PD_SOffs	Sector Base Offset
$5C	PD_SSize	Sector Size (in bytes)
$5E	PD_Cntl	Control Word
$60	PD_Trys	Number of Tries
$61	PD_LUN	SCSI Unit Number of Drive
$62	PD_WPC	Cylinder to Begin Write Precompensation
$64	PD_RWR	Cylinder to Begin Reduced Write Current
$66	PD_Park	Cylinder to Park Disk Head
$68	PD_LSNOffs	Logical Sector Offset
$6C	PD_TotCyls	Number of Cylinders On Device
$6E	PD_CtrlrID	SCSI Controller ID
$6F	PD_Rate	Data transfer/Disk Rotation Rates
$70	PD_ScsiOpt	SCSI Driver Options Flags
$74	PD_MaxCnt	Maximum Transfer Count

NOTE: In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, you must make the following adjustment: (M$DTyp - PD_OPT)

For example, to access the drive number in a device descriptor, use PD_DRV + (M$DTyp - PD_OPT). To access the drive number in the path descriptor, use PD_DRV. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Name	Description
PD_DTP	**Device Type** This field is set to one for RBF devices.
PD_DRV	**Drive number** This field is used to associate a one-byte logical integer with each drive that a driver/controller will handle. Each controller's drives should be numbered 0 to n-1 (n is the maximum number of drives the controller can handle and is set into V_NDRV by the driver's INIT routine). This number defines which drive table the driver and RBF access for this device. RBF uses this number to set up the drive table pointer (PD_DTB). Prior to initializing PD_DTB, RBF verifies that PD_DRV is valid for the driver by checking for a value less than V_NDRV in the driver's static storage. If not, RBF aborts the path open and returns an error. On simple hardware, this logical drive number is often the same as the physical drive number.
PD_STP	**Step rate** This field contains a code that sets the drive's head-stepping rate. To reduce access time, the step rate should be set to the fastest value of which the drive is capable. For floppy disks, the following codes are commonly used:

Step Code	5" Disks	8" Disks
0	30ms	15ms
1	20ms	10ms
2	12ms	6ms
3	6ms	3ms

For hard disks, the value in this field is usually driver dependent.

PD_TYP **Disk Type**

Defines the physical type of the disk, and indicates the revision level of the descriptor.

If bit 7 = 0, floppy disk parameters are described in bits 0-6:

bit 0:	0 =	5 1/4" floppy disk (pre-Version 2.4 of OS-9)
	1 =	8" floppy disk (pre-Version 2.4 of OS-9)
bits 1-3:	0 =	(pre-Version 2.4 descriptor) Bit 0 describes type/rates.
	1 =	8" physical size
	2 =	5 1/4" physical size
	3 =	3 1/2" physical size
	4-7:	Reserved
bit 4:	Reserved	
bit 5:	0 =	Track 0, side 0, single density
	1 =	Track 0, side 0, double density
bit 6:	Reserved	

If bit 7 = 1, hard disk parameters are described in bits 0-6:

bits 0-5:	Reserved	
bit 6:	0 =	Fixed hard disk
	1 =	Removable hard disk

PD_DNS **Disk Density ***

Indicates the hardware density capabilities of a floppy disk drive:

bit 0:	0 =	Single bit density (FM)
	1 =	Double bit density (MFM)
bit 1:	1 =	Double track density 96 TPI/135 TPI)
bit 2:	1 =	Quad track density (192 TPI)
bit 3:	1 =	Octal track density (384 TPI)

PD_CYL **Number of cylinders (tracks) ***

Indicates the logical number of cylinders per disk. Format uses this value, PD_SID, and PD_SCT to determine the size of the drive. PD_CYL is often the same as the physical cylinder count (PD_TotCyls), but can be smaller if using partitioned drives (PD_LSNOffs) or track offsetting (PD_TOffs).

If the drive is an autosize drive (PD_Cntl), format ignores this field.

*** These parameters are format specific.**

Name	Description
PD_SID	**Heads or Sides *** This field indicates the number of heads for a hard disk (Heads) or the number of surfaces for a floppy disk (Sides). If the drive is an autosize drive (PD_Cntl), format ignores this field.
PD_VFY	**Verify Flag** This field indicates whether or not to verify write operations. 0 = verify disk write 1 = no verification **NOTE:** Write verify operations are generally performed on floppy disks. They are not generally performed on hard disks because of the lower soft error rate of hard disks.
PD_SCT	**Default sectors/track*** This field indicates the number of sectors per track. If the drive is an autosize drive (PD_Cntl), format ignores this field.
PD_T0S	**Default Sectors/Track (Track 0) *** This field indicates the number of sectors per track for track 0. This may be different than PD_SCT (depending on specific disk format). If the drive is an autosize drive (PD_Cntl), format ignores this field.
PD_SAS	**Segment allocation size** Indicates the default minimum number of sectors to be allocated when a file is expanded. Typically, this is set to the number of sectors on the media track (for example, 8 for floppy disks, 32 for hard disks), but can be adjusted to suit the requirements of the system.
PD_ILV	**Sector interleave factor *** Indicates the sequential arrangement of sectors on a disk (for example, 1, 2, 3... or 1, 3, 5...). For example, if the interleave factor is 2, the sectors are arranged by 2's (1, 3, 5...) starting at the base sector (see PD_SOffs). **NOTE:** Optimized interleaving can drastically improve I/O throughput. **NOTE:** PD_ILV is typically only used when the media is formatted, as format uses this field to determine the default interleave. However, when the media format occurs (I$SetStat, SS_WTrk call), the desired interleave is passed in the parameters of the call.

.

*** These parameters are format specific.**

Name	Description
PD_TFM	**DMA (Direct Memory Access) transfer mode** Indicates the mode of transfer for DMA access, if the driver is capable of handling different DMA modes. Use of this field is driver dependent.
PD_TOffs	**Track base offset *** This field is the offset to the first accessible physical track number. Track 0 is not always used as the base track because it is often a different density.
PD_SOffs	**Sector base offset *** This field is the offset to the first accessible physical sector number on a track. Sector 0 is not always the base sector.
PD_SSize	**Sector Size** Indicates the physical sector size in bytes. The default sector size is 256. Depending upon whether the driver supports non-256 byte logical sector sizes (that is, a variable sector size driver), the field is used as follows:

- **Variable Sector Size Driver**
  If the driver supports variable logical sector sizes, RBF inspects this value during a path open (specifically, after the driver returns "no error" on the SS_VarSect GetStat call) and uses this value as the *logical* sector size of the media. This value is then copied into PD_SctSiz of the path descriptor options section, so that applications programs can know the logical sector size of the media, if required. RBF supports logical sector sizes from 256 bytes to 32,768 bytes, in integral binary multiples (256, 512, 1024, etc.).

  During the SS_VarSect call, the driver can validate or update this field (or the media itself) according to the driver's conventions. These typically are:

  ¿ If the driver can dynamically determine the media's sector size, and PD_SSize is passed in as 0, the driver updates this field according to the current media setting.

  ¡ If the driver can dynamically set the media's sector size, and PD_SSize is passed in as a non-zero value, the driver sets the media to the value in PD_SSize (this is typical when re-formatting the media).

  ¬ If the driver cannot dynamically determine or set the media sector size, it usually validates PD_SSize against the supported sector sizes, and returns an error (E$SectSiz) if PD_SSize contains an invalid value.

*** These parameters are format specific.**

- **Non-Variable Sector Size Driver**
  If the driver does not support variable logical sector sizes (that is, logical sector size is fixed at 256 bytes), RBF ignores PD_SSize. In this case, PD_SSize can be used to support deblocking drivers that support various physical sector sizes.

  NOTE: A non-variable sector sized driver is defined as a driver which returns the E$UnkSvc error for GetStat (SS_VarSect).

PD_Cntl **Device Control Word**

Indicates options that reflect the capabilities of the device. These options may be set by the user, as follows:

bit 0: 0 = Format enable
1 = Format inhibit

bit 1: 0 = Single-Sector I/O
1 = Multi-Sector I/O capable

bit 2: 0 = Device has non-stable ID
1 = Device has stable ID

bit 3: 0 = Device size determined from descriptor values
1 = Device size obtained by SS_DSize GetStat call

bit 4: 0 = Device cannot format a single track
1 = Device can format a single track

bit 5-15: Reserved

Name	Description
PD_Trys	**Number of Tries** Indicates whether a driver should try to access the disk again before returning an error. Depending upon the driver in use, this field may be implemented as a flag or a retry counter:

Value	Flag	Counter
0	retry ON	default number of retries
1	retry OFF	no retries
other	retry ON	specified number of retries

Drivers that work with controllers that have error correcting functions (for example, E.C.C. on hard disks) should treat this field as a flag so they can set the controller's error correction/retry functions accordingly.

When formatting media, especially hard disks, the format-enabled descriptor should set this field to one (retry OFF) to ensure that marginal media sections are marked out of the media free space.

Name	Description
PD_LUN	**Logical Unit Number of SCSI Drive** Used in the SCSI command block to identify the logical unit on the SCSI controller. To eliminate allocation of unused drive tables in the driver static storage, this number may be different from PD_DRV. PD_DRV indicates the logical number of the drive to the driver, that is, the drive table to use. PD_LUN is the physical drive number on the controller.
PD_WPC	**First Cylinder to Use Write Precompensation** Indicates the cylinder to begin write precompensation.
PD_RWR	**First Cylinder to Use Reduced Write Current** Indicates the cylinder to begin reduced write current.

Name	Description
PD_Park	**Cylinder Used to Park Head** Indicates the cylinder at which to park the hard disk's head when the drive is shut down. Parking is usually done on hard disks when they are shipped or moved and is implemented by the SS_SQD SetStat to the driver.
PD_LSNOffs	**Logical Sector Offset** The offset to use when accessing a partitioned drive. The driver adds this value to the logical block address passed by RBF prior to determining the physical block address on the media. Typically, using PD_LSNOffs is mutually exclusive to using PD_TOffs.
PD_TotCyls	**Total Cylinders on Device** Indicates the actual number of physical cylinders on a drive. It is used by the driver to correctly initialize the controller/drive. PD_TotCyls is typically used for physical initialization of a drive that is partitioned or has PD_TOffs set to a non-zero value. In this case, PD_CYL denotes the *logical* number of cylinders of the drive. If PD_TotCyls is zero, the driver should determine the physical cylinder count by using the sum of PD_CYL and PD_TOffs.
PD_CtrlrID	**SCSI Controller ID** The ID number of the SCSI controller attached to the drive. The driver uses this number to communicate with the controller.
PD_ScsiOpt	**SCSI Driver Options Flags** Indicate the SCSI device options and operation modes. It is the driver's responsibility to use or reject these values, as applicable. bit 0: 0 = ATN not asserted (no disconnect allowed) 1 = ATN asserted (disconnect allowed) bit 1: 0 = Device cannot operate as a target 1 = Device can operate as a target bit 2: 0 = Asynchronous data transfer 1 = Synchronous data transfer bit 3: 0 = Parity off 1 = Parity on All other bits are reserved.

Name	Description
PD_Rate	**Data Transfer/Rotational Rate** Contains the data transfer rate and rotational speed of the floppy media. Note that this field is normally used only when the physical size field (PD_TYP, bits 1-3) is non-zero.

bits 0-3: Rotational speed

0 = 300 RPM
1 = 360 RPM
2 = 600 RPM

All other values are reserved.

bits 4-7: Data transfer rate

0 = 125K bits/sec
1 = 250K bits/sec
2 = 300K bits/sec
3 = 500K bits/sec
4 = 1M bits/sec
5 = 2M bits/sec
6 = 5M bits/sec

All other values are reserved.

Name	Description
PD_MaxCnt	**Maximum Transfer Count** Contains the maximum byte count that the driver can transfer in one call. If this field is 0, RBF defaults to the value of $ffff (65,535).

RBF Path Descriptor Definitions

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

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

Name	Description
PD_ATT	**File Attributes** The file's attributes are defined as follows: bit 0: Set if owner read. bit 1: Set if owner write. bit 2: Set if owner execute. bit 3: Set if public read. bit 4: Set if public write. bit 5: Set if public execute. bit 6: Set if only one user at a time can open the file. bit 7: Set if directory file.
PD_FD	**File Descriptor** The LSN (Logical Sector Number) of the file's file descriptor is written here.
PD_DFD	**Directory File Descriptor** The LSN of the file's directory's file descriptor is written here.
PD_DCP	**File's Directory Entry Pointer** The current position of the file's entry in its directory.
PD_DVT	**Device Table Pointer (copy)** The address of the device table entry associated with the path.
PD_SctSiz	**Logical Sector Size** The logical sector size of the device associated with the path. If this is 0, assume a size of 256 bytes.
PD_NAME	**File Name**

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

Offset	Name	Description
$80	PD_DTP	Device Class
$81	PD_DRV	Drive Number
$82	PD_STP	Step Rate
$83	PD_TYP	Device Type
$84	PD_DNS	Density
$85		Reserved
$86	PD_CYL	Number of Cylinders
$88	PD_SID	Number of Heads/Sides
$89	PD_VFY	Disk Write Verification
$8A	PD_SCT	Default Sectors/Track
$8C	PD_TOS	Default Sectors/Track 0
$8E	PD_SAS	Segment Allocation Size
$90	PD_ILV	Sector Interleave Factor
$91	PD_TFM	DMA Transfer Mode
$92	PD_TOffs	Track Base Offset
$93	PD_SOffs	Sector Base Offset
$94	PD_SSize	Sector Size (in bytes)
$96	PD_Cntl	Control Word
$98	PD_Trys	Number of Tries
$99	PD_LUN	SCSI Unit Number of Drive
$9A	PD_WPC	Cylinder to Begin Write Precompensation
$9C	PD_RWR	Cylinder to Begin Reduced Write Current
$9E	PD_Park	Cylinder to Park Disk Head
$A0	PD_LSNOffs	Logical Sector Offset
$A4	PD_TotCyls	Number of Cylinders On Device
$A6	PD_CtrlrID	SCSI Controller ID
$A7	PD_Rate	Data Transfer/Rotational Rates
$A8	PD_ScsiOpt	SCSI Driver Option Flag
$AC	PD_MaxCnt	Maximum Transfer Count
$B0		Reserved
$B5	PD_ATT	File Attributes
Offset	**Name**	**Description**

$B6	PD_FD	File Descriptor
$BA	PD_DFD	Directory File Descriptor
$BE	PD_DCP	File's Directory Entry Pointer
$C2	PD_DVT	Device Table Pointer (copy)
$C6		Reserved
$C8	PD_SctSiz	Logical Sector Size
$CC		Reserved
$E0	PD_NAME	File Name

RBF Device Drivers

RBF reads and writes in logical blocks, called sectors. A logical sector can be any integral binary power from 256 to 32768. The file manager takes care of all file system processing and passes the driver a starting logical sector number (LSN), a sector count, and the address of the data buffer for each read or write operation.

The logical sector size of the media is determined by RBF when a path is opened to the device. RBF queries the driver to determine whether the driver can support variable sector sizes or not, using the SS_VarSect GetStat call.

If the driver supports variable sector size, RBF assumes that the logical and physical sector sizes are the same, with the size that is specified in PD_SSize.

If the driver does not support variable sector sizes, RBF assumes a logical sector size of 256 bytes, and ignores the value in PD_SSize. If the media physical sector size is not 256 bytes, it is the driver's responsibility to translate and deblock RBF LSNs into the media's LSNs. For example, if PD_SSize is set to 512, and a read request of eight sectors at LSN four is made, the driver should translate the operation into a read of four sectors at LSNtwo.

Read and write calls to the driver initiate the sector read/write operations and, if required, a prior seek operation.

If the controller cannot be interrupt-driven, it must wait until the media is ready, and then transfer the data by polling. If possible, avoid disk controllers that cannot be interrupt-driven. They cause the driver to dominate the system CPU while disk I/O is in progress.

For interrupt-driven systems, the driver initiates the I/O operation and suspends itself (F$Sleep or F$Event) until the interrupt arrives. The interrupt service routine then services the interrupt and "wakes up" the driver.

NOTE: If the driver is awakened by a signal (for example, a keyboard abort) while waiting for the I/O interrupt to occur, it should suspend itself again until the I/O interrupt has occurred. This is because many read/write calls to a driver are made by RBF on behalf of itself, such as in directory searching or bitmap updating. If a signal causes a process to terminate, RBF determines the appropriate time to return to the kernel. Failure to enforce the I/O interrupt completion may result in "locked" disks or corrupted media.

If DMA (Direct Memory Access) hardware support is available, I/O performance increases dramatically because the driver will not have to move the data between memory and the controller.

When the driver reads sector zero, it should copy the first 21 bytes of the sector into the drive table (PD_DTB) associated with the logical unit. Sector zero of the disk media has format information recorded by the format utility. This information allows the driver to determine the actual format of the media and to compare the device physical capabilities specified in the path descriptor options with the media format. This allows the driver to adapt its operation for reading and writing multiple formats in one physical drive. For example, a floppy drive that can read/write double-sided, double-density disks can be made to operate with single-sided or single-density media.

RBF always reads sector zero of the media when a file is opened. Many RBF drivers provide ***caching*** of sector zero to improve the performance of I$Open calls by RBF. This function is generally associated with media that is non-removable (for example, fixed hard disks). When a hard disk driver reads sector zero, it updates the drive table and copies the full sector zero into a local buffer. The state of the buffered sector for the unit is recorded in the logical unit drive table variables V_ZeroRd and V_ScZero. This enables the driver to return sector zero data on subsequent calls by RBF without accessing the disk. Removable media should not have sector zero buffered unless the driver is capable of automatically detecting the media removal (by an interrupt) and marking sector 0 unbuffered.

GetStat calls to RBF devices are generally processed by RBF itself, and thus are not normally seen by the driver. The main exception is the SS_VarSect call, which RBF uses to inquire about the driver's ability to support non-256-byte logical sectors.

The INIT and TERM routines of RBF drivers are called directly by the kernel when the device is attached and detached. Typically, the INIT routine only performs controller-specific initialization such as adding the controller to the IRQ polling table, setting default values in the drive tables, and initializing the controller hardware interface.

NOTE: The INIT routine generally does not perform initialization of the logical units attached to the controller, for example, disk parameter definitions for SCSI drives. This type of initialization should normally be done when the first Read/Write/GetStat/SetStat call is made to the unit.

The TERM routine typically disables the device's interrupts, if required, and removes the controller from the IRQ polling table.

Main Driver Types

The complexity of RBF drivers depends on the capabilities of the hardware involved. Simple hardware controllers require more effort by the driver than do intelligent controllers. Generally RBF drivers fall into three levels of complexity:

- **Simple Floppy Interfaces**
  These types of drivers perform all physical drive movement operations explicitly: seek head, wait for head settle delay, etc. They translate the RBF LSN into a track/head/sector, select the drive, move the disk head to the required position, and then issue the I/O command. If multiple drives are connected to the controller, the driver often has to maintain a record of the current head position of each drive.

- **Combined Hard/Floppy Interfaces**
  These types of drivers deal with “medium” intelligence controllers. Typically, the physical drive selection and automatic seeking are handled by the controller itself. The driver becomes somewhat simpler because it must only translate the RBF LSN into a track/head/sector value. The addition of hard disk operation to the driver adds some minor complexity to the driver due to the differences in floppy vs. hard disk operation.

- **Intelligent Controllers**
  These types of drivers are typically used with SCSI or similar style controllers. These controllers usually accept only a command “packet” indicating the operation required and the address of the operation. These drivers are similar to “medium” intelligence controllers, with the exception that the RBF LSN is usually accepted directly by the controller as the physical sector number.

RBF Device Driver Storage Definitions

RBF device driver modules contain a package of subroutines that perform sector-oriented I/O to or from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical I/O controllers.

The kernel allocates a static storage area for each device (which may control several drives). The size of the storage area is specified in the device driver module header (M$Mem). Some of this storage area is required by the kernel and RBF; the device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the rbfstat.a and drvstat.a DEFS files. Static storage is used as follows:

Offset	Name	Maintained By	Description
$00	V_PORT	Kernel	Device base port address
$04	V_LPRC	File Manager	Last active process ID
$06	V_BUSY	File Manager	Current active process
$08	V_WAKE	Driver	Process ID to awaken
$0A	V_PATHS	Kernel	Linked List of Open Paths
$2E	V_NDRV	Driver	Number of Drives
$2F			Reserved
$36	DRVBEG	Driver/File Man.	Drive Tables

NOTE: ***Offset*** refers to the location of a field, relative to the starting address of the static storage. Offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l.

Name	Description
V_PORT	**Device base port address** The device's physical port address. It is copied from M$Port in the device descriptor when the device is attached by the kernel.
V_LPRC	**Last active process ID** The process ID of the most recent process to use the device. This field is required by the kernel for all device driver static storage, but is not used by RBF.
V_BUSY	**Current active process** The process ID of the process currently using the device. It is used to implement I/O Blocking by RBF. This field is also used by the interrupt drivers when they wish to suspend themselves, by copying V_BUSY to V_WAKE (prior to suspending themselves). A value of zero indicates the device is not busy.
V_WAKE	**Process ID to awaken** The process ID of any process that is waiting for the device to complete I/O. A value of zero indicates that no process is waiting. V_WAKE is set by the driver from V_BUSY and provides the interlock between the driver and the driver's interrupt service routine.
V_PATHS	**Linked List of Open Paths** This is a singly-linked list of all paths currently open on this device.
V_NDRV	**Number of drives** This field is set by the driver's INIT routine to indicate the maximum number of logical drives the driver can use. RBF validates the logical drive number of the drive (PD_DRV) against this number prior to setting the drive table pointer (PD_DTB). PD_DRV must be less than V_NDRV.
V_DRVBEG	**Drive Tables** This section contains one table for each drive that the controller will handle. The drive table associated with the drive is indicated by the drive table pointer (PD_DTB) in the path descriptor.

Device Driver Tables

After the driver's INIT routine has been called, RBF requests the driver to read the identification sector (LSN 0) from the drive. After reading sector zero, the driver must initialize the corresponding drive table. It does this by copying the number of bytes specified by DD_SIZ (21) from the beginning of sector 0 into the appropriate table (PD_DTB). The following is the format of each drive table:

Offset	Name	Maintained By	Description
$00	DD_TOT	Sector 0	Total Number of Sectors
$03	DD_TKS	Sector 0	Track Size (in sectors)
$04	DD_MAP	Sector 0	Number of Bytes in Allocation Map
$06	DD_BIT	Sector 0	Number of Sectors/Bit (cluster size)
$08	DD_DIR	Sector 0	LSN of Root Directory FD
$0B	DD_OWN	Sector 0	Owner ID
$0D	DD_ATT	Sector 0	Attributes
$0E	DD_DSK	Sector 0	Disk ID
$10	DD_FMT	Sector 0	Disk Format: Density/Sides
$11	DD_SPT	Sector 0	Sectors/Track
$13	DD_RES		Reserved
$16	V_TRAK	Driver	Current Track Number
$18	V_FileHd	File Manager	Open File List for Disk
$1C	V_DiskID	File Manager	Disk ID
$1E	V_BMapSz	File Manager	Bitmap Size
$20	V_MapSct	File Manager	Lowest Bitmap Sector to Search
$22	V_BMB	File Manager	Bitmap In Use Flag
$24	V_ScZero	Driver	Pointer to Sector 0
$28	V_ZeroRd	Driver	Sector 0 Read Flag
$29	V_Init	Driver	Drive Initialized Flag
$2A	V_Resbit	File Manager	Reserved Bitmap Sector Number
$2C	V_SoftEr	Driver	Number of Recoverable Errors
$30	V_HardEr	Driver	Number of Non-Recoverable Errors
$34	V_Cache	Cache Utility	Drive Cache Queue Head
$38	V_DText	Driver	Drive Table Extension pointer
$3A	V_MapMax	File Manager	Maximum Bitmap Sector Number
$3C			Reserved (22 bytes)

NOTE: There must be as many tables as are specified in V_NDRV. All references to Sector 0 in the **Maintained By** column mean that this field is initialized by the driver with information obtained from Sector 0 when it is first read.

Name	Description
DD_TOT	**Total Number of Sectors** Contains the size of the media in sectors. RBF uses this field to set the size of the "raw" device file ("@" file opens). The driver can also use this value to verify that the LSN passed by RBF is in range for the media. Driver INIT routines typically initialize this field in the drive table to a non-zero value, so that sector 0 may be read initially.
DD_TKS	**Track Size (in sectors)** Contains the number of sectors per track, as a byte value.
DD_MAP	**Number of Bytes in Allocation Map** Contains the size of the media bitmap.
DD_BIT	**Number of Sectors/Bit (cluster size)** Contains the size of a cluster of sectors on the disk. This value is always an integral power of two.
DD_DIR	**LSN of Root Directory FD** Contains a pointer to the file descriptor of the media's root directory.
DD_OWN	**Owner ID** The user ID of the disk owner.
DD_ATT	**Attributes** Defines the access attributes of the media.
DD_DSK	**Disk ID** Contains a pseudo-random number which identifies the media volume. This number is put here by the format utility.
DD_FMT	**Disk Format: Density/Sides** Defines the format of the media volume, to enable drivers to adapt to different formats: bit 0: 0 = Single-sided 1 = Double-sided bit 1: 0 = Single-density (FM) 1 = Double-density (MFM) bit 2: 1 = Double-track density (96 TPI/135 TPI) bit 3: 1 = Quad track density (192 TPI) bit 4: 1 = Octal track density (384 TPI)

Name	Description
DD_SPT	**Sectors/Track** A two byte value of DD_TKS.
V_TRAK	**Current Track Number** This value is used to record the current track number of a logical unit for those drivers that need to perform seek functions explicitly. Typically, driver INIT routines initialize this field to an unknown track number (for example, $FF), so that the first access to the drive results in a restore operation.
V_FileHd	**Open File List for Disk** A pointer to the list of all files open on the logical unit.
V_DiskID	**Disk ID** A copy of DD_DSK.
V_BMapSz	**Bitmap Size** The size of the media's bitmap.
V_MapSct	**Lowest Bitmap Sector to Search** The starting sector number to begin bitmap allocation functions.
V_BMB	**Bitmap In Use Flag** Indicates whether or not the bitmap is in use.
V_ScZero	**Pointer to Sector 0** A pointer to a buffered sector zero for the unit. This is only used by drivers that perform this function.
V_ZeroRd	**Sector 0 Read Flag** Used by the driver to indicate whether or not the buffered sector zero is valid. If the data is valid, this flag should be non-zero.
V_Init	**Drive Initialized Flag** Used by the driver to indicate whether or not the logical unit has been initialized. If the unit has been initialized, this field should be non-zero.
V_Resbit	**Reserved Bitmap Sector Number** Indicates the bitmap sector number to ignore during RBF bitmap allocation functions. It is set by the SS_RsBit SetStat call.

Name	Description
V_SoftEr	**Number of Recoverable Errors** Allows the driver to keep a count of "soft" errors during I/O operations. The value is typically returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
V_HardEr	**Number of Non-Recoverable Errors** Allows the driver to keep a count of "hard" errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
V_Cache	**Drive Cache Queue Head** A pointer to the cache queue for the drive.
V_DTExt	**Drive Table Extension Pointer** A pointer to an extension of the drive table. Drivers that require storage of additional drive table variables can use this field as a pointer to the extra information.
V_MapMax	**Maximum Bitmap Sector Number** The sector number of the last sector of the bitmap.

Linking RBF Drivers

After a RBF driver has been assembled into its relocatable object file (ROF), the driver needs to be linked to produce the final driver module. Linking resolves all code references in drivers that are comprised of several ROF files. It also resolves the external data and static storage references by the driver.

The most important part of linking is to correctly resolve the static storage references. Generally, the static storage area is composed of three sections in this order (see Figure 2-1):

¨ I/O globals

¦ Drive tables (one per logical drive)

Æ Driver-declared variables

The driver-declared variables are declared in vsect areas of the driver, but they *must* be allocated after the drive table storage areas. The method that you must use to allocate all of the storage, *in the correct order*, is to link one of the drvsX.l library files *before* the user written ROF files. The drvsX.l files are usually found in the system's LIB directory. Each drvsX.l file contains vsect declarations that allocate the I/O system variables and the appropriate number of drive tables. For example, drvs1.l allocates the I/O system-defined section and one drive table, while drvs4.l allocates the I/O system-defined section and four drive tables. The following is a typical linker command line for an RBF driver:

```
l68 /dd/LIB/drvs4.l REL/rb320.r -O=OBJS/rb320
```

NOTE: Specifying the drvsX.l file first causes the vsect variables declared by the file to be allocated *before* the vsect variables in the ROF file. Failure to correctly allocate the I/O system and drive table variables first, or failure to link the correct number of drive tables at all, results in erratic driver operation.

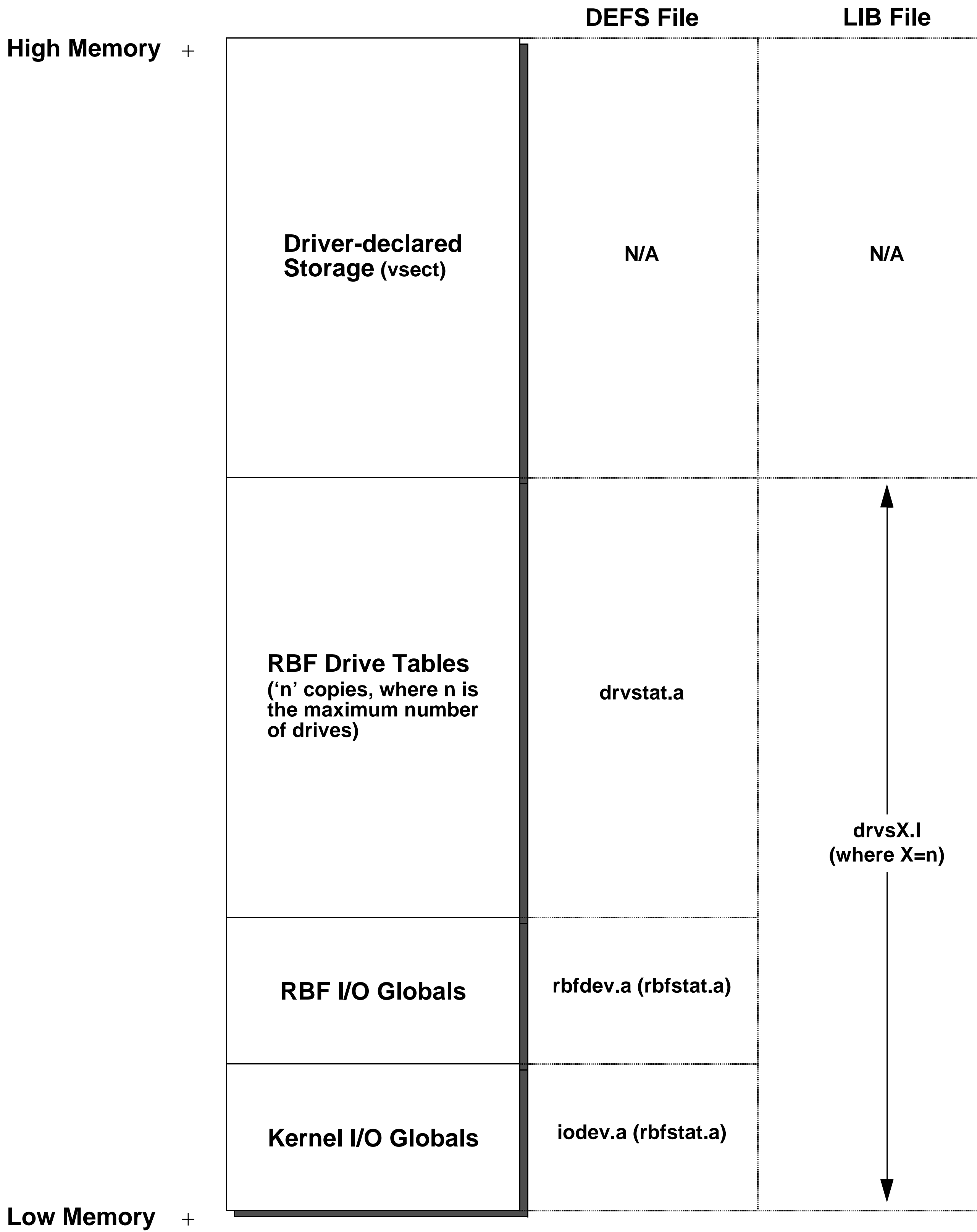


Figure 2-1: RBF Static Storage Layout

RBF Device Driver Subroutines

As with all device drivers, RBF device drivers use a standard executable memory module format with a module type of Drivr (code $E0). RBF drivers are called in system state.

NOTE: I/O system modules must have the following module attributes:

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

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of a corresponding subroutine. The branch table appears as follows:

```
ENTRY dc.w   INIT      initialize device
      dc.w   READ      read character
      dc.w   WRITE     write character
      dc.w   GETSTAT   get device status
      dc.w   SETSTAT   set device status
      dc.w   TERM      terminate device
      dc.w   TRAP      handle illegal exception (0 = none)
```

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, the carry bit should be set and an appropriate error code returned in the least significant word of register d1.w.

The TRAP entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

The following pages describe each subroutine.

INIT — Initialize Device and its Static Storage Area

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The INIT routine must:

¨ Initialize the device's permanent storage. Minimally, this consists of:

- Initializing V_NDRV to the number of drives with which the controller will work.
- Initializing DD_TOT in each drive table to a non-zero value so that sector zero may be read or written to.
- If the driver must perform explicit seeks, initializing V_TRAK to $FF so that the first seek will find track zero.

¦ Place the IRQ service routine on the IRQ polling list by using the F$IRQ system call.

Æ Initialize device control registers (enable interrupts if necessary).

Prior to being called, the device static storage is cleared (set to zero), except for V_PORT which contains the device address. The driver should initialize each drive table entry appropriately for the type of disk the driver expects to be used on the corresponding drive.

If INIT returns an error, it does not have to clean up its operation, for example, remove device from polling table or disable hardware. The kernel calls TERM to allow the driver to clean up INIT's operation before returning to the calling process.

Usually, the INIT routine should only perform controller-specific initialization, as opposed to drive-specific initialization. This is because the controller may have more than one type of drive connected to it.

NOTE: If the INIT routine causes an interrupt to occur, you can handle the interrupt in one of the following ways:

- Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, and then restoring the previous interrupt level. This is the preferred technique unless the interrupt is time-consuming.

- Allow the interrupt service routine to service the hardware. In this case, the process descriptor contains the process ID (P$ID) to which V_WAKE should be set. V_BUSY cannot be used because it is zero when INIT is called.

READ

Read Sector(s)

INPUT:

d0.l = number of contiguous sectors to read
d2.l = disk logical sector number to read
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT:

Sector(s) returned in the sector buffer

ERROR OUTPUT:

cc = carry bit set
d1.w = error code

FUNCTION:

The READ routine must perform the following operations:

¿ Locate the associated drive table (PD_DTB) and determine if it is initialized. If not, perform any drive initialization required and mark the drive initialized in the drive table. If the driver will perform sector zero buffering for the unit, allocate a sector zero buffer.

¡ Verify the starting LSN and ending LSN (if a multi-sector read) against the size of the media (DD_TOT).

¬ Compute the physical disk address (track/head/sector) from the LSN, if required.

Ð If the driver supports sector 0 buffering, and the read request is for sector 0, return the sector 0 data to the buffer specified. If no further sectors are requested, return to RBF. Otherwise, proceed to read the remaining sectors into the remainder of the buffer.

× For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require that you save the current track position in the last selected drive's drive table (V_TRAK).

± Prepare the hardware for the read request and start the I/O operation. The data should be read into the buffer specified by PD_BUF.

ð Wait for the I/O operation to complete (with interrupts, if possible).

Š If the starting LSN of the read was not LSN 0, return to RBF. Otherwise:

a) Update the unit's drive table by copying the number of bytes specified by DD_SIZ (21) from the beginning of sector 0 into the appropriate table.

b) If the driver supports buffering sector zero for the unit, copy sector zero into the driver's local buffer (V_ScZero) and mark the buffer valid (V_ZeroRd).

¥ If the logical unit and driver support multiple disk formats, the driver should validate that the media is readable by the drive. If not, the driver should return a Bad Type error (E$BTyp). If it can, the driver should ready itself for the new format by either:

a) Marking the logical unit as uninitialized (V_Init cleared), so the next access will cause the unit to be re-initialized by the driver.

b) Re-initializing the unit hardware for the new format.

µ Return the status of the read to RBF.

Sector/Transfer Count

The number of sectors to transfer is passed by RBF. If bit number one in PD_Cntl is clear, RBF always requests only one sector. If the bit is set, RBF requests a maximum count, based on the value in PD_MaxCnt. The value in PD_MaxCnt is truncated to an exact sector count, so that the device always sees requests in terms of an integral number of sectors.

Sector Zero Reads

Whenever logical sector zero is read from the media, the first part of it must be copied into the drive table for the logical unit. PD_DTB contains the pointer to the drive table. The number of bytes to copy is DD_SIZ.

Drivers that buffer sector zero also update their local copy when sector zero is read from the media. The drive table variables V_ScZero (pointer to sector zero) and V_ZeroRd (sector zero valid flag) allow the driver to maintain this buffer. When the driver receives a read request for LSN zero, it can check these flags. If the buffer is valid, it can simply return the buffered data to RBF without performing any disk I/O.

Sector zero buffering should normally be performed only on fixed media (fixed hard disks). This ensures that media volume changes are noticed by RBF. Failure to detect media changes correctly can result in corruption of the new volume.

If the driver can detect media removal (for example, via an interrupt when the door is opened), it is permissable for the driver to buffer sector 0 while the media is installed.

Sector Size Support

If the driver supports variable sector sizes, RBF assumes that the size of a sector is specified by PD_SSize, and that the logical and physical sector sizes are the same. Drivers operating under this mode simply process the RBF transfer count and LSN address according to the disk's requirements.

If the driver does not support variable sector sizes (logical sector size is 256 bytes) and the physical sector size of the media (PD_SSize) is not 256 bytes, the driver must deblock the media sectors. Typically, this involves the following steps:

¨ Determine if RBF's starting LSN falls at the start of a media physical sector. If not, check if the physical sector is currently buffered by the driver. If the physical sector is currently buffered by the driver, copy the appropriate part of the buffer to RBF's buffer. If not, read the physical sector into the driver's buffer and return the appropriate part to RBF's buffer.

¦ If any sectors remain to be read, convert the remaining start address and count into the physical start address and count. Then, read (and count) those sectors into the RBF buffer.

Æ If any partial sector remains to be read, read that physical sector into the driver's physical buffer. Then, return the appropriate part of the buffer to the end of the RBF buffer.

Interrupt-driven Operation

If the hardware uses interrupts to perform I/O, the driver should perform the following steps:

Synchronization using Signals

1. Issue the I/O command to the hardware.
2. Copy V_BUSY to V_WAKE in the static storage.
3. The driver should then suspend itself (F$Sleep).
4. The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and sends a wake-up signal (S$Wake) to the driver. The driver's process ID is in V_WAKE. After sending the signal, the IRQ service routine should clear V_WAKE to signify that the interrupt occurred.
5. When the driver awakens, it should check V_WAKE. If zero, the interrupt has occurred and the driver can continue to check status, etc. If non-zero, the driver should suspend itself again.

Synchronization using Events

1. Issue the I/O command to the hardware.
2. The driver should suspend itself using the event system's "wait" function.
3. The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and then uses the event system's "signal" function to awaken the driver.
4. When the driver awakens, it should determine if the event value is within range. If so, the interrupt was serviced and the driver can check the status, etc. If not, the driver should suspend itself again.

WRITE **Write Sector(s)**

INPUT:
d0.l = number of contiguous sectors to write
d2.l = disk logical sector number
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT: **The sector buffer is written to disk.**

ERROR OUTPUT:
cc = carry bit set
d1.w = error code

FUNCTION: The WRITE routine must perform the following operations:

¨ Determine the starting LSN. If zero, the driver should check the format control flag for format protection (PD_Cntl, bit 0). If bit 0 is clear, the media can be formatted and sector 0 may be written. If bit 0 is set, the media is format protected and the driver should return an E$Format error.

¡ Locate the associated drive table (PD_DTB) and check if the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.

¬ If the driver supports buffering of sector 0 for the unit, and sector 0 is being written, the driver should clear V_ZeroRd to mark that sector 0 is unbuffered.

Ø Verify the starting LSN (and ending LSN, if a multi-sector write) against the size of the media (DD_TOT).

× Compute the physical disk address (track/head/sector) from the LSN, if required.

± For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require you to save the current track position in the last selected drive's drive table (V_TRAK).

ð Prepare the hardware for the write request and start the I/O operation. The data should be written from the buffer specified by PD_BUF.

Š Wait for the I/O operation to complete (with interrupts, if possible).

¥ Return the status of the write to RBF.

Sector/Transfer Count

The number of sectors to transfer is passed by RBF. If bit number one in PD_Cntl is clear, RBF always requests only one sector. If the bit is set, RBF requests a maximum count, based on the value in PD_MaxCnt. The value in PD_MaxCnt is truncated to an exact sector count, so that the device always sees requests in terms of an integral number of sectors.

Sector Zero Writes

Whenever the starting LSN is zero, the driver should check whether the media may be formatted (PD_Cntl, bit 0). If bit 0 is set, the media is format protected and sector zero may not be written. The driver should return a E$Format (format protected) error in this case.

If the driver buffers sector zero of the media, it should clear V_ZeroRd to mark the buffer invalid. This ensures that the next read of sector zero will access the media.

Sector Size Support

If the driver supports variable sector sizes, RBF assumes that the size of a sector is specified by PD_SSize, and that the logical and physical sector sizes are the same. Drivers operating under this mode simply process the RBF transfer count and LSN address according to the disk’s requirements.

If the driver does not support variable sector sizes (logical sector size is 256 bytes) and the physical sector size of the media (PD_SSize) is not 256 bytes, the driver must deblock the media sectors. Typically, this involves the following steps:

¨ Determine if RBF’s starting LSN falls at the start of a media physical sector. If not, and the physical sector is not currently cached, read the physical sector into the driver’s local buffer. Update the appropriate part of the buffer with RBF’s data and write the local buffer to the media.

¦ If any sectors remain to be written, convert the remaining start address and count into the physical start address and count. Then, write (and count) those sectors from the RBF buffer.

Æ If any partial sector remains to be written, read that physical sector into the driver’s local buffer. Next, update the appropriate part of the buffer with RBF’s data and write the local buffer to the media.

Interrupt Operation

If the hardware uses interrupts to perform I/O, the driver should perform the following steps:

Synchronization using Signals

¨ Issue the I/O command to the hardware.

¦ Copy V_BUSY to V_WAKE in the static storage.

Æ The driver should suspend itself (F$Sleep).

Ø The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and sends a wake-up signal (S$Wake) to the driver. The driver's process ID is in V_WAKE. After sending the signal, the IRQ service routine should clear V_WAKE to signify that the interrupt occurred.

× When the driver awakens, it should check V_WAKE. If zero, the interrupt has occurred and the driver can continue to check status, etc. If non-zero, the driver should suspend itself again.

Synchronization using Events

¨ Issue the I/O command to the hardware.

¦ The driver should suspend itself using the event system's "wait" function.

Æ The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and then uses the event system's "signal" function to awaken the driver.

Ø When the driver awakens, it should check that the event value is within range. If so, the interrupt was serviced and the driver can check the status, etc. If not, the driver should suspend itself again.

GETSTAT/SETSTAT — Get/Set Device Status

INPUT: **d0.w = status code**
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT: **Depends on the function code**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: These routines are wild-card calls used to get/set the device's operating parameters as specified for the I$GetStt and I$SetStt service requests.

Calls which involve parameter passing require the driver to examine or change the register stack variables. These variables contain the contents of the MPU registers at the time of the I$Getstt/I$SetStt request was made. Parameters passed to the driver are set up by the caller prior to using the service call. Parameters passed back to the caller are available when the service call completes.

Typical RBF drivers handle the following I$GetStt/I$SetStt calls:

I$GetStt: SS_DSize, SS_VarSect

I$Setstt: SS_Reset, SS_SQD, SS_WTrk

Any unsupported I$GetStt/I$SetStt calls to the driver should return an unknown service error (E$UnkSvc).

NOTE: A minimal RBF driver should support SS_Reset and SS_WTrk, so that media may be formatted.

The following pages describe the driver implementation of the above I$GetStt/I$SetStt calls.

GetStat Call:

SS_DSize This routine is used to return the media size for autosize devices (PD_Cntl, bit three set). The routine must perform the following steps:

¿ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.

¡ Prepare the hardware for the request and start the I/O operation.

¬ Wait for the I/O operation to complete (with interrupts, if possible).

Ð Return the media size (in terms of its logical sector size) to the caller's d2 register (R$d2 offset from passed a5). Note that if the driver supports deblocking (logical and physical sizes are not the same), the returned sector count should be a "logical" sector count.

ƒ Return status to RBF.

SS_VarSect This routine is called by RBF whenever a path is opened to the device, so that RBF can determine the logical sector size of the media. The driver should indicate its support for variable logical sector sizes as follows:

- If variable logical sector sizes are supported , the driver should return a "no error" status. Upon return to RBF, RBF uses the value in PD_SSize as the media's logical sector size. It is permissable for the driver to query the drive for its current sector size setting and update PD_SSize during this call.

WARNING: Querying the drive *does not* mean issuing a physical read of the disk's sector 0 (to read DD_LSNSize) as RBF has not yet set up the buffer pointers for the path (PD_BUF = 0). Unless you take special care, attempting to perform physical data I/O at this point will probably crash the system. The only type of I/O operations valid at this point are generally internal driver operations (for example, Mode Sense command to a SCSI drive). Drivers that deal with media that cannot return "current sector size" generally require that PD_SSize be set correctly in the device descriptor. The driver returns "no error" to indicate that RBF can use PD_SSize as the logical media size.

- If the driver does not support variable logical sector sizes, it should return an "unknown service request" (E$UnkSvc) error, to indicate to RBF that the logical sector size of the media is 256 bytes and that PD_SSize should be ignored.
- If the driver returns any error other than "unknown service request", RBF aborts the path open operation and returns the error to the caller.

SetStat Calls:

SS_Reset Recallibrate (restore) the media head to the outer track. This is mainly used by format to ensure the media is at a known position.

The restore routine must perform the following functions:

¿ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.

¡ Prepare the hardware for the request and start the I/O operation.

¬ Wait for the I/O operation to complete (with interrupts, if possible).

Ð Return the status of the restore to RBF.

SS_SQD This is mainly used to move (park) the heads of hard disk drives to a safe area. The park routine must perform the following steps:

¨ Check whether the media may be parked. This typically involves the following:

- Check if the device is a floppy disk. If so, return an E$UnkSvc error.
- Check the PD_Park value. If it is zero or within the range of the RBF media area, return an E$UnkSvc error.

¡ Locate the associated drive table (PD_DTB) and initialize the drive according to the parking function. This typically involves setting the drive's cylinder count to the PD_Park value. After initialization, *do not* mark the drive initialized (V_Init should be clear). This ensures that any subsequent accesses to the drive will cause the drive to be re-initialized correctly (PD_CYL or PD_TotCyls count instead of PD_Park).

¬ Prepare the hardware for the park request and start the I/O operation.

Ð Wait for the I/O operation to complete (with interrupts, if possible).

ƒ Return the status of the park to RBF.

The park operation typically consists of issuing a seek or read command and specifying a sector address on the desired cylinder. On some drives/controllers, this may fail because the parking cylinder is not formatted and the controller attempts to verify the seek/read. In these situations, it is typical for the driver to perform a write track operation on the desired track.

SS_WTrk This is used by format to perform physical initialization of the media. The write track routine must perform the following steps:

¨ Check whether the media may be formatted (PD_Cntl, bit 0 clear). If not, the media is format protected and the driver should return an E$Format error.

¡ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform the required drive initialization and mark the drive initialized in the drive table. If the driver supports buffering sector 0 for the unit, and the track being formatted is the first track of the media (PD_TOffs), the driver should clear V_ZeroRd to mark that sector 0 is unbuffered.

¬ If the driver supports any buffering of physical sectors (non "VarSect" driver with physical sectors not equal to 256 bytes), it should mark any active buffers as invalid.

Ð For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require the current track position to be saved in the last selected drive's drive table (V_TRAK).

ƒ Prepare the hardware for the write track request and start the I/O operation.

Ý Wait for the I/O operation to complete (with interrupts, if possible).

ý Return the status of the write track to RBF.

The method of formatting disk drives varies with the hardware in use. However, note the following points:

¨ The parameters passed are physical parameters, with one exception: the sector interleave table. If the driver must pass the interleave table to the hardware (or prepare its own table), it must add the PD_SOffs value to each interleave table entry so that a physical interleave table is passed to the hardware.

¦ The driver typically only initializes the drive when the track number passed is equal to the PD_TOffs value (that is, at the beginning of the format operation).

Æ SS_WTrk calls to the driver issued by format are dependent on the autosize flag in PD_Cntl (bit three) in the following manner:

- If the media is *autosize capable* (bit three set), format makes only one SS_WTrk call to the driver with the passed track number being equal to PD_TOffs. The driver is expected to format the entire media from this call.
- If the media is *non-autosize capable* (bit three clear), format issues a SS_WTrk call for each track on the media (PD_CYLS x PD_SID). The driver is expected to format the media one track at a time. If the hardware cannot handle individual tracks, the driver must perform a *format all media* operation on the first SS_WTrk call (PD_TOffs equal to the passed track number and side number zero) and simply ignore all other SS_WTrk calls without returning an error.

TERM — Terminate Device

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage area
(a6) = system global static storage pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: This routine is called when a device is no longer in use in the system (see I$Detach).

The TERM routine must:

¨ Wait until any pending I/O has completed.

¦ Disable the device interrupts.

Æ Remove the device from the IRQ polling list.

Ø Return any buffers the driver has requested on behalf of itself, for example, sector zero buffers or physical sector deblocking buffers.

NOTE: The driver should not attempt to return buffers within its defined static storage area. The kernel releases this memory when the TERM routine completes.

NOTE: If an error occurs during the device's INIT routine, the kernel calls the TERM routine to allow the driver to clean up. If the TERM routine uses static storage variables (for example, interrupt mask values, dynamic buffer pointers), it should validate these variables prior to using them. The INIT routine may not have set up all the variables prior to exiting with the error.

IRQ Service Routine

Service Device Interrupts

INPUT: **(a2) = static storage address**
(a3) = port address
(a6) = system global static storage

OUTPUT: **None**

ERROR OUTPUT: **cc = carry set (interrupt not serviced)**

FUNCTION: This routine is called directly by the kernel's IRQ polling table routines. Its function is to:

¨ Check the device for a valid interrupt. If the device does not have an interrupt pending, the carry bit must be set and the routine exited with an RTS instruction as quickly as possible. Setting the carry bit signals the kernel that the next device on the vector should have its IRQ service routine called.

¦ Service device interrupts.

Æ Wake up the driver mainline, using the synchronization method of the driver:

Signals: Send a wake-up signal to the process whose process ID is in V_WAKE, when the I/O is complete. Also, clear V_WAKE as a flag to the mainline program that the IRQ has occurred.

Events: Signal the event that the IRQ has occurred, using the event system's signal function.

Ø Clear the carry bit and exit with an RTS instruction after servicing an interrupt.

Avoid exception conditions (for example, a Bus Error) when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

NOTE: IRQ service routines may destroy the contents of the following registers only: d0, d1, a0, a2, a3, and a6. You must preserve the contents of all other registers or unpredictable system errors (system crashes) will occur.

End of Chapter 2

NOTES

Sequential Character File Manager (SCF)

SCF General Description

The Sequential Character File Manager (SCF) is a re-entrant subroutine package for I/O service requests to devices which operate on a character-by-character basis, such as terminals, printers, and modems. SCF can handle any number or type of character-oriented devices. It includes some input and output editing functions for line-oriented operations such as backspace, line delete, repeat line, auto line feed, screen pause, and return delay padding.

The following I/O service requests are handled by SCF:

I$Close	I$Create	I$GetStt	I$Open	I$Read
I$ReadLn	I$SetStt	I$Write	I$Writln	

The following I/O service requests are not valid for SCF:

I$ChgDir   I$Delete   I$MakDir   I$Seek

When an I$ChgDir, I$Delete, or I$MakDir is made to SCF, an appropriate error code is returned. I$Seek does not return an error.

The following I/O service requests do not call SCF:

I$Attach   I$Detach   I$Dup

SCF device drivers are responsible for the actual transfer of data between their own internal buffers and the device hardware.

SCF transfers data to/from the driver in register d0. The driver typically operates as follows, depending upon whether or not the driver uses interrupts:

Polled Mode

The WRITE routine writes the data to the hardware and the driver returns immediately. The READ routine checks for available data, waits if there is no data, and returns the data when ready. Polled-mode drivers usually do not buffer the data internally.

NOTE: Polled I/O operation can have a harmful effect on real-time system operation. Polled I/O is acceptable if the device is always ready to send or receive data (for example, output to a memory-mapped video display). Polled I/O is not acceptable if the driver has to wait for the device to send or receive data.

Interrupt Mode

Interrupt-driven drivers typically use input FIFO and output FIFO buffers for the data being read and written. The WRITE routine deposits the data in the output FIFO buffer, arms the output interrupts (if necessary), and allows the device's output interrupt service routine to empty the output FIFO. When the output FIFO is empty, output interrupts are usually disabled. The READ routine checks the input FIFO buffer. If data is available, READ takes the next character from the buffer and returns. If no data is available, READ suspends itself until data is available. The device's input interrupt service routine is responsible for filling the input FIFO and waking any waiting process. Input interrupts are usually enabled for the time that the device is attached to the system.

SCF Line Editing

The I$Read and I$Write service requests to SCF devices pass data to/from the device without modification; SCF does not add line feeds or NULLs after writing a carriage return.

The I$ReadLn and I$Writln service requests to SCF devices perform all line editing functions enabled for the particular device.

Line editing functions are initialized when a path is first opened by copying the option table from the device descriptor associated with that device into the path descriptor. They may be altered later by programs using the I$GetStt and I$SetStt (SS_Opt) service requests. You can use the xmode utility to modify the option table of SCF device descriptors in writable memory, so that changes can be applied prior to opening a path to the device. You can also use the tmode utility to modify the options from the keyboard. Line editing functions are disabled when the option table field is set to zero.

CAVEAT: If software handshaking (X-ON/X-OFF) is enabled, these characters are intercepted by the device driver and not processed by SCF.

SCF I/O Service Requests

When a process makes one of the following system calls to a SCF device, SCF executes the file manager functions described for that call.

I$Close SCF performs the following functions:

- **Checks for additional paths open to the device by the calling process**
  If no additional paths are open, a SS_Relea SetStat is performed to release the device signal conditions and disassociate the device signals from the process.
- **Checks for any other users of the path**
  If there are none, SBF:
  - **Performs a SS_Close SetStat to the driver**
  - **Performs an I$Detach if the device has an output (echo) device**
  - **Returns buffers allocated by the original I$Open call**

I$Create SCF considers this system call synonymous with I$Open.

I$GetStt The SS_Opt GetStat function is supported by SCF. It is passed to the driver to enable the driver to update hardware specific parameters such as the baud rate. If the driver returns an E$UnkSvc error, it is ignored. All other GetStat calls are passed directly to the driver.

Refer to the I$GetStt system call description in the ***OS-9 Technical Manual*** for specific information on the various SCF-oriented I$GetStt functions.

I$Open SCF performs the following functions:

- **Validates the pathname**
- **Allocates memory for the “path buffer”**
- **Initializes the path descriptor with the default options section**
- **Performs an I$Attach if the device has an output (echo) device**
- **Calls the driver with an SS_Open SetStat**
  If the driver returns an E$UnkSvc error, SCF ignores it.

I$Read

I$Read requests read input from the device without modifying the data. The read terminates under any of these circumstances:

- The requested number of bytes has been read.
- An end-of-record character is detected (PD_EOR).
- An end-of-file (PD_EOF) is detected as the first character of the read.
- An error occurs.

You have control over the method of transfer in the following ways:

- De-select (set to zero) the end-of-record (PD_EOR) character using I$GetStt and I$SetStt. This prevents the read from terminating early, due to PD_EOR detection. The read continues until the requested number of characters has been read.
- De-select (set to zero) the end-of-file (PD_EOF) character using I$GetStt and I$SetStt. This prevents the read from terminating when receiving an end-of-file character as the first character of the read.

If the requested data is not immediately available, the driver waits (F$Sleep) for the data. This will "busy" the driver (other processes I/O block) until the data READ request has completed. If you do not wish a process to wait for data, use the SS_Ready GetStat or SS_SSig SetStat calls to detect when an I$Read can be issued.

I$ReadLn

I$ReadLn requests read input from the device and may edit the data. The read terminates under any of these circumstances:

- An end-of-record character is detected (PD_EOR).
- An end-of-file (PD_EOF) is detected as the first character of the read.
- An error occurs.

If the end-of record character is not encountered before the requested number of bytes has been read, SCF echos the line overflow character (PD_OVF) for each subsequent character read. This indicates that the characters are being ignored. This condition is maintained until the end-of-record character is read. You have control over how the data stream is edited by setting the path descriptor options using I$GetStt and I$SetStt.

NOTE: *Never* use I$ReadLn on a path that has its end-of-record (PD_EOR) function disabled, as I$ReadLn can then only terminate on an error or end-of-file condition.

I$SetStt The SS_Opt SetStat function is supported by SCF. After SCF updates the path descriptor option section, it is passed to the driver to enable the driver to update hardware specific parameters such as the baud rate. If the driver returns an E$UnkSvc error, SCF ignores it. All other SetStat calls are passed directly to the driver.

Refer to the I$SetStt system call description in the ***OS-9 Technical Manual*** for specific information on the various SCF-oriented I$SetStt functions.

I$Write I$Write requests output data to the device without modifying the data being passed. The write terminates only when all characters have been sent or an error occurs.

I$WritIn I$WritIn is similar to I$Write except that I$WritIn writes data until an end-of-record character (PD_EOR) is written or until the specified number of bytes has been sent. The line editing that I$WritIn performs for SCF devices consists of auto line feed, null byte padding at end-of-record, tabulation, and auto page pause.

SCF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for SCF devices. The initialization table immediately follows the standard device descriptor module header fields and defines initial values for the I/O editing features. The size of the table is defined in the M$Opt field.

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Type
$49	PD_UPC	Upper Case Lock
$4A	PD_BSO	Backspace Option
$4B	PD_DLO	Delete Line Character
$4C	PD_EKO	Echo
$4D	PD_ALF	Automatic Line Feed
$4E	PD_NUL	End Of Line Null Count
$4F	PD_PAU	End Of Page Pause
$50	PD_PAG	Page Length
$51	PD_BSP	Backspace Input Character
$52	PD_DEL	Delete Line Character
$53	PD_EOR	End Of Record Character
$54	PD_EOF	End Of File Character
$55	PD_RPR	Reprint Line Character
$56	PD_DUP	Duplicate Line Character
$57	PD_PSC	Pause Character
$58	PD_INT	Keyboard Interrupt Character
$59	PD_QUT	Keyboard Abort Character
$5A	PD_BSE	Backspace Output
$5B	PD_OVF	Line Overflow Character (bell)
$5C	PD_PAR	Parity Code, # of Stop Bits, and # of Bits/Character
$5D	PD_BAU	Adjustable Baud Rate
$5E	PD_D2P	Offset To Output Device Name
$60	PD_XON	X-ON Character
$61	PD_XOFF	X-OFF Character
$62	PD_TAB	Tab Character
$63	PD_TABS	Tab Column Width

NOTE: In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, you must make the following adjustment: (M$DTyp - PD_OPT).

For example, to access the letter case in a device descriptor, use PD_UPC + (M$DTyp - PD_OPT). To access the letter case in the path descriptor, use PD_UPC. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

NOTE: You can change or disable most of these special editing functions by changing the corresponding control character in the path descriptor. Do this with the I$SetStt service request, the tmode utility, or the xmode utility.

Name	Description
PD_DTP	**Device Type** Set to zero for SCF devices.
PD_UPC	**Letter case** If PD_UPC is not equal to zero, input or output characters in the range “a..z” are made “A..Z”.
PD_BSO	**Destructive Backspace** If PD_BSO is zero when a backspace character is input, SCF echoes PD_BSE (backspace echo character). If PD_BSO is non-zero, SCF echoes PD_BSE, space, PD_BSE.
PD_DLO	**Delete** If PD_DLO is zero, SCF deletes by backspace-erasing over the line. If PD_DLO is not zero, SCF deletes by echoing a carriage return/line-feed.
PD_EKO	**Echo** If PD_EKO is not zero, then all input bytes are echoed, except undefined control characters which are printed as periods. If PD_EKO is zero, input characters are not echoed.
PD_ALF	**Automatic line feed** If PD_ALF is not zero, carriage returns are automatically followed by line-feeds.
PD_NUL	**End of line null count** Indicates the number of NULL padding bytes to be sent after a carriage return/line-feed character.
PD_PAU	**End of page pause** If PD_PAU is not zero, an auto page pause occurs upon reaching a full screen of output. See PD_PAG for setting page length.

Name	Description
PD_PAG	**Page length** Contains the number of lines per screen (or page).

PD_BSP	**Backspace "input" character** Indicates the input character recognized as backspace. See PD_BSE and PD_BSO.
PD_DEL	**Delete line character** This field indicates the input character recognized as the delete line function. See PD_DLO.
PD_EOR	**End of record character** This field defines the last character on each line entered (I$Read, I$ReadLn). An output line is terminated (I$Writln) when this character is sent. Normally PD_EOR should be set to $0D. **WARNING:** If PD_EOR is set to zero, SCF's I$ReadLn will *never* terminate, unless an EOF or error occurs.
PD_EOF	**End of file character** This field defines the end-of-file character. SCF returns an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input.
PD_RPR	**Reprint line character** If this character is input, SCF (I$ReadLn) reprints the current input line. A carriage return is also inserted in the input buffer for PD_DUP (see below) to make correcting typing errors more convenient.
PD_DUP	**Duplicate last line character** If this character is input, SCF (I$ReadLn) duplicates whatever is in the input buffer through the first PD_EOR character. Normally, this is the previous line typed.
PD_PSC	**Pause character** If this character is typed during output, output is suspended before the next end-of-line. This also deletes any "type ahead" input for I$ReadLn.
PD_INT	**Keyboard interrupt character** If this character is input, SCF sends a keyboard interrupt signal to the last user of this path. It terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code. PD_INT is normally set to a control-C character.

Name	Description
PD_QUT	**Keyboard abort character** If this character is input, SCF sends a keyboard abort signal to the last user of this path. It terminates the current I/O request (if any) with an error code identical to the keyboard abort signal code. PD_QUT is normally set to a control-E character.
PD_BSE	**Backspace "output" character (echo character)** This field indicates the backspace character to echo when PD_BSP is input. See PD_BSP and PD_BSO.

PD_OVF **Line overflow character**

If I$ReadLn has satisfied its input byte count, SCF ignores any further input characters until an end-of-record character (PD_EOR) is received. It echoes the PD_OVF character for each byte ignored. PD_OVF is usually set to the terminal's bell character.

PD_PAR **Parity code, number of stop bits & bits/character**

Bits zero and one indicate the parity as follows:

0 = no parity
1 = odd parity
3 = even parity

Bits two and three indicate the number of bits per character as follows:

0 = 8 bits/character
1 = 7 bits/character
2 = 6 bits/character
3 = 5 bits/character

Bits four and five indicate the number of stop bits as follows:

0 = 1 stop bit
1 = 1 1/2 stop bits
2 = 2 stop bits

Bits six and seven are reserved.

Name **Description**

PD_BAU **Software adjustable baud rate**

This one-byte field indicates the baud rate as follows:

0 =	50 baud	6 =	600 baud	C =	4800 baud
1 =	75 baud	7 =	1200 baud	D =	7200 baud
2 =	110 baud	8 =	1800 baud	E =	9600 baud
3 =	134.5 baud	9 =	2000 baud	F =	19200 baud
4 =	150 baud	A =	2400 baud	10 =	38400 baud
5 =	300 baud	B =	3600 baud	FF =	External

PD_D2P **Offset to output device descriptor name string**

SCF sends output to the device named in this string. Input comes from the device named by the M$PDev field. This permits two separate devices (a keyboard and video display) to be one logical device. Usually PD_D2P refers to the name of the same device descriptor in which it appears.

PD_XON **X-ON character**

See PD_XOFF below.

PD_XOFF **X_OFF character**

The X-ON and X-OFF characters are used to support software handshaking. Output from a SCF device is halted immediately when PD_XOFF is received and will not be resumed until PD_XON is received. This allows the distant end to control its incoming data stream. Input to a SCF device is controlled by the driver. If the input FIFO is nearly full, the driver sends PD_XOFF to the distant end to halt input. When the FIFO has been emptied sufficiently, the driver resumes input by sending the PD_XON character. This allows the driver to control its incoming data stream.

NOTE: When software handshaking is enabled, the driver consumes the PD_XON and PD_XOFF characters itself.

PD_Tab **Tab character**

In I$WritLn calls, SCF expands this character into spaces to make tab stops at the column intervals specified by PD_Tabs. **NOTE:** SCF does not know the effect of tab characters on particular terminals. Tab characters may expand incorrectly if they are sent directly to the terminal.

PD_Tabs **Tab field size**

See PD_Tab.

SCF Path Descriptor Definitions

The first 27 fields of the path options section (PD_OPT) of the SCF path descriptor are copied directly from the SCF device descriptor initialization table. The table is shown on the following page.

The fields can be examined or changed using the I$GetStt and I$SetStt service requests or the tmode and xmode utilities.

You may disable the SCF editing functions by setting the corresponding control character value to zero. For example, if you set PD_INT to zero, there is no "keyboard interrupt" character.

NOTE: Full definitions for the fields copied from the device descriptor are available in the previous section. The additional path descriptor fields are defined below:

Name	Description
PD_TBL	**Device Table Entry** Contains a user-visible copy of the device table entry for the device.
PD_COL	**Current Column** Contains the current column position of the cursor.
PD_ERR	**Most Recent Error Status** Contains the most recent I/O error status.

Offset	Name	Description
$80	PD_DTP	Device Type
$81	PD_UPC	Upper Case Lock
$82	PD_BSO	Backspace Option
$83	PD_DLO	Delete Line Character
$84	PD_EKO	Echo
$85	PD_ALF	Automatic Line Feed
$86	PD_NUL	End Of Line Null Count
$87	PD_PAU	End Of Page Pause
$88	PD_PAG	Page Length
$89	PD_BSP	Backspace Input Character
$8A	PD_DEL	Delete Line Character
$8B	PD_EOR	End Of Record Character
$8C	PD_EOF	End Of File Character
$8D	PD_RPR	Reprint Line Character
$8E	PD_DUP	Duplicate Line Character
$8F	PD_PSC	Pause Character
$90	PD_INT	Keyboard Interrupt Character
$91	PD_QUT	Keyboard Abort Character
$92	PD_BSE	Backspace Output
$93	PD_OVF	Line Overflow Character (bell)
$94	PD_PAR	Parity Code, # of Stop Bits, and # of Bits/Character
$95	PD_BAU	Adjustable Baud Rate
$96	PD_D2P	Offset To Output Device Name
$98	PD_XON	X-ON Character
$99	PD_XOFF	X-OFF Character
$9A	PD_TAB	Tab Character
$9B	PD_TABS	Tab Column Width
$9C	PD_TBL	Device Table Entry
$A0	PD_Col	Current Column
$A2	PD_Err	Most Recent Error Status
$A3		Reserved

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

SCF Device Drivers

SCF device drivers support I/O devices that read and write data one character at a time, such as serial devices.

Generally, the input data (usually from a keyboard) is buffered by the driver's interrupt service routine. Each read request returns one character at a time from the driver's circular input FIFO buffer. If the buffer is empty when the request occurs, the driver must suspend the calling process until an input character is received. Input interrupts are usually enabled throughout the time the device is attached to the system. If the device is incapable of interrupt-driven operation, the driver must poll the device until the data becomes available. This situation has a harmful effect on real-time system performance.

The output data may or may not be buffered, depending on the physical characteristics of the output device. If the device is a memory-mapped video display driven by the main CPU, buffering and interrupts are not usually needed. If the device is a serial interface, use buffering and interrupts. Each write request passes a single output character to the driver which is placed in a circular FIFO output buffer. The output interrupt routine takes output characters from this buffer. If the buffer is full when a write request is made, the driver should suspend the calling process until the buffer empties sufficiently.

The I$GetStt system call (SS_Ready) and I$SetStt system call (SS_SSig) permit an application program to determine if the input buffer contains any data. By checking first, the program is not suspended if data is not available.

The driver may optionally handle full input buffer conditions using X-ON/X-OFF or similar protocols. The input routine must also handle the special pause, abort, and quit control characters. All other control characters (such as backspace, line delete, etc.) are handled at the file manager level.

Special Characters and NULLs

Line-editing functions (if any) are generally dealt with at the file manager level by SCF. Device drivers are, however, required to deal with the following special characters in their input character routine:

- **NULL character**
  The driver's input routine should first determine if the received character is a NULL. If so, it should skip all special character tests, because the disabled state of these special characters is indicated by a NULL in the appropriate path option field. Failure to check for a received NULL results in erratic terminal and/or line-editing operation.

- **Abort and Interrupt Characters**
  The abort and interrupt characters should cause the appropriate signal to be sent to the last process that used the device. The received character should then be buffered.

- **Page Pause**
  The page pause character should cause a page pause request to be set in the echo device's static storage. The received character should then be buffered.

- **Software Flow Control**
  The start and stop transmission characters should cause the resumption/suspension of output data transmission. When this protocol is used, these characters are consumed by the driver's input character routine.

Parity Stripping

SCF device drivers do not usually modify the raw data stream when receiving and transmitting data. The drivers are expected to pass eight-bit data characters "as is." When parity is enabled, however, the driver may have to be sensitive to the issue of "parity stripping."

For eight-bit data characters, parity is not normally an issue (except for error checking), because the character parity status is signalled "out-of-band" from the character itself (there is a parity-error status flag). For smaller sized data characters (for example, seven-bit characters), the hardware sometimes passes the value of the parity bit in the high-bit of the received character. If a driver supports parity checking and non-eight-bit character formats, then the driver's input character routine must be sensitive to the current communications mode and strip the parity flag from the data prior to processing and buffering the character. Failure to strip this parity value from the received character may cause erratic terminal operation (for example, the software flow control characters may not be recognized correctly).

Data Flow Control

Data flow control is the process used to control the transfer of data over the physical interface. It ensures that each end of the connection only transmits data when the other end is capable of receiving data. The data flow may be controlled by either hardware and/or software:

Hardware Flow Control

Hardware flow control uses physical signal lines to indicate the state of the interface. The Ready To Send (RTS) and Clear To Send (CTS) signals on the RS-232 Standard Interface are examples of these physical lines.

The level of implementation of hardware handshaking in a SCF driver is determined by the capabilities of the serial interface itself, which include the capabilities of the interface-chip and the board-level implementation of the interface.

A driver that implements fully functional hardware flow control performs the following functions:

- Configures the transmitter to only send data when the distant end's "ready-to-receive" is active.
- Controls the distant end's "ready-to-transmit" line so that input buffer over-runs do not occur.
- Supports the SS_EnRTS, SS_DsRTS, SS_DCDOn, and SS_DCDOff SetStat calls, to allow a user application to directly control/monitor the serial connection.

A driver that provides minimal (or no) support for hardware flow control usually configures the hardware control lines so that the interface is "ready" whenever the device is attached. Drivers that provide this level of operation usually implement software flow control.

Software Flow Control

Software flow control uses a software protocol to indicate the "ready" state of the two ends of the interface.

Support for software flow control is provided via the PD_XON (start transmission) and PD_XOFF (stop transmission) fields of the device descriptor. When these fields are enabled (both non-zero), then the driver implements the protocol as follows:

- If the driver receives the stop transmission character, it should immediately suspend data transmission. The driver can resume transmission when a start transmission character is received. Thus, the distant end is allowed to control its incoming data stream.

- If the driver's input routine detects that its input buffer is about to fill, then it causes a stop transmission character to be sent to the distant end. When the buffer has been sufficiently emptied, the driver can cause transmission of a start transmission character. Thus, the driver is capable of controlling its incoming data stream.

When implementing software flow control, note the following points:

- The start transmission and stop transmission characters are *consumed* by the driver's input routine. If pure binary transfers are desired (the character values for flow control are actually part of the data stream), then software flow control must be disabled and hardware flow control enabled.
- Software flow control only works reliably with interrupt-driven drivers, because the detection of the incoming stop transmission character must take place immediately.
- The characters involved with the protocol must be "agreed upon" by both ends of the connection. Most systems default to the ASCII control characters X-ON and X-OFF. However, any other pair of characters may be used if both ends concur.
- When controlling the input data, the driver's input routine and Read routine will cooperate in the protocol as follows:
  - The input routine detects a "high-water" mark; a point at which the input buffer is almost full. When this mark is reached (ten characters remaining in buffer), the input routine causes the stop transmission character to be sent. The "head room" provided by the high-water mark should be set so that the distant end has time to suspend transmission before the buffer actually fills.
  - The Read routine simply takes characters from the input buffer until the buffer count reaches the "low-water" mark. Then, the Read routine causes the start transmission character to be sent to resume input. The low-water mark is usually set to a low value to keep the total overhead in the software flow control to a minimum.

SCF Device Driver Storage Definitions

SCF device driver modules contain a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical I/O controllers.

The kernel allocates a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header (M$Mem). Some of this storage area is required by the kernel and SCF; the device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the scfstat.a DEFS file. Static storage is used as follows:

Offset	Name	Maintained By	Description
$00	V_PORT	Kernel	Device base address
$04	V_LPRC	File Manager	Last active process ID
$06	V_BUSY	File Manager	Active process ID
$08	V_WAKE	Driver	Process ID to awaken
$0A	V_Paths	Kernel	Linked list of open paths
$0E			Reserved
$2E	V_DEV2	Kernel	Addr. of attached device static storage
$32	V_TYPE	File Manager	Device type or parity
$33	V_LINE	File Manager	Lines left until end of page
$34	V_PAUS	Driver/File Man.	Pause request
$35	V_INTR	File Manager	Keyboard interrupt character
$36	V_QUIT	File Manager	Keyboard abort character
$37	V_PCHR	File Manager	Pause character
$38	V_ERR	Driver	Error accumulator
$39	V_XON	File Manager	X-ON character
$3A	V_XOFF	File Manager	X-OFF character
$3B			Reserved
$3C	V_Presvd		Reserved
$46	V_Hangup	Driver/File Man.	Path lost flag
$54			Device Driver Variables begin here

NOTE: ***Offset*** refers to the location of a static storage field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l.

Name	Description
V_PORT	**Device base address** The device's physical port address. It is copied from M$Port in the device descriptor when the device is attached by the kernel.
V_LPRC	**Last active process ID** The process ID of the last process to use the device. The IRQ service routine sends this process the proper signal when an interrupt or quit character is received.
V_BUSY	**Current active process** The process ID of the process currently using the device. It is used to implement I/O Blocking by SCF. This field is also used by the interrupt drivers when they wish to suspend themselves, by copying V_BUSY to V_WAKE (prior to suspending themselves). A value of zero indicates the device is not busy.
V_WAKE	**Process ID to awaken** The process ID of any process that is waiting for the device to complete I/O. A value of zero indicates that no process is waiting. V_WAKE is set by the driver from V_BUSY and provides the interlock between the driver and the driver's interrupt service routine.
V_PATHS	**Linked list of open paths** A singly-linked list of all paths currently open on this device.
V_DEV2	**Attached device static storage** The address of the echo (output) device's static storage area. A device is typically its own echo device, but may not be, as in the case of a keyboard and a memory mapped video display. The interrupt service routine uses this pointer to set an output pause request (see V_PAUS and V_PCHR). If the value in V_DEV2 is zero, there is no echo device.
V_TYPE	**Device type or parity** This value is copied from PD_PAR in the path descriptor by SCF, so that it may be used by interrupt service routines, if required.
V_LINE	**Lines left until end of page** The number of lines left until the end of the page. Paging is handled by SCF.

Name	Description
V_PAUS	**Pause request** A flag used to signal SCF that a pause character has been received. Setting its value to anything other than 0 causes SCF to stop transmitting characters at the end of the next line. Device driver input routines must set V_PAUS in the echo device's static storage area. SCF checks this value in the echo device's static storage when output is sent. Once paused, SCF clears any type-ahead (I$ReadLn), waits for and consumes the next input character, clears V_PAUS, and resumes output (see V_DEV2 and V_PCHR).
V_INTR	**Keyboard interrupt characters** This value is copied from PD_INT in the path descriptor by SCF, so that it may be used by the driver's input routine. Receipt of this character should cause a signal (S$Intrp) to be sent to the last user of the device (V_LPRC).
V_QUIT	**Quit character** This value is copied from PD_QUT in the path descriptor by SCF so that it may be used by the driver's input routine. Receipt of this character should cause a signal (S$Quit) to be sent to the last user of the device (V_LPRC).
V_PCHR	**Pause character** This value is copied from PD_PSC in the path descriptor by SCF, so that it may be used by the driver's input routine. When the input routine receives this character, it should set the output pause request flag (V_PAUS) in the echo device's static storage (V_DEV2). (See V_DEV2 and V_PAUS.)
V_ERR	**Error accumulator** This location is used to accumulate I/O errors. Typically, the IRQ service routine uses it to record input errors so that they may be reported later when SCF calls the device driver read routine.
V_XON	**X-ON character** This character is copied from PD_XON of the path descriptor by SCF, so that it may be used for software handshaking by interrupt service routines, if required.
V_XOFF	**X-OFF character** This character is copied from PD_XOFF of the path descriptor by SCF, so that it may be used for software handshaking by interrupt service routines, if required.
V_Hangup	**Path Lost Flag** This flag should be set to a non-zero value when the driver detects that the path has been lost (for example, carrier lost on a modem).

Linking SCF Drivers

After a SCF driver has been assembled into its relocatable object file (ROF), the driver needs to be linked to produce the final driver module. Linking resolves all code references in drivers that are comprised of several ROF files. It also resolves the external data and static storage references by the driver.

The most important part of linking is to correctly resolve the static storage references. Generally, the static storage area is composed of two sections, in this order (see Figure 3-1):

- I/O globals
- Driver-declared variables

The driver-declared variables are declared in vsect areas of the driver, but they *must* be allocated after the I/O globals. To allocate all of the storage, *in the correct order*, the scfstat.l *must* be the first module specified. The scfstat.l file is usually found in the system's LIB directory. The following is a typical linker command line for an SCF driver:

l68 /dd/LIB/scfstat.l REL/sc335.r -O=OBJS/sc335

NOTE: Failure to link the I/O global storage first, or not at all, results in erratic driver operation.

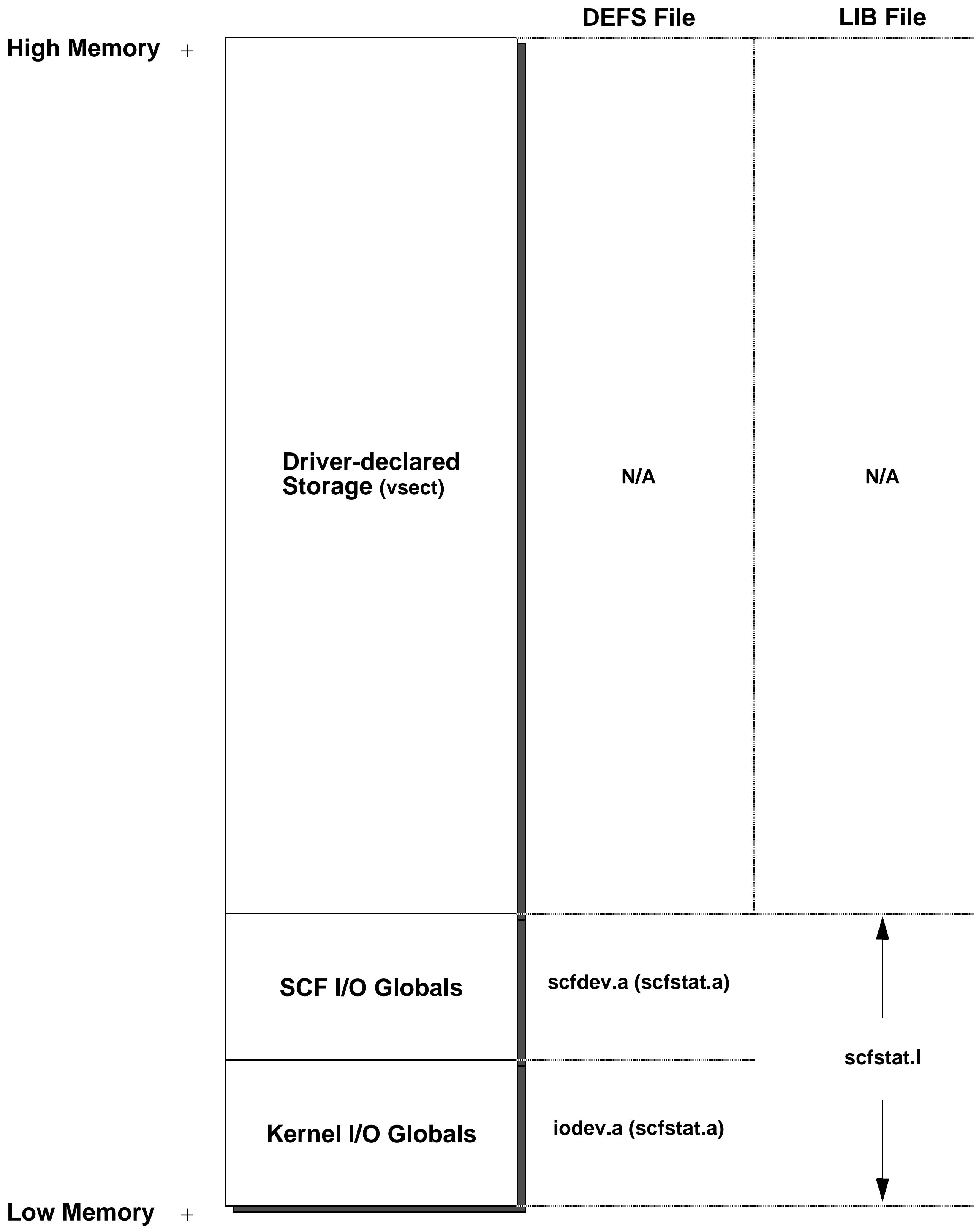


Figure 3-1: SCF Static Storage Layout

SCF Device Driver Subroutines

As with all device drivers, SCF device drivers use a standard executable memory module format with a module type of Drivr (code $E0). SCF drivers are called in system state.

NOTE: I/O system modules must have the following module attributes:

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

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of the corresponding subroutine. The branch table appears as follows:

```
ENTRY dc.w INIT        initialize device
      dc.w READ        read character
      dc.w WRITE       write character
      dc.w GETSTAT     get device status
      dc.w SETSTAT     set device status
      dc.w TERM        terminate device
      dc.w TRAP        handle illegal exception (0 = none)
```

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, set the carry bit and return an appropriate error code in the least significant word of register d1.w.

The TRAP entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

The following pages describe each subroutine.

INIT

Initialize Device and its Static Storage

INPUT: **(a1) = address of device descriptor module**
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The INIT routine must:

¨ Initialize the device static storage.

¦ Initialize the device control registers.

Æ Place the driver IRQ service routine on the IRQ polling list by using the F$IRQ service request, if required.

Ø Enable interrupts if necessary.

Prior to being called, the device static storage is cleared (set to zero) except for V_PORT which contains the device port address. Do not initialize the portion of static storage used by SCF.

If INIT returns an error, it does not have to clean up its operation, for example, remove device from polling table or disable hardware. The kernel calls TERM to allow the driver to clean up INIT's operation before returning to the calling process.

NOTE: If the INIT routine causes an interrupt to occur, the interrupt can be handled in one of the following ways:

- Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, and then restoring the previous interrupt level. This is the preferred technique unless the interrupt is time-consuming.

- Allow the interrupt service routine to service the hardware. In this case, the process descriptor contains the process ID (P$ID) to which V_WAKE should be set. V_BUSY cannot be used because it is zero when INIT is called.

READ Get Next Character

INPUT: **(a1) = address of path descriptor**
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data pointer

OUTPUT: **d0.b = input character**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: This routine returns the next character available. Depending upon whether or not the routine is interrupt-driven, READ typically operates as follows:

Polled I/O Mode

A polled I/O read routine checks the hardware for available data. If there is none, the routine must wait until data is available. When data is available, READ should strip parity (if required) and then determine whether or not the character requires special handling:

- If the character is the output pause character (V_PCHR), READ sets a pause request (V_PAUS) in the echo device's static storage (V_DEV2).
- If the character is a keyboard interrupt (V_INTR) or quit (V_QUIT) character, READ sends the appropriate signal to the last process to use the device (V_LPRC).

NOTE: If the received character is a NULL character, then special character tests should be ignored.

NOTE: Software handshaking, as specified by V_XON/V_XOFF is not usually implemented for polled-mode I/O, as the lack of interrupt-driven operation makes this handshake feature unreliable. Polled I/O drivers can usually only perform hardware handshaking.

The character read is returned to SCF in register d0.

Interrupt I/O Mode

For interrupt-driven drivers, READ gets data from the driver's input FIFO buffer. This buffer is filled by the input interrupt service routine. The following describes how READ operates.

- READ determines if another process has set up a "send signal on data ready"

condition. If so, READ returns a “not ready” (E$NotRdy) error (the device is busy for reading, but not for writing).

¦ READ then determines if data is available in the input FIFO buffer. If not, the driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and then performing an F$Sleep service request to put itself to sleep indefinitely.

When the driver awakens, either data is available in the FIFO or a signal occurred. If a signal occurred, either the signal value is in P$Signal (process descriptor) or the process is condemned (condemn bit set in P$State). If the process is condemned or the signal value is deadly to I/O (less than S$Deadly), then the driver should return immediately to SCF with the carry bit set and the signal code (if any) as the error code.

Æ READ should get the next character from the input FIFO.

Ø If software handshaking is implemented, READ should determine if input has been halted (V_XOFF sent to distant end). If so, and reading this character causes the FIFO count to go below the “low-water mark” of the FIFO, then resume input by sending a V_XON character to the distant end and flagging input resumed.

× READ should determine if any errors have been logged by the input interrupt service routine (V_ERR). If so, READ returns an error (E$Read) to SCF and clears V_ERR. Otherwise, READ returns the character read to SCF in register d0.

NOTE: Data buffers for queueing data between the main driver and the IRQ service routine are *not* automatically allocated by SCF. They should be defined in the device driver’s static storage area (vsect) or allocated dynamically by the driver (for example, at INIT call).

NOTE: Normally, READ should not have to enable the device’s “data-buffer-full” interrupt. The device should normally be configured so that any input while the device is attached causes an interrupt. This is usually done during INIT. Input interrupts are typically disabled only when the device is detached (TERM routine).

WRITE Output a Character

INPUT:
d0.b = character to write
(a1) = address of the path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT:
cc = carry bit set
d1.w = error code

FUNCTION: The WRITE routine writes a character. Depending upon whether or not the routine is interrupt-driven, WRITE typically operates as follows:

Polled I/O Mode

A polled I/O driver checks the hardware for "ready-to-transmit". When ready, the character is written to the hardware and the driver returns to SCF without an error.

Interrupt I/O Mode

For interrupt-driven drivers, WRITE attempts to put the character into the driver's output FIFO buffer and then ensures that output interrupts are enabled. The driver's output interrupt service routine empties the output FIFO. WRITE operates as follows:

¨ WRITE determines if space is available in the output FIFO buffer. If not, the device driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and then performing a F$Sleep service request to put itself to sleep indefinitely.

When the driver awakens, either space is available in the output FIFO or a signal occurred. If a signal occurred, either the signal value is in P$Signal (process descriptor) or the process is condemned (condemn bit set in P$State). If the process is condemned or the signal value is deadly to I/O (less than S$Deadly), the driver should return immediately to SCF with the carry bit set and the signal code (if any) as the error code.

¦ WRITE puts the character into the output FIFO buffer.

Æ WRITE determines if output interrupts are currently enabled. If so, this implies that output is currently active (using the output IRQ service routine) and the driver can simply return to SCF without an error.

Ø If output interrupts are disabled, then output is halted due to software handshaking (V_XOFF received from distant end) or a previously empty output

FIFO. If output is halted due to software handshaking, the driver should return to SCF without an error. Otherwise, the driver should enable output interrupts on the device (allowing the output interrupt service routine to empty the output FIFO) and return to SCF without an error.

NOTE: Data buffers for queueing data between the main driver and the IRQ service routine are *not* automatically allocated by SCF. They should be defined in the device driver's static storage area (vsect) or allocated dynamically by the driver (for example, at INIT call).

NOTE: Typically, this routine should ensure that output interrupts are enabled only when necessary. After an output interrupt is generated, the IRQ service routine continues to transmit data until the output FIFO is empty and then it typically disables the device's "ready-to-transmit" interrupts.

This dynamic enabling/disabling of the device's transmit interrupts is essential to some serial devices, as the "transmit ready" interrupt is generated every "character period" (that is, at the device's baud rate), regardless of whether a character is actually transmitted. This type of situation leads to excessive and unnecessary overhead to the system, and should be avoided.

GETSTAT/SETSTAT — Get/Set Device Status

INPUT:
d0.w = function code
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data pointer

OUTPUT: **Depends upon function code**

ERROR OUTPUT:
cc = carry bit set
d1.w = error code

FUNCTION: These routines are wild-card calls used to get/set the device's operating parameters as specified for the I$GetStt and I$SetStt service requests.

Calls which involve parameter passing require the driver to examine or change the register stack variables. These variables contain the contents of the MPU registers at the time the I$GetStt/I$SetStt request was made. Parameters passed to the driver are set up by the caller prior to using the service call. Parameters passed back to the caller are available when the service call completes.

Typical SCF drivers handle the following I$GetStt/I$SetStt calls:

I$Getstt: SS_EOF, SS_Opt, SS_Ready

I$SetStt: SS_Break, SS_DCOff*, SS_DCOn*, SS_DsRTS, SS_EnRTS, SS_Open, SS_Opt, SS_Relea*, SS_SSig*,

* only for interrupt-driven drivers

Any unsupported I$GetStt/I$SetStt calls to the driver should return an unknown service error (E$UnkSvc).

NOTE: A minimal SCF driver should support SS_Ready and SS_EOF, and if interrupt-driven, SS_SSig.

The following pages describe the driver's role in the implementation of the above I$GetStt/I$SetStt calls.

GetStat Calls:

SS_EOF This routine should exit without an error.

SS_Opt This routine is called when SCF is asked to return the current path options. SCF calls the driver so that the driver can update the path descriptor's baud rate (PD_BAU) and communications mode (PD_PAR) to the current hardware values. This function is usually done by drivers that support dynamic changes to baud rate, etc. Drivers that do not support these changes typically return an unknown service request error (E$UnkSvc).

SS_Ready This routine returns the current count of data available in the input FIFO buffer. If data is available, the count should be returned in the caller's d1 register (R$d1 offset from passed a5) and the driver should return to SCF without an error. If no data is available, then a "not ready" error (E$NotRdy) should be returned to SCF.

SetStat Calls:

SS_Break This routine is called when an application wishes to assert a "break" condition on the outgoing serial line.

SS_DCOff
SS_DCOn These routines are called when you wish to notify an application that the Data Carrier has been asserted(SS_DCOn) or negated (SS_DCOff). Typically, this routine saves the process ID (PD_CPR), path number (PD_PD), and signal code (user's d2 register) in static storage and then returns without error. The IRQ service routine detects the presence or loss of the Data Carrier, sends the signal, and clears down the signal condition.

Drivers which have hardware detection of a change-of-state only on the Data Carrier line typically have to track the current state (asserted or negated) of the line and signal a change of state accordingly.

NOTE: Only interrupt-driven drivers should implement these calls.

SS_DsRTS These routines are called by applications that wish to SS_EnRTS explicitly assert (SS_EnRTS) or negate (SS_DsRTS) the RTS handshake line. Typically, the driver performs the hardware action and returns without an error.

SS_Open This routine is called by SCF whenever a new path to the device is opened. Typically, drivers handle this call in the same way as a SetStat (SS_Opt) call, i.e. check for baud-rate, configuration mode changes.

SS_Opt This routine is called when SCF is asked to change the current path options. SCF passes the call to the driver so that it may implement baud-rate, configuration mode, etc., changes to the hardware. Typically, the driver checks PD_BAU and PD_PAR to determine if they have changed. If not, the driver simply returns without an error. If one or both of these have changed, the driver validates the requested change and if correct, implements the change in hardware (for example, new baud rate). If the request is for an unsupported or illegal I/O mode (for example, invalid stop-bit count), then the driver typically returns a “bad I/O mode” error (E$BMode) and refuses the change.

SS_Relea This routine is called when either SCF or an application wishes to clear down device signalling. This routine should erase any pending signal conditions (due to SS_SSig, SS_DCOn, SS_DCOff) and return without an error.

NOTE: When clearing down the signal condition(s), the driver should only clear the signal if the process ID (PD_CPR) and path number (PD_PD) of the caller match the process ID and path number of the original set-up call.

SS_SSig This routine is called when applications wish to have a signal sent to them when input data is available. Typically, the routine operates as follows:

¨ It determines if another process has set up a SS_SSig condition. If so, a “not ready” error (E$NotRdy) is returned.

¦ It determines if data is available in the input FIFO buffer. If so, the specified signal (user’s d2 register value) is sent to the process (PD_CPR) and the routine returns.

Æ If no data is available, the process ID, path number (PD_PD), and signal are saved in static storage and the routine simply returns. When the data arrives, the input IRQ service routine sends the signal and releases the send-signal condition.

NOTE: Setting up a “send signal on data ready” condition will “busy” the driver for read requests (see READ description), but allow writes to proceed as normal.

NOTE: Only interrupt-driven drivers should implement this call.

TERM — Terminate Device

INPUT: **(a1) = device descriptor pointer**
(a2) = pointer to device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: This routine is called when a device is no longer in use in the system (see I$Detach).

The TERM routine must:

¨ Copy the process ID from the process descriptor (P$ID) into V_BUSY and V_LPRC.

¦ Determine if the output FIFO buffer contains any data waiting to be written. If so, the driver should suspend itself by copying its process ID from V_BUSY to V_WAKE and performing an F$Sleep service request to put itself to sleep indefinitely.

If the driver awakens before the output FIFO has emptied (due to a signal), the driver should suspend itself again until the buffer is empty.

Æ After the pending output data has been written, the driver should disable hardware handshake protocols and then disable all device interrupts, if the driver is interrupt-driven. The device should then be removed from the system's IRQ polling table (F$IRQ), if applicable.

Ø Return any buffers the driver has requested on behalf of itself. **NOTE:** The driver should not attempt to return buffers within its defined static storage area. The kernel releases this memory when the TERM routine completes.

NOTE: If an error occurs during the device's INIT routine, the kernel calls the TERM routine to allow the driver to clean up. If the TERM routine uses static storage variables (for example, interrupt mask values, dynamic buffer pointers), it should validate these variables prior to using them. The INIT routine may not have set up all the variables prior to exiting with the error.

IRQ Service Routine — Service Device Interrupts

INPUT: **(a2) = static storage**
(a3) = port address
(a6) = system global static storage

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set (interrupt not serviced)**

FUNCTION: This routine is called directly by the kernel's IRQ polling table routines. Its function is to:

¨ Check the device for a valid interrupt. If the device does not have an interrupt pending, the carry bit must be set and the routine exited with an RTS instruction as quickly as possible. Setting the carry bit signals the kernel that the next device on the vector should have its IRQ service routine called.

¦ Service device interrupts. There are three categories of interrupts: control interrupts, input interrupts, and output interrupts. Usually, input interrupts are checked first, because most serial hardware devices have minimal (or no) hardware data buffering. After the interrupt is serviced, many drivers check for another pending interrupt prior to exiting to the kernel. This technique (for example, service input interrupt, service pending output interrupt, service next input interrupt) provides efficient interrupt servicing because it allows the driver to service multiple interrupts with one call to the IRQ service routine.

Æ Clear the carry bit and exit with a RTS instruction after servicing an interrupt.

Avoid exception conditions (for example, a Bus Error) when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

NOTE: IRQ service routines may destroy the contents of the following registers only: d0, d1, a0, a2, a3, and a6. You must preserve the contents of all other registers or unpredictable system errors (system crashes) will occur.

The interrupt categories (control, input, and output) are described in the following pages.

Control Interrupts

These interrupts are usually associated with non-data type information on the serial port, such as the receipt of a break character or a change in the Data Carrier line. Control interrupts may also signal error conditions on the data stream (for example, parity error).

When signaling is set up for Data Carrier transactions (see SetStat, SS_DCOn, SS_DCOff), the routine should send the specified signal to the specified process, clear down the signal condition, mark the path as “lost” (V_HangUp set to non-zero), and then exit (carry bit clear) or service more interrupts.

Input Interrupts

The input interrupt routine typically performs the following:

¨ Read the character from the hardware, clear down the interrupt, and strip parity (if required).

¦ Check character error status. If in error, update V_ERR to indicate the error.

Æ If the character is not a NULL character, determine whether or not the character requires special handling.

a) If the character is the output pause character (V_PCHR), set a pause request (V_PAUS) in the echo device’s static storage (V_DEV2).

b) If the character is a keyboard interrupt (V_INTR) or quit character (V_QUIT), send the appropriate signal to the last process to use the device (V_LPRC).

c) If the character is a software handshake character (V_XON or V_XOFF), service the handshake request. For an “output resume” case (V_XON), this typically involves clearing the “output halted due to X-OFF” flag, checking for data in the output FIFO, and enabling output interrupts, if so. For an “output halt” case (V_XOFF), this typically involves setting the “output halted due to X-OFF” flag and disabling output interrupts on the hardware.

NOTE: The software handshake characters are consumed by this routine. After processing these characters, the IRQ service routine exits to the kernel (carry bit clear) or services the next pending device interrupt.

Ø Put the character into the input FIFO buffer. If there is no room in the buffer, the character is lost and the driver should indicate “input buffer overrun” in the accumulated error status (V_ERR). In this case, the driver often returns to the kernel at this point, after waking the driver process (V_WAKE).

× Determine if any process has set up a “send signal on data ready” condition (SS_SSig). If so, signal the process, clear down the signaling condition, and exit (carry bit clear) or service the next pending interrupt.

± Examine the number of characters in the input FIFO, if the driver supports handshaking.

For software handshaking, if the buffer is nearly full (reached the “high-water mark”), the driver should send a suspend transmission character (V_XOFF) to the distant end and flag that input has been halted. This function allows the driver to prevent input FIFO overrun errors when the data is being received at a faster rate than it is being read from the FIFO. Typically, the READ routine re-enables input data flow when it has emptied the input FIFO to a suitable low value (“low-water mark”) by causing the V_XON character to be sent.

For hardware handshaking, the input interrupt routine should signal its desire to suspend input by negating its “ready to receive” line.

ð If desired, the input IRQ service routine can now service more interrupts. Once fully completed, it should exit to the kernel with the carry bit clear. Prior to exiting, it should send a wake-up signal (S$Wake) to any waiting driver process. You can find the process ID in V_WAKE, which you should clear.

Output Interrupts

The output interrupt routine typically performs the following:

¨ Determine if V_XON or V_XOFF is pending, due to input buffer software handshaking. If so, send the required character, flag it sent, and mark the current state of input (halted or resumed). The driver should then determine if output is currently halted (buffer empty or software handshake). If so, it should disable output interrupts and return to the kernel (carry bit clear). If not, further interrupts may be processed or an exit may be made to the kernel (carry bit clear).

¦ Determine if output is halted due to software handshaking. If so, disable output device interrupts and return to the kernel (carry bit clear).

Æ Determine if any data is waiting in the output FIFO for transmission. If so, write the data to the hardware.

Ø Determine the remaining data count in the output FIFO.

a) If zero, flag the buffer empty, disable output device interrupts, wake any waiting process (V_WAKE) and exit to the kernel (carry bit clear).

b) If not zero, check if current count is below the output buffer's "low-water mark". If not, exit to the kernel (carry bit clear) *without* waking the driver process. If so, wake the driver process before exiting.

This technique minimizes contention between the driver's WRITE routine (filling the output buffer) and the output IRQ service routine (emptying the output buffer), as the buffer is allowed to empty significantly before the WRITE process is re-activated.

End of Chapter 3

Sequential Block File Manager (SBF)

SBF General Description

The Sequential Block File Manager (SBF) is a re-entrant subroutine package for I/O service requests to sequential block-oriented mass storage devices, such as tape systems. SBF can handle any number or type of such systems simultaneously.

The following I/O service requests are handled by SBF:

I$Close	I$Create	I$GetStt	I$Open	I$Read
I$ReadLn	I$SetStt	I$Write	I$Writln	

The following I/O service requests are not valid for SBF:

I$ChgDir   I$Delete   I$MakDir   I$Seek

When one of these service requests is made to SBF, an appropriate error code is returned.

The following I/O service requests do not call SBF:

I$Attach   I$Detach   I$Dup

SBF is designed to support both buffered and unbuffered I/O. It is capable of handling variable logical block sizes. SBF has no knowledge of the media's physical block size, and the driver is responsible for translating the logical block requests by SBF into the media's physical block requests. The logical block size for an SBF device is defined in the PD_BlkSiz field of the path descriptor.

Unbuffered I/O

Unbuffered I/O is used when the PD_NumBlk field of the path descriptor is set to 0.

When operating in unbuffered mode, SBF uses a single buffer for I$ReadLn and I$Writln calls. I$Read and I$Write calls do not use an intermediate buffer, and the data is transferred directly between the caller's data buffer and the driver.

Unbuffered I/O operates synchronously with the requesting process. The process makes a read or write request and SBF returns to the caller when the I/O operation has completed.

Buffered I/O

Buffered I/O is used when the PD_NumBlk field of the path descriptor is set to a positive number. All buffered I/O is initiated asynchronously by an auxiliary process created by SBF. SBF uses a "pool" of buffers to accomplish this. The maximum number of buffers to use is specified by the PD_NumBlk field of the path descriptor. The size of each buffer is specified by the PD_BlkSiz field of the path descriptor.

I$Read requests cause SBF to copy data from the buffer pool. If a full buffer is not yet available, SBF allocates a new buffer and passes it to the auxiliary process. SBF then waits for the auxiliary process to return the buffer containing the next block. Multiple buffers (up to the number specified by PD_NumBlk) may be allocated, thus allowing SBF to copy data from one buffer while the auxiliary process reads data into others.

I$Write requests cause SBF to copy data into a buffer and return to the user immediately. When a buffer fills, SBF passes it to the auxiliary process for writing. If another buffer is required before the auxiliary process has had time to write the previous buffer, SBF allocates a new buffer and copies data to it. This allows SBF to copy data into one buffer while the auxiliary process writes from others.

Considerations When Writing to Tapes

When an SBF path is opened, any I/O operations may be done on the path. However, after an I$Write call is made, SBF flags the path as "in write mode" and will not allow any I$Read calls until an I$SetStt call is made. Typically, when writing a tape, an I$Close call follows an I$Write call and SBF performs its normal close processing. When an I$SetStt call follows an I$Write call, SBF waits for any pending writes to complete, clears the write mode flag, and performs the I$SetStt. It is recommended that I$SetStt writes one or more filemarks, to ensure that a filemark follows the data written.

End-Of-Tape Processing

There is no “end-of-tape” error on Read requests. Consequently, SBF requires an end-of-file mark to be present or the user process to handle the situation (to know the size of the file or use an end-of-data record).

I$Write requests return a media full error (E$Full) when end-of-tape is reached. All prior writes will have completed; no other data may be written other than filemarks after the end-of-tape has been reached.

SBF I/O Service Requests

When a process makes one of the following system calls to an SBF device, SBF executes the file manager functions described for that call.

I$Close SBF performs the following functions:

- If the use count for the path is non-zero (other processes are still using this path), SBF does not return an error.
- If the use count is zero, SBF determines if the path is in write mode. If so, SBF calls the device driver to write two filemarks to the tape.
- If the path is in write mode and the f_eras_b flag is set in the PD_Flags field of the path descriptor, SBF calls the device driver to erase to the end of the tape.
- If the f_rest_b flag is set in PD_Flags, SBF calls the device driver to rewind the tape. If the path is in write mode and f_rest_b is not set, SBF calls the device driver to skip back one filemark. This positions the tape between the two filemarks just written.
- If the f_offl_b flag is set in PD_Flags, SBF calls the device driver to take the tape drive off-line.
- Any buffers associated with the path are returned to the system.

I$Create SBF considers I$Create to be synonymous with I$Open.

I$GetStt Refer to the I$GetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the SBF-supported I$GetStt functions:

SS_Ready	Test for data ready.
SS_EOF	Check for end of file condition.

All other GetStat calls are passed to the driver.

I$Open SBF performs the following functions:

- Validates the pathname.
- Verifies that the drive number (PD_TDrv) is legal for the device driver (SBF_NDRV).
- Initializes path descriptor variables.
- Creates the auxiliary process for the driver (SBF_DPrc), if required.

I$Read SBF calls the driver as needed to read the data. Complete blocks of data are transferred directly to the user's buffer while incomplete blocks are transferred into SBF's buffer. The portion of the data requested by the calling process is copied into the calling process' buffer. If buffers are required for the read (for example, buffered I/O mode), these are allocated as required.

I$ReadLn I$ReadLn is similar to I$Read, except that SBF stops the read if an end-of-record character (carriage return) is found. I$ReadLn requests always transfer the data through an intermediate SBF buffer.

I$SetStt Refer to the I$SetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the SBF supported I$SetStt functions:

SS_Opt Write the path descriptor options.

All other SetStat calls are passed to the driver. If the block size (PD_BlkSiz) has changed, SBF ensures that all current buffers are flushed prior to calling the device driver. **NOTE:** Only SS_Opt is passed to the driver after processing by SBF. If an unknown service request error (E$UnkSvc) is returned by the driver, it is ignored.

I$Write SBF calls the driver as needed to transfer the data as follows:

Buffered I/O

SBF copies the user's data into the next free buffer in the SBF buffer pool. The user process is reactivated immediately. As each buffer fills (PD_BlkSiz), SBF calls the driver to write the data when the driver is available.

Unbuffered I/O

SBF calls the driver with the data pointer pointing to the user's data buffer. The driver writes the data to tape; the user process is reactivated when the driver completes the write operation.

I$Writln I$Writln is similar to I$Write, except that SBF only writes data up to and including the first end-of-record character (carriage return), if there is one in the calling process's buffer. If no end-of-record character is found, SBF writes the amount of data specified by the calling process. I$Writln requests always transfer the data through an intermediate SBF buffer.

SBF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for SBF devices. The initialization table immediately follows the standard device descriptor module header fields. The size of the table is defined in the M$Opt field.

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Type
$49	PD_TDrv	Tape Drive Number
$4A	PD_SBF	Reserved
$4B	PD_NumBlk	Maximum Number of Blocks to Allocate
$4C	PD_BlkSiz	Logical Block Size
$50	PD_Prior	Driver Process Priority
$52	PD_SBFFlags	SBF Path Flags
$53	PD_DrivFlag	Driver Flags
$54	PD_DMAMode	Direct Memory Access Mode
$56	PD_ScsiID	SCSI Controller ID
$57	PD_ScsiLUN	LUN on SCSI Controller
$58	PD_ScsiOpts	SCSI Options Flags

NOTE: In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, the following adjustment must be made: (M$DTyp - PD_OPT).

For example, to access the tape drive number in a device descriptor, use the following value: PD_TDrv + (M$DTyp - PD_OPT). To access the tape drive number in the path descriptor, use PD_TDrv. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Name	Description
PD_DTP	**Device class** This field is set to three for SBF devices.
PD_TDrv	**Tape Drive number** Used to associate a one-byte integer with each drive that a controller will handle. If using dedicated (for example, non-SCSI bus) controllers, this field usually defines both the *logical* and *physical* drive number of the tape drive. If using tape drives connected to SCSI controllers, this number defines the *logical* number of the tape drive to the device driver. The *physical* controller ID and LUN are specified by the PD_ScsiID and PD_ScsiLUN fields. Each controller's drives should be numbered 0 to n-1 (n is the maximum number of drives the controller can handle). This number also defines how many drive tables are required by the driver and SBF. SBF verifies this number against SBF_NDRV prior to calling the driver.
PD_NumBlk	**Number of Buffers/Blocks Used For Buffering** Specifies the maximum number of buffers to be allocated by SBF for use by the auxiliary process in buffered I/O. If this field is set to 0, unbuffered I/O is specified.
PD_BlkSiz	**Logical Block Size Used For I/O** Specifies the size of the buffer to be allocated by SBF. This buffer size is used when allocating multiple buffers used in buffered I/O. Unless the driver manages partial physical blocks, this size should be an integer multiple of the physical tape block size.
PD_Prior	**Driver Process Priority** The priority at which SBF's auxiliary process will run. This value is used during initialization. Changing this value after initialization has no effect.
PD_SBFFlags	**SBF Path Flags** Specifies the actions that SBF takes when the path is closed. A user can update this field using GetStat/SetStat (SS_Opt). SBF supports the following flag definitions: bit 0: (f_rest_b) 0 = No rewind on close. 1 = Rewind on close. bit 1: (f_offl_b) 0 = Do not put drive off-line on close. 1 = Put drive off-line on close. bit 2: (f_eras_b) 0 = Do not erase to end-of-tape on close. 1 = Erase to end-of-tape on close.

Name	Description
PD_DrivFlag	**Driver Flags** This field is available for use by the device driver. **NOTE:** References to these flags are often made using the PD_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD_SBFFlags. References to PD_DrivFlag should use a value of PD_Flags + 1.
PD_DMAMode	**Direct Memory Access Mode** This field is hardware specific. If available, you can use this word to specify the DMA Mode of the driver.
PD_ScsiID	**SCSI Controller ID** This is the ID number of the SCSI controller attached to the device. The driver uses this number when communicating with the controller.
PD_ScsiLUN	**Logical Unit Number of SCSI Device** This number is the value to use in the SCSI command block to identify the logical unit on the SCSI controller. This number may be different from PD_TDrv, to eliminate allocation of unused drive table storage. PD_TDrv indicates the logical number of the drive to the driver and SBF (drive table to use). PD_ScsiLUN is the physical drive number on the controller.
PD_ScsiOpts	**SCSI Driver Options Flags** This field allows SCSI device options and operation modes to be specified. It is the driver's responsibility to use or reject these if applicable: bit 0: 0 = ATN not asserted (no disconnects allowed). 1 = ATN asserted (disconnects allowed). bit 1: 0 = Device cannot operate as a target. 1 = Device can operate as a target. bit 2: 0 = asynchronous data transfers. 1 = synchronous data transfers. bit 3: 0 = parity off. 1 = parity on. All other bits are reserved.

SBF Path Descriptor Definitions

The reserved section (PD_OPT) of the path descriptor used by SBF is copied directly from the initialization table of the device descriptor. The following table is provided to show the offsets used in the path descriptor. For a full explanation of the path descriptor fields, refer to the previous pages.

Offset	Name	Description
$80	PD_DTP	Device Type
$81	PD_TDrv	Tape Drive Number
$82	PD_SBF	Reserved
$83	PD_NumBlk	Maximum Number of Blocks to Allocate
$84	PD_BlkSiz	Logical Block Size
$88	PD_Prior	Driver Process Priority
$8A	PD_SBFFlags*	SBF Path Flags
$8B	PD_DrivFlag*	Driver Flags
$8C	PD_DMAMode	Direct Memory Access Mode
$8E	PD_ScsiID	SCSI Controller ID
$8F	PD_ScsiLUN	LUN on SCSI controller
$90	PD_ScsiOpts	SCSI Options Flags

* References to these flags are often made using the PD_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD_SBFFlags. References to PD_DrivFlag should use a value of PD_Flags + 1.

NOTE: ***Offset*** refers to the location of a path descriptor field relative to the starting address of the path descriptor. Path descriptor offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l or usr.l.

SBF Device Drivers

SBF device drivers are designed to support any sequential storage device which reads and writes data in fixed or variable size blocks (tapes).

Because SBF is intended for sequentially accessed files, it does not support a directory structure or provide a byte-oriented file positioning mechanism. Consequently, I$Makdir, I$ChgDir, I$Delete, and I$Seek return the error E$UnkSvc.

Read and write calls to the driver are made by SBF in terms of logical blocks. The logical block size is specified in the PD_BlkSiz field of the path descriptor. The driver is responsible for translating the block request into the appropriate number of physical media blocks. If a "partial" physical block results from this translation, drivers must either buffer the partial block or return an error.

GetStat calls are passed straight to the driver, with the exception of SS_EOF and SS_Ready, which are handled by SBF. Typical drivers ignore all GetStat calls and return an unknown service request error (E$UnkSvc).

SetStat calls are passed straight to the driver, with the exception of SS_Opt. SBF determines if the buffer size has changed, and if so, flushes any pending buffers to tape prior to calling the driver. SetStat calls to the driver are used for control and positioning operations (for example, write filemark, rewind tape) on the media. These calls can originate from the user or from SBF internal operations (for example, write filemark when file closed).

Sensing the End-of-Tape

All tape drives can sense the physical end-of-tape (EOT). Many drives also provide an “early” EOT warning. The type of warning(s) provided by the drive determines whether or not buffered I/O (PD_NumBlk) is usable, as follows:

Early EOT Warning

Drives which provide an early EOT capability notify the driver of the EOT condition prior to reaching the end of the physical tape. The amount of tape between the early EOT mark and physical tape end varies among drive models; however, typical drives allow about 1000 physical blocks to be written after the early EOT warning.

When a driver that is writing blocks encounters the early EOT warning, it should write the blocks to the tape and return a media full error (E$Full). If the device is using buffered I/O, subsequent write calls may still be made by SBF to the driver to flush all currently buffered blocks to the tape. The driver should not refuse these write requests: it should continue to write the data to tape and continue returning E$Full.

The driver should maintain this mode of operation until a “control” operation occurs (for example, write filemark or rewind), at which time the driver can clear its EOT status. This technique of writing all currently buffered blocks to tape ensures that the application knows which blocks are on which tape.

When setting up the device descriptors block size (PD_BlkSiz) and buffer count (PD_NumBlk), you should ensure that there is enough room on the tape after the early EOT mark to accommodate the total amount of data that could be buffered (PD_NumBlk * PD_BlkSiz).

Drives which provide early EOT warning can operate in buffered or unbuffered I/O mode.

Physical EOT Warning

Drives which only provide a physical EOT warning notify the driver when the actual end-of-tape is about to be reached. There is sufficient tape remaining to allow the last write to complete and a filemark to be written. No additional blocks can be written to the tape.

You can only operate physical EOT devices in unbuffered I/O mode, because there is no guarantee that you can write SBF-buffered blocks to tape after the physical EOT is detected. When the driver detects EOT, it should ensure that the last write has completed and return a media full error (E$Full). The next access to the driver is typically a write filemark operation and rewind.

Tape Positioning Operations

SetStat functions are available to allow tape positioning operations. These calls allow the driver to skip forward or backward on the tape, using a specified block or filemark count.

Depending upon the capabilities of the tape drive in use, reverse tape movement may require driver assistance. If the tape drive supports reverse movement, the driver simply hands the count to the drive. If the tape drive only supports forward movement, the driver has to maintain counters for the current filemark and block position on the tape. The driver must use movement commands supported by the tape drive to simulate reverse movement. For example, if the tape's current position is filemark #2, block #20, then a request to move back five blocks would (typically) be simulated by:

¨ Rewind tape
¦ Skip forward two filemarks
Æ Skip forward 15 blocks

When this situation is in effect, drivers maintain these tape position counters in an external module (for example, data module), so that the counters are not erased when the device is attached and detached. The INIT routine attempts to create and link to the module, while the TERM routine unlinks the module.

Some tape motion commands (for example, rewind, skip blocks, retension) may take a long time. When using SCSI tape drives, these types of functions can busy the SCSI bus to other users for excessive lengths of time. To improve this situation, drivers should follow these guidelines:

- If possible, set the "immediate return" flag in the SCSI command packet, to enable the tape drive to return status without waiting for motion to complete.

- If possible, implement disconnect/reconnect, to enable the tape drive to release the bus during long motion functions, allowing other SCSI activity (such as disk accesses) to occur.

Tape Streaming

Tape "streaming" is achieved when the process and driver are able to send/receive data to/from the tape device at a rate that is equal to or faster than the tape drive's data I/O rate. The tape drive can keep the tape in motion continuously, thus achieving the minimum data transfer time. If the data rate falls below this threshold, the tape drive has to perform stop-motion/reverse/start-motion functions whenever it has to wait for the process/driver to issue the next I/O request. This stop/start motion can significantly increase the time it takes for the overall tape operations.

To achieve maximum streaming on tapes, drivers should follow these guidelines:

- Use buffered I/O (PD_NumBlk) on tape drives that support early EOT detection.
- Set the logical block size (PD_BlkSiz) to the size of the tape drive's internal buffer (typical tape drives have an internal buffer to assist streaming).
- If the tape drive supports "immediate returns" on writes, turn this function on. Immediate returns allow the tape drive's controller to indicate "command complete" to the driver when the data is in the controller's internal buffer, but prior to writing the data to physical tape. The controller then begins writing to tape while SBF is preparing for the next write.
- On SCSI-based systems, implement disconnect/reconnect if possible, so that tape operations minimize SCSI bus occupancy. This allows situations such as SCSI-disk to SCSI-tape backups to achieve maximum overlaps of disk/tape activity.

SBF Device Driver Storage Definitions

SBF device driver modules contain a package of subroutines that perform block-oriented I/O to or from a specific hardware controller. Because these modules are re-entrant, one "copy" of the module can simultaneously run several identical I/O controllers.

The kernel allocates a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header (M$Mem). Some of this storage area is required by the kernel and SBF; the device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the sbfdev.a and sbfdrvtb.a DEFS files. Static storage is used as follows:

Offset	Name	Maintained By	Description
$00	V_PORT	Kernel	Device base address
$04	V_LPRC	Kernel	Last active process ID
$06	V_BUSY	File Manager	Active process ID
$08	V_WAKE	Driver	Process ID to awaken
$0A	V_Paths	Kernel	Linked list of open paths
$0E			Reserved
$30	SBF_NDRV	Driver	Number of Drives
$32	SBF_Flag	File Manager	Driver Flags
$34	SBF_Drvr	File Manager	Driver Module Pointer
$38	SBF_DPrc	File Manager	Driver Process Pointer
$3C	SBF_IPrc	Driver	Interrupt Process Pointer
$40			Reserved
$80			Drive Tables Begin

NOTE: ***Offset*** refers to the location of a static storage field relative to the starting address of the static storage. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: sys.l.

Name	Description
V_PORT	**Device port address** Contains the device's physical port address. It is copied from M$Port in the device descriptor when the device is attached by the kernel.
V_LPRC	**Last active process ID** Contains the process ID of the last process to use the device. While this field is required for all static storage by the kernel, it is not used by SBF.

Name	Description
V_BUSY	**Current active process** The process ID of the process currently using the device. It is used to implement I/O Blocking by SBF. This field is also used by the interrupt drivers when they wish to suspend themselves, by copying V_BUSY to V_WAKE (prior to suspending themselves). A value of zero indicates the device is not busy.
V_WAKE	**Process ID to awaken** The process ID of any process that is waiting for the device to complete I/O. A value of zero indicates that no process is waiting. The driver sets V_WAKE from V_BUSY. V_WAKE provides the interlock between the driver and the driver's interrupt service routine.
V_PATHS	**Linked List of Open Paths** A singly-linked list of all paths currently open on this device.
SBF_NDRV	**Number of drives** Contains the number of drives that the controller can use. It is defined by the device driver as the maximum number of logical drives with which the controller can work. SBF assumes that there is a drive table for each drive. SBF validates the tape drive number (PD_TDrv) against this value to ensure that the logical drive number is valid for the driver.
SBF_Flag	**Driver Flags** Contains flags used by SBF to indicate the current state of the path.
SBF_Drvr	**Driver Module Pointer** Contains the pointer to the device driver.
SBF_DPrc	**Driver Process Pointer** Contains the pointer to the process associated with the driver. SBF initializes this when a path is opened to the device. The driver's TERM routine should check this field, and if non-zero, delete the process (F$DelPrc). SBF_IPrc**Interrupt Process Pointer (obsolete)** This field is available for the driver to use when the driver wishes to create its own process (for example, interrupt handler process). **NOTE:** Do not confuse this process with the SBF process created for buffered I/O. (See SBF_DPrc.) **Drive Tables** Contains one table per drive that the controller will handle. SBF assumes there are as many tables as specified in SBF_NDRV.

Device Driver Tables

There must be as many drive tables as were specified in SBF_NDRV. The format of each drive table is given below:

Offset	Name	Maintained By	Description
$00	SBF_DFlg	File Manager	Drive Flag
$02	SBF_NBuf	File Manager	Buffer Count
$04	SBF_IBH	File Manager	Pointer to Head of Input Buffer List
$08	SBF_IBT	File Manager	Pointer to Tail of Input Buffer List
$0C	SBF_OBH	File Manager	Pointer to Head of Output Buffer List
$10	SBF_OBT	File Manager	Pointer to Tail of Output Buffer List
$14	SBF_Wait	File Manager	Pointer to Waiting Process
$18	SBF_SErr	Driver	Number of Recoverable Errors
$1C	SBF_HErr	Driver	Number of Non-Recoverable Errors
$20			Reserved

Name	Description

SBF_DFlg **Drive Flag**

The high byte of this field contains the current status of the logical drive. The flags are maintained by SBF, and are defined as follows:

bit 1: Set if write mode.

bit 2: Set if driver servicing this drive.

bit 3: Set if EOF (end of file).

All other bits and the low byte bits are reserved.

SBF_NBuf **Buffer Count**

Contains the number of buffers currently allocated to the drive.

SBF_IBH **Pointer to Head of Input Buffer List**
SBF_IBT **Pointer to Tail of Input Buffer List**

These fields contain the head and tail pointers, respectively, of the buffers being returned to SBF by the driver.

SBF_OBH **Pointer to Head of Output Buffer List**
SBF_OBT **Pointer to Tail of Output Buffer List**

These fields contain the head and tail pointers, respectively, of the buffers being sent to the driver by SBF.

Name	Description
SBF_Wait	**User process' process descriptor pointer** This pointer is set when the user process is suspended, waiting for driver I/O to complete.
SBF_SErr	**Number of Recoverable Errors** This field allows the driver to keep a count of "soft" errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
SBF_HErr	**Number of Non-Recoverable Errors** This field allows the driver to keep a count of "hard" errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.

Linking SBF Drivers

After a SBF driver has been assembled into its relocatable object file (ROF), the driver needs to be linked to produce the final driver module. Linking resolves all code references in drivers that are comprised of several ROF files. It also resolves the external data and static storage references by the driver.

The most important part of linking is to correctly resolve the static storage references. Generally, the static storage area is composed of three sections in this order (see Figure 4-1):

¨ I/O globals

¦ Drive tables (one per logical drive)

Æ Driver-declared variables

The driver-declared variables are declared in vsect areas of the driver, but they *must* be allocated after the drive table storage areas. The method that must be used to allocate all of the storage, *in the correct order*, is to link the sbfstat.r library file, ‘n’ instances of sbfdrvtb.r, and then the driver vsect. The sbfstat.r and sbfdrvtb.r files are located in the system’s LIB directory.

The following examples show how a driver should be linked. The first link line creates a driver that supports one logical drive, as only one drive table vsect is allocated:

l68 /dd/LIB/sbfstat.r /dd/LIB/sbfdrvtb.r RELS/sbviper.r -O=OBJS/sbviper

The second link line creates a driver that supports two logical drives, as two drive table vsects are allocated:

l68 /dd/LIB/sbfstat.r /dd/LIB/sbfdrvtb.r /dd/LIB/sbfdrvtb.r RELS/sbtape.r
-O=OBJS/sbtape

NOTE: Failure to link the I/O system globals and the correct number of drive tables, and in the correct order, results in erratic driver operation.

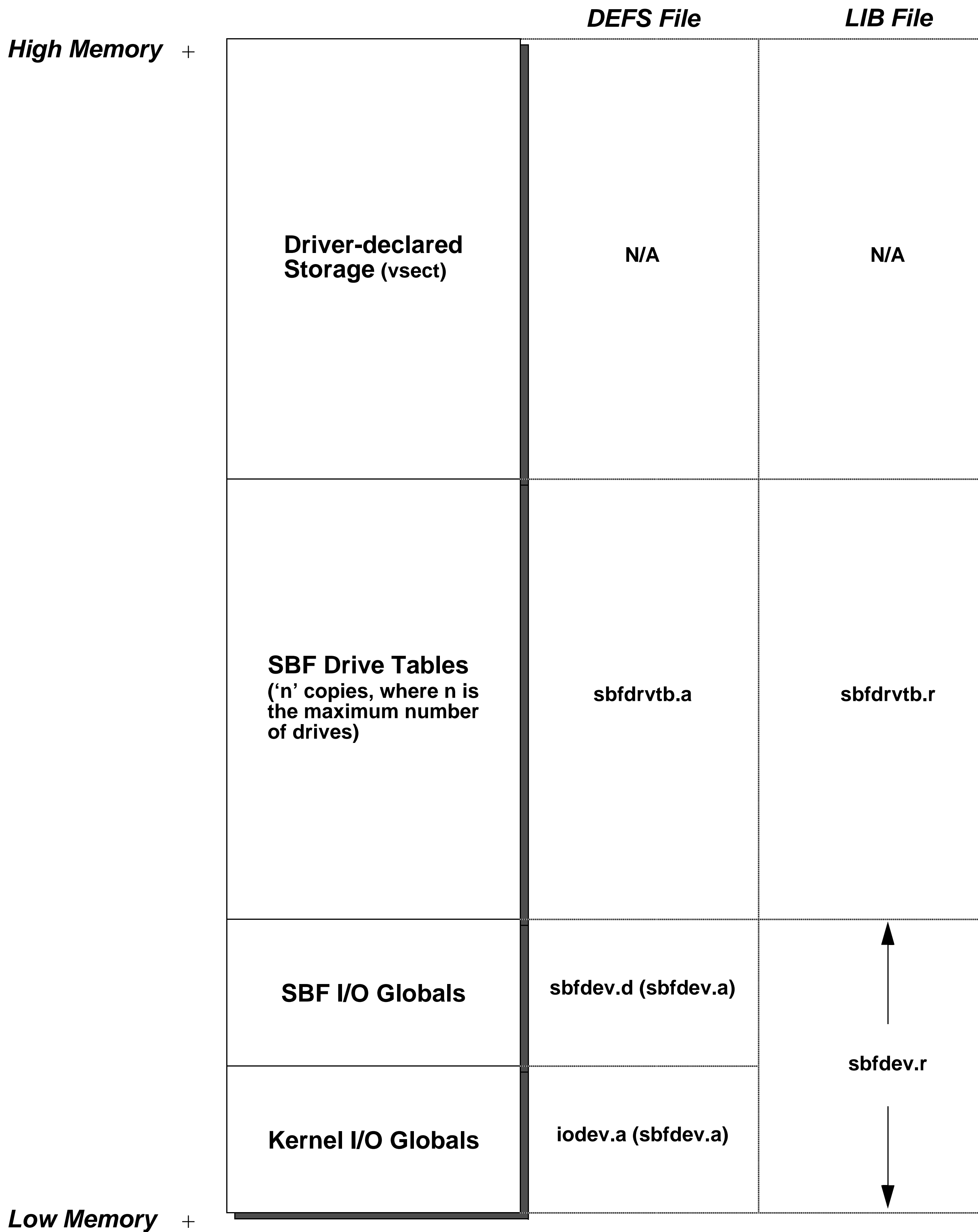


Figure 4-1: SBF Static Storage Layout

SBF Device Driver Subroutines

As with all device drivers, SBF device drivers use a standard executable memory module format with a module type of Drivr (code $E0). SBF drivers are called in system state.

NOTE: I/O system modules must have the following module attributes:

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

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of the corresponding subroutine. The branch table appears as follows:

```
ENTRY   dc.w   INIT      initialize device
        dc.w   READ      read character
        dc.w   WRITE     write character
        dc.w   GETSTAT   get device status
        dc.w   SETSTAT   set device status
        dc.w   TERM      terminate device
        dc.w   TRAP      handle illegal exception (0 = none)
```

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, the carry bit should be set and an appropriate error code returned in the least significant word of register d1.w.

The TRAP entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

The following pages describe each subroutine.

INIT — Initialize Device and its Static Storage

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The INIT routine must:

¨ Initialize the device's permanent storage. Minimally, this consists of initializing SBF_NDRV to the number of drives with which the controller will work.

If the driver maintains flags/variables that must "span" detach/attach sequences (for example, for reverse movement simulation), then the INIT routine should create/link to an external module (for example, a data module). The module pointer should then be saved. If the module was created, its storage area should then be initialized.

¦ Place the IRQ service routine on the IRQ polling list by using the F$IRQ service request, if required.

Æ Initialize device control registers (enable interrupts if necessary).

Prior to being called, the device permanent storage is cleared (set to zero) except for V_PORT which will contain the device address.

If INIT returns an error, it does not have to clean up its operation (for example, remove device from polling table or disable hardware). The kernel calls TERM to allow the driver to clean up INIT's operation before returning to the calling process.

NOTE: If the INIT routine causes an interrupt to occur, handle the interrupt in one of two ways:

¨ Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, then restoring the previous interrupt level. This is the preferred technique unless the interrupt is time-consuming.

- Allow the interrupt service routine to service the hardware. In this case, the process descriptor contains the process ID (P$ID) to which V_WAKE should be set. You cannot use V_BUSY because it is zero when INIT is called.

READ — Read Block(s)

INPUT:
d0.l = buffer size
(a0) = address of buffer
(a2) = address of device static storage
(a3) = drive table
(a4) = process descriptor pointer
(a6) = system global data storage pointer

OUTPUT:
d1.l = block size read

ERROR OUTPUT:
cc = carry bit set
d1.w = error code

FUNCTION: The READ routine must:

¨ Initialize the drive, if required.

¦ Convert the requested byte-count into the block-count for the media. If the requested count does not specify an integral number of media blocks, the driver should return an error (typical case) or take steps to buffer the partial block.

Æ Issue the READ command to the device and wait for I/O to complete (using interrupts if possible).

Ø When the I/O operation is complete, check the status of the READ. If a fatal error occurred, return it to SBF.

× If no error, or a non-fatal error occurred, check the amount of data actually read and return that count to SBF.

Most tape devices terminate a READ request when a filemark is encountered. The tape device returns the data from the current position up to the filemark. Thus, the byte-count returned may be less than the requested amount. This is a typical non-fatal error on tape devices.

WRITE — Write Block(s)

INPUT:
d0.l = buffer size
(a0) = address of buffer
(a2) = address of the device static storage area
(a3) = drive table
(a4) = process descriptor pointer
(a6) = system global data storage pointer

OUTPUT: **The buffer is written to tape.**

ERROR OUTPUT:
cc = carry bit set
d1.w = error code

FUNCTION: The WRITE routine must:

¨ Initialize the drive, if required.

¦ Convert the requested byte-count into the block-count for the media. If the requested count does not specify an integral number of media blocks, then the driver should return an error (typical case) or take steps to buffer the partial block.

Æ Issue the WRITE command to the device and wait for I/O to complete (using interrupts if possible).

Ø When the I/O operation has completed, check the status of the WRITE. If a fatal error occurred, return it to SBF.

× If no error, or a non-fatal error occurred, check the amount of data actually written.

Many tape devices terminate a write request when an early end-of-tape (EOT) is detected. For these types of devices, the data can still be written to tape because the EOT state is a warning that there is a small amount of tape remaining. The driver should ensure that this write is fully completed, and return a media full error (E$Full).

Subsequent write calls should not be refused at this point, as SBF may need to flush its current buffers (if in buffered I/O mode) to the tape. The application is notified of the media full condition on its next write, so that it may close the file. When the file closes, SBF issues appropriate SetStats (for example, write filemark) to finalize tape operation.

If the tape device is one which only detects a physical EOT condition, then the driver should only be operated in unbuffered I/O mode. In this case, the driver should ensure that the write invoking the physical EOT condition is written to tape and a media full error (E$Full) returned to SBF. No further writes should be presented to the driver, as the application is notified immediately of the media full condition. The application can then close the path, allowing SBF to write the final filemark and finalize tape operation.

GETSTAT/SETSTAT — Get/Set Device Status

INPUT:	**d0.w = status code** **d2.l = argument count** **(a1) = address of the path descriptor** **(a2) = address of the device static storage area** **(a3) = drive table** **(a4) = process descriptor pointer** **(a6) = system global data storage pointer**
OUTPUT:	**Depends on the function code**
ERROR OUTPUT:	**cc = carry bit set** **d1.w = error code**

FUNCTION: These routines are wild-card calls used to get/set the device's operating parameters as specified for the I$GetStt and I$SetStt service requests.

Calls which involve parameter passing require the driver to examine or change the register stack variables. These variables contain the contents of the MPU registers at the time the I$GetStt/I$SetStt request was made. Parameters passed to the driver are set up by the caller prior to using the service call. Parameters passed back to the caller are available when the service call completes. The register stack image pointer is stored in the path descriptor (PD_RGS).

Typical SBF drivers have routines to handle the following I$SetStt codes:

SS_Feed	Erase tape
SS_Opt	Write path options section
SS_Reset	Rewind tape
SS_Reten	Retension tape
SS_RFM	Skip past tape mark(s)
SS_Skip	Skip block(s)
SS_SQD	Place drive off-line
SS_WFM	Write tape mark(s)

Usually all I$GetStt codes and other I$SetStt codes return with an unknown service request error (E$UnkSvc).

The following pages describe the driver's role in the implementation of the above I$SetStt calls.

SS_Feed

This call erases all or part of the tape. The number of blocks to be erased is passed in register d2. If the count is -1, the entire tape is to be erased from the current position to end-of-tape (EOT), otherwise, the specified count of blocks should be written, starting at the current tape position.

The erase routine should:

¨ Initialize the drive, if required.

¦ Issue the appropriate command to achieve the desired erase function. Many tape devices support a direct “erase” command. If the tape device does not support this feature, the driver should perform “writes” to simulate the desired effect. Once the command is issued, the driver should wait for I/O to complete (with interrupts if possible).

Æ Check the status of the I/O command and return any error to SBF.

Ø If the driver maintains flags pertaining to current tape position, these should be updated.

× Return status to SBF.

SS_Opt

This routine is called when the path descriptor options are changed by the user. Typically, the driver ignores this call.

SS_Reset

This call rewinds the tape to beginning-of-tape (BOT). The rewind routine should:

¨ Initialize the drive, if required.

¦ Issue the appropriate command to the device and wait for I/O to complete (with interrupts, if possible).

Æ Check the status of the I/O command and return any error to SBF.

Ø If the driver maintains internal flags pertaining to current tape position, they should be reset. Typical flags would be end-of-file and end-of-tape. For drivers that count current filemark/block positions, these counters should also be cleared.

× Return status to SBF.

SS_Reten

This call performs a retension pass on the tape. Typically, the tape moves to BOT, moves to EOT, then rewinds to BOT. The sequence of actions for SS_Reten is the same as that for SS_Reset.

Retensioning tape media is highly recommended for new media, shipped media, or any media that has been stored for a long period.

SS_RFM

This routine is called when the tape position is to be moved forward or backwards by the specified number of filemarks. (This number is passed in register d2.) If the tape device is incapable of directly skipping backward, the driver has to simulate the reverse movement using rewind and skip forward commands. The sequence of actions for SS_RFM is the same as that for SS_SQD.

SS_Skip

This routine is called when the tape position is to be moved forward or backward the specified number of tape blocks. The number of blocks to skip is passed as a logical block count (PD_BlkSz) in register d2. The driver must translate this count into the media's physical block count. If the tape is incapable of directly skipping backward, it has to simulate the reverse movement using rewind and skip forward commands.

The sequence of actions for SS_Skip is the same as that for SS_SQD.

SS_SQD

This routine is called to unload the tape (put the tape device off-line). Depending upon the capabilities of the tape device, this action may turn off the drive-select LED, or unload and eject the media.

The unload routine should:

¨ Initialize the drive, if required.

¦ Issue the appropriate command to the device and wait for I/O to complete (with interrupts, if possible).

Æ Check the status of the I/O command and return any error to SBF.

Ø If the driver maintains flags pertaining to current tape position, these should be updated.

× Return status to SBF.

SS_WFM

This routine is called to write the specified number of filemarks to the tape. (This number is passed in register d2.) Applications may place filemarks on the tape as they see fit. The sequence of actions for SS_WFM is the same as that for SS_SQD.

TERM — Terminate Device

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage area
(a4) = process descriptor pointer
(a6) = system global static storage

OUTPUT: **None**

ERROR OUTPUT: **cc = carry set**
dl.w = error code

FUNCTION: This routine is called when a device is no longer in use in the system (see I$Detach).

The TERM routine must:

¨ Wait until any pending I/O has completed.

¦ Disable the device interrupts.

Æ Remove the device from the IRQ polling list.

Ø Kill the driver process created by SBF. If SBF_DPrc is non-zero, this is a pointer to the driver's process descriptor. This process is returned by making a F$DelPrc system call with the process ID from P$ID.

× If the driver maintains flags/variables that must "span" detach/attach sequence, then the TERM routine should unlink any external modules linked to during INIT.

NOTE: If an error occurs during the device's INIT routine, the kernel calls the TERM routine to allow the driver to clean up. If the TERM routine uses static storage variables (for example, interrupt mask values, dynamic buffer pointers), it should validate these variables prior to using them. The INIT routine may not have set up all the variables prior to exiting with the error.

IRQ Service Routine — Service Device Interrupts

INPUT: **(a2) = static storage address**
(a3) = port address
(a6) = system global static storage

OUTPUT: **None**

ERROR OUTPUT: **cc = carry set (interrupt not serviced)**

FUNCTION: This routine is called directly by the kernel's IRQ polling table routines. Its function is to:

¨ Check the device for a valid interrupt. If the device does not have an interrupt pending, the carry bit must be set and the routine exited with an RTS instruction as quickly as possible. Setting the carry bit signals the kernel that the next device on the vector should have its IRQ service routine called.

¦ Service device interrupts.

Æ Wake up the driver mainline, using the synchronization method of the driver:

Signals: Send a wake-up signal to the process whose process ID is in V_WAKE, when the I/O is complete. Also, clear V_WAKE as a flag to the mainline program that the IRQ has occurred.

Events: Signal the event that the IRQ has occurred, using the event system's signal function.

Ø Clear the carry bit and exit with an RTS instruction after servicing an interrupt.

Avoid exception conditions (for example, a Bus Error) when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

NOTE: IRQ service routines may destroy the contents of following registers only: d0, d1, a0, a2, a3, and a6. The contents of all other registers must be preserved or unpredictable system errors (system crashes) will occur.

End of Chapter 4

NOTES