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.

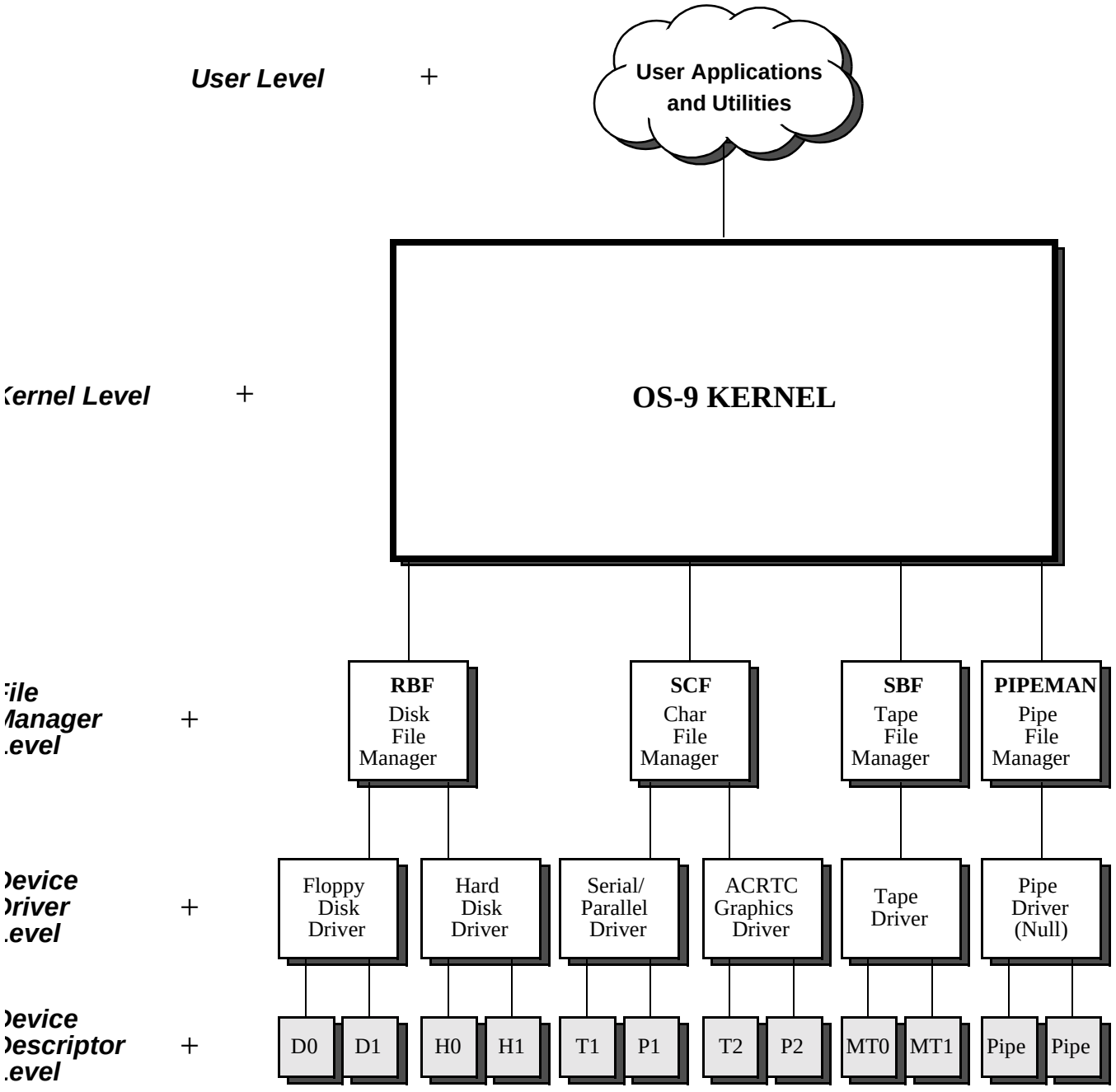


Figure 1-1: OS-9 I/O System Module Organization

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.

- 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.
- 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).

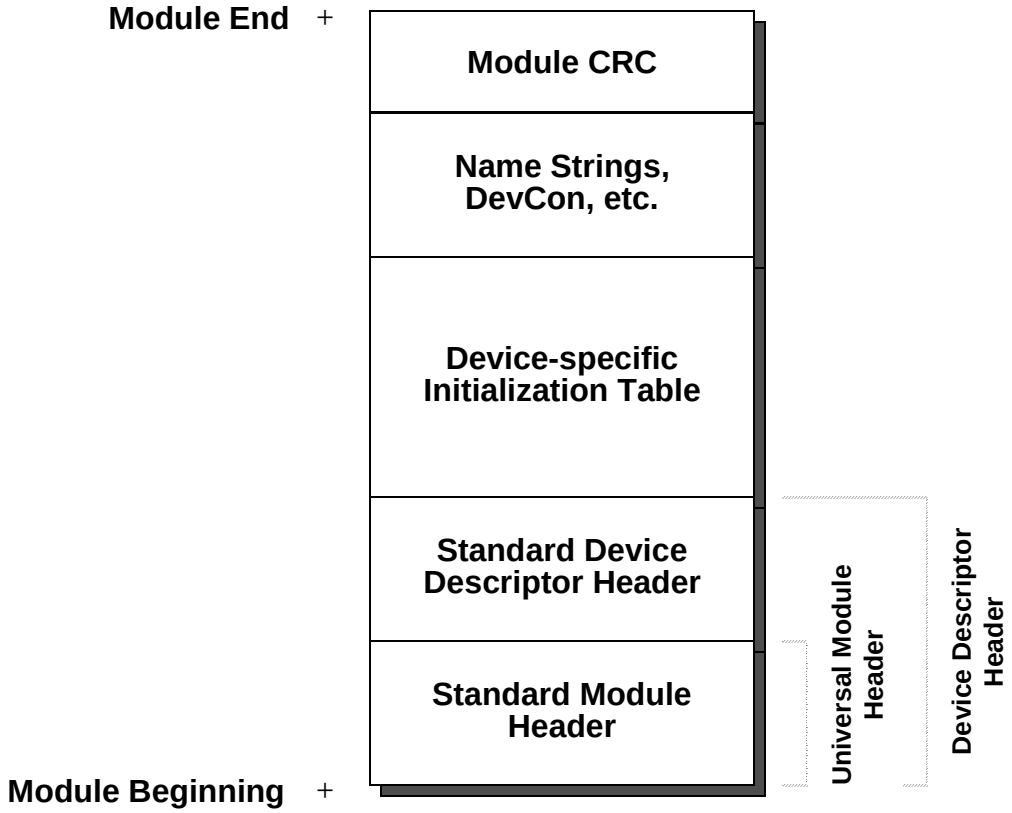


Figure 1-2: Device Descriptor Layout

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.

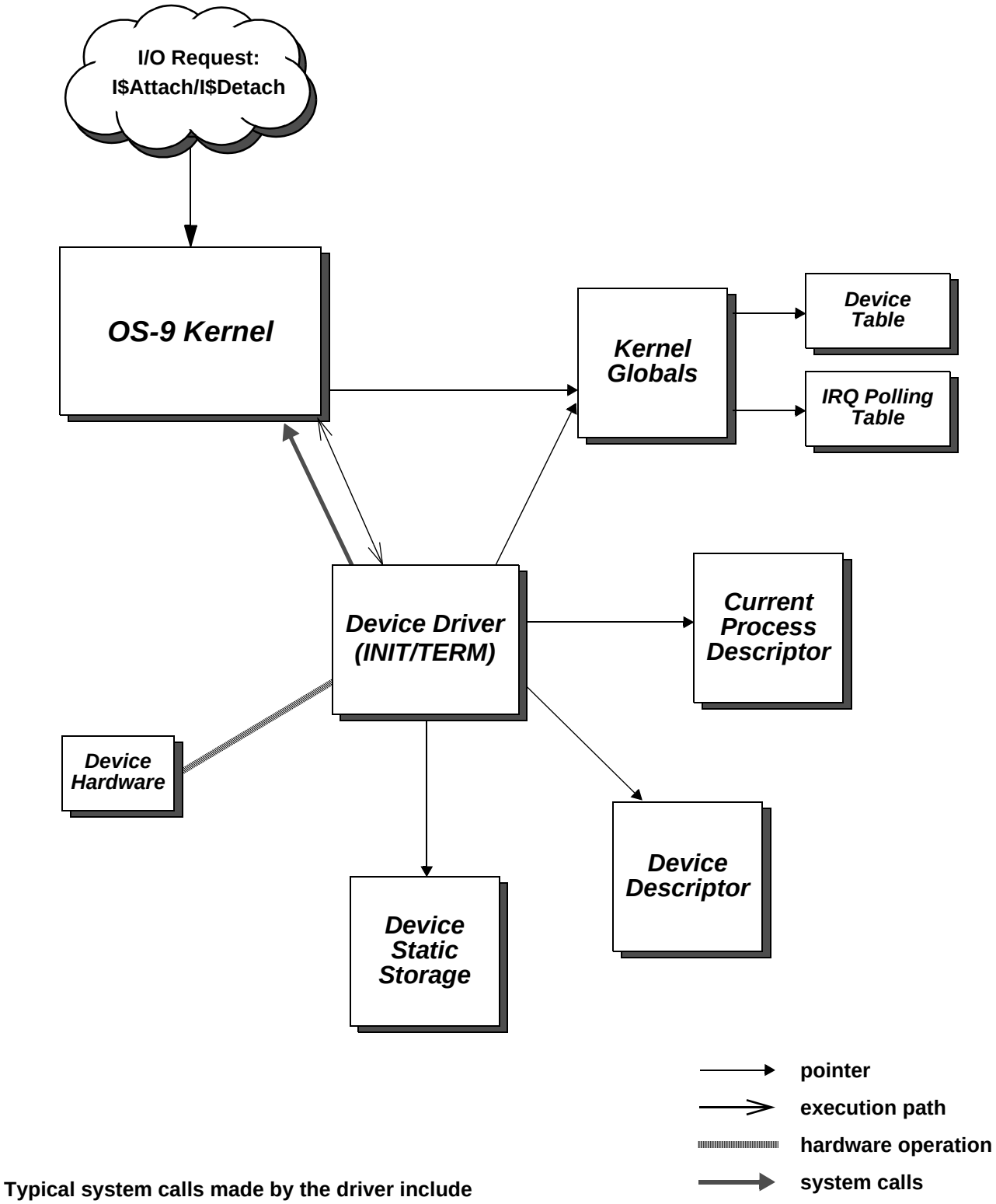
The IRQ subroutine is not called by the file manager, but by the kernel's interrupt polling routine. It communicates with the driver's main section through the static storage and certain system calls.

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

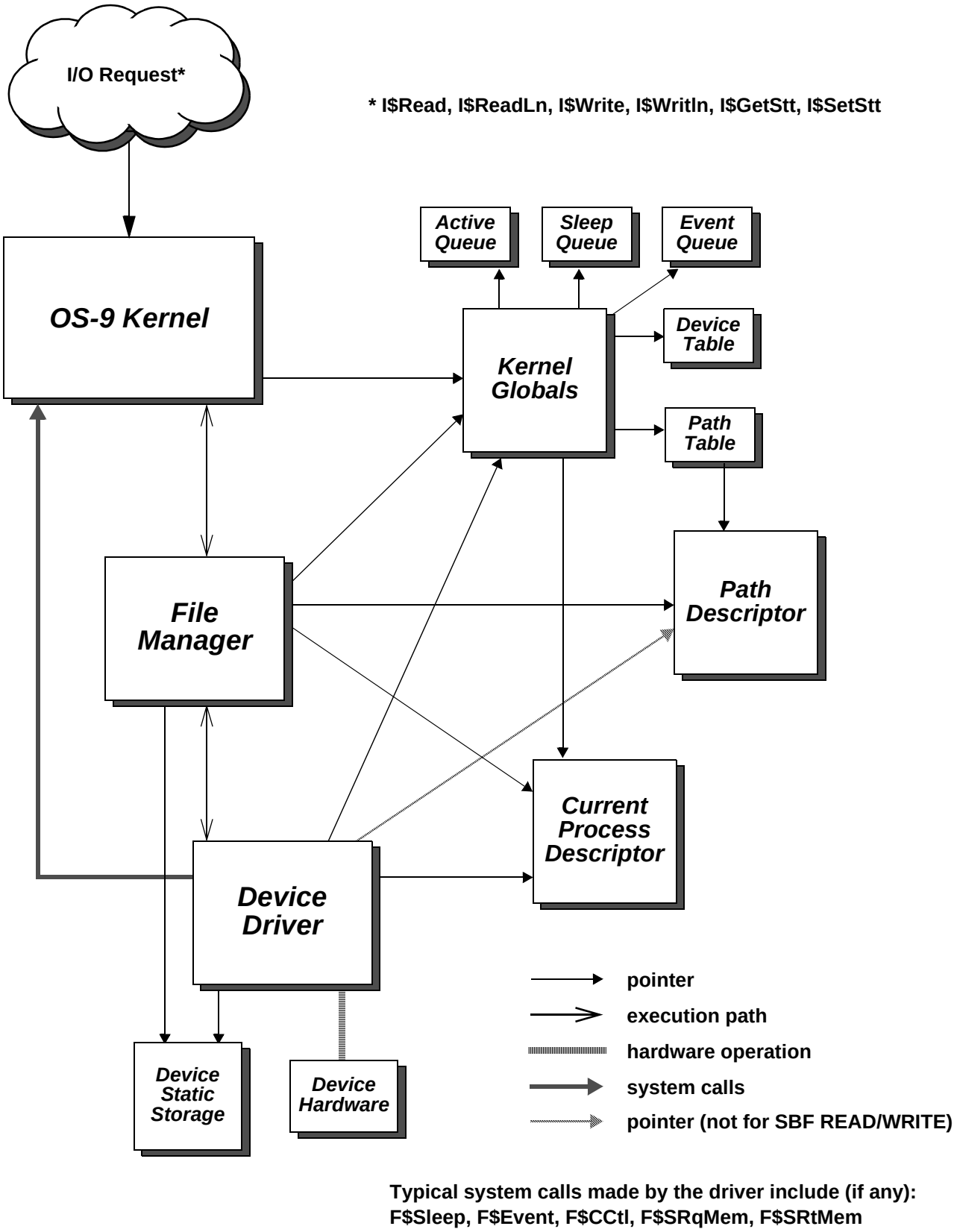


Figure 1-4: I/O System Layout for READ/WRITE/GETSTAT/SETSTAT Routines

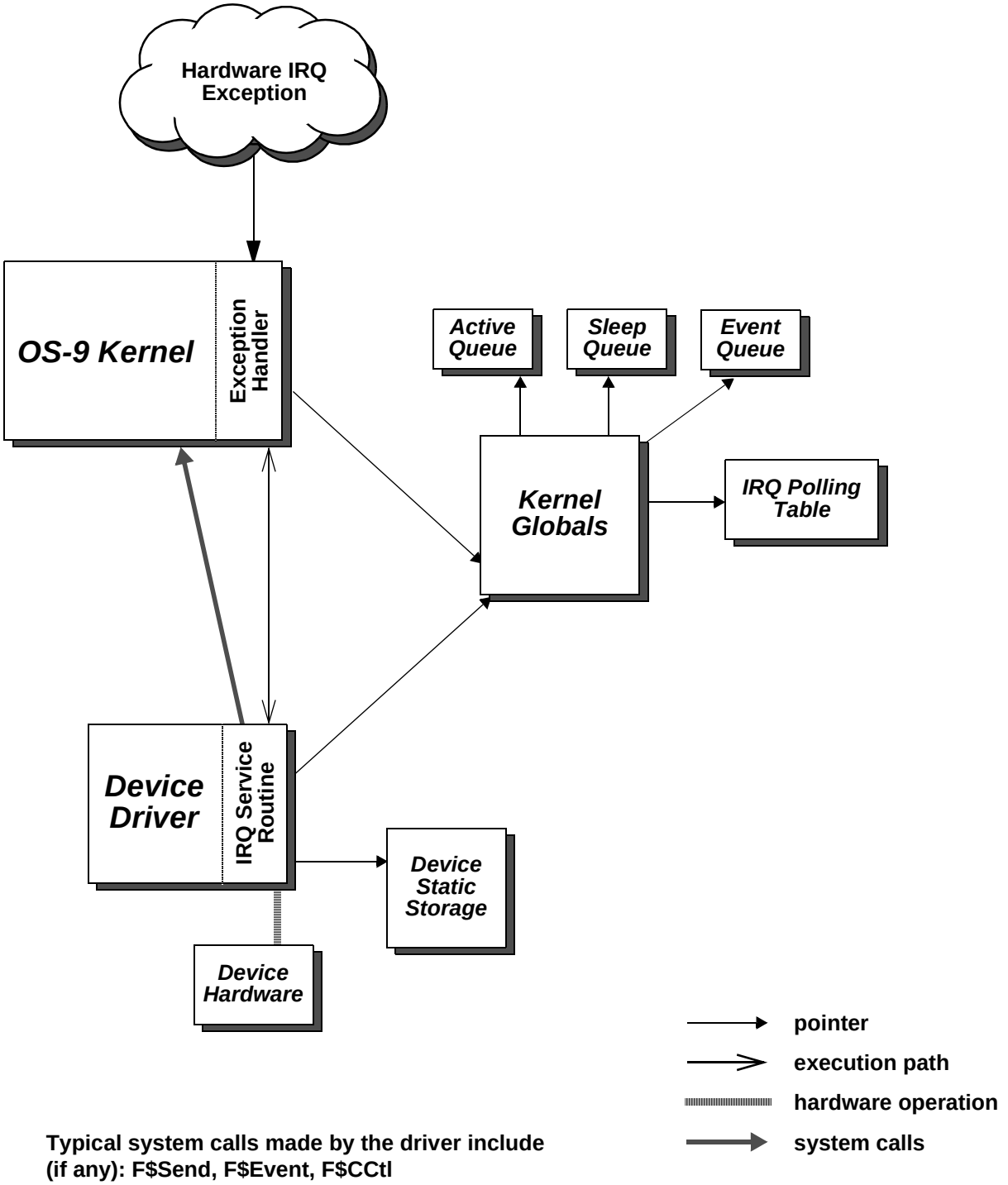


Figure 1-5: I/O System Layout for IRQ Service Routine

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.

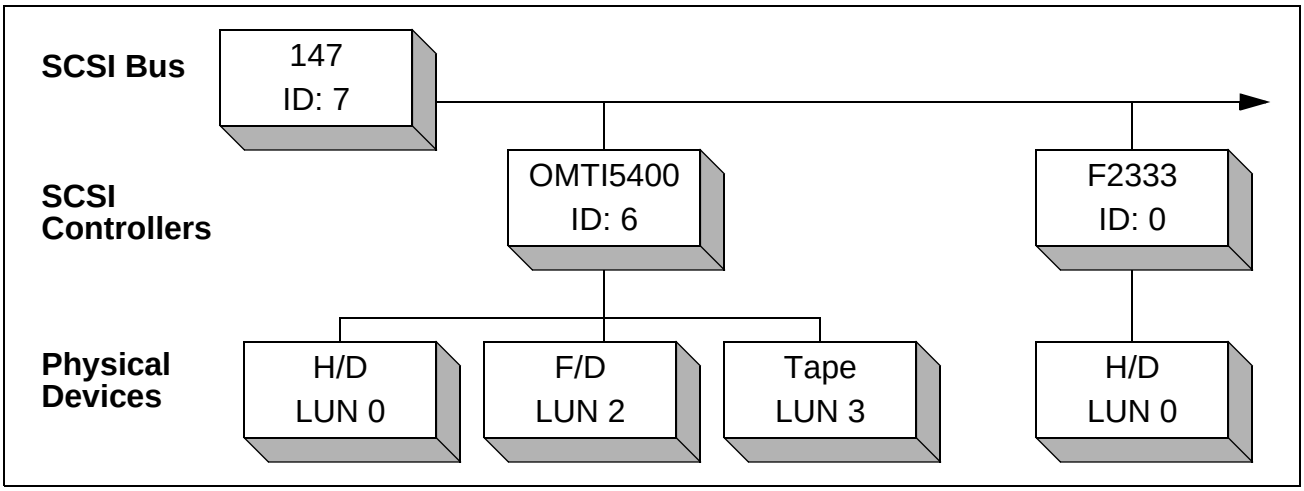
Fujitsu 2333 Hard Disk with Embedded SCSI Controller:

- 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:

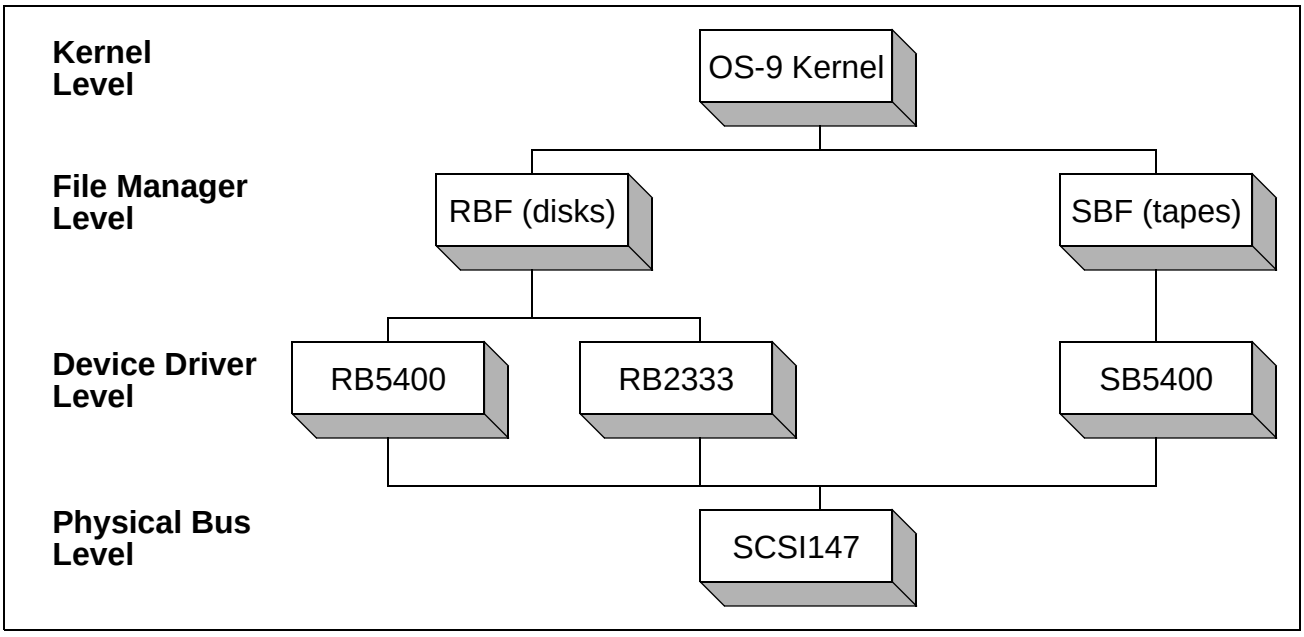
The high-level drivers associated with this configuration are:

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:

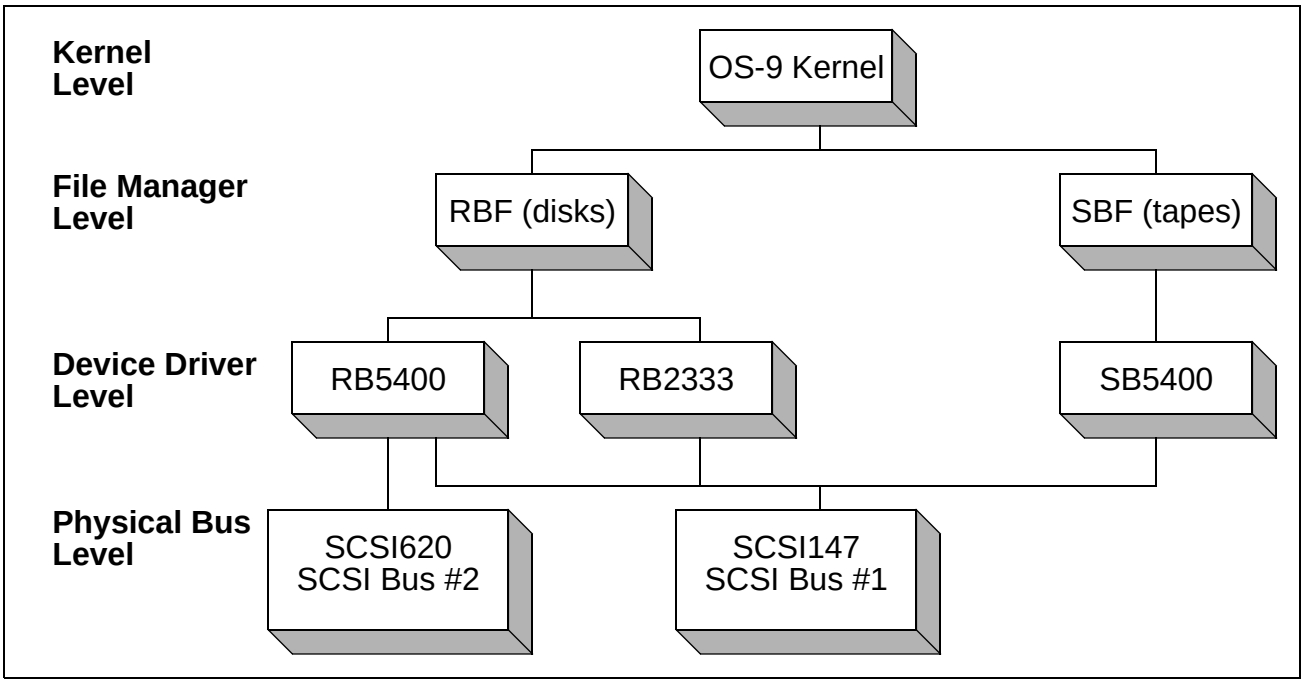
Example One

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:

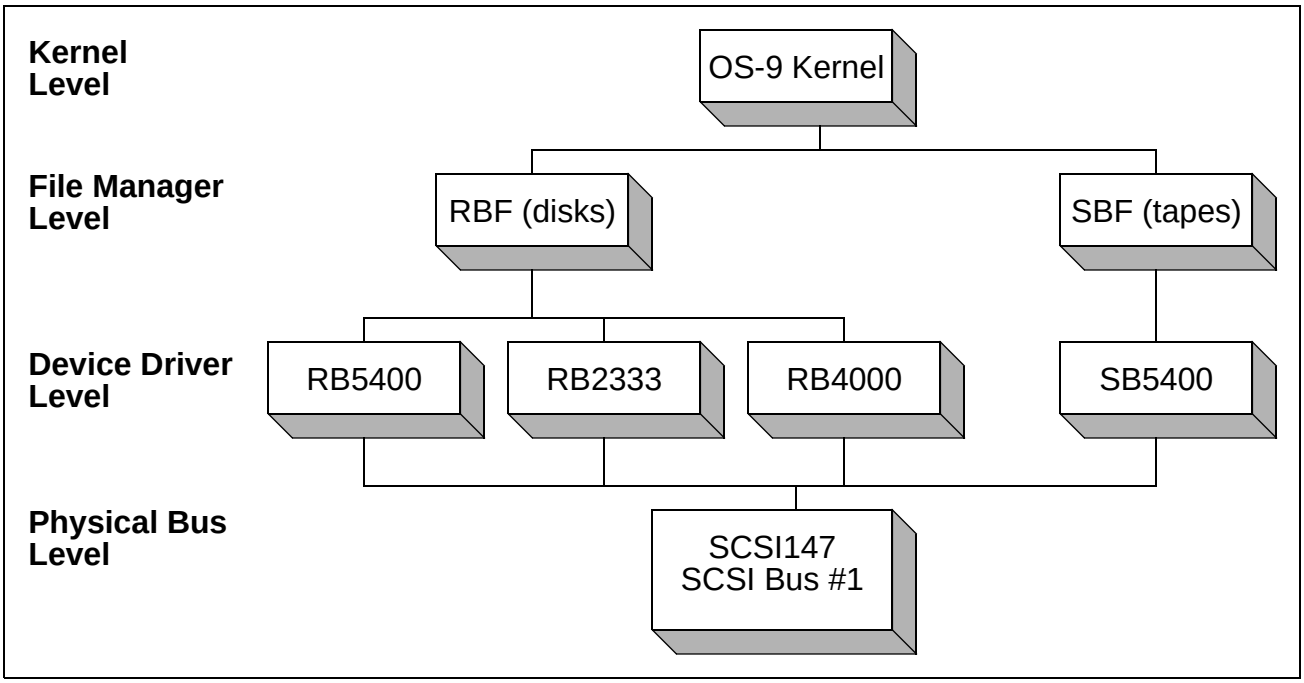


Example Two

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 boot-strap ROM.

End of Chapter 1