# 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 particular 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`.

## The Kernel and I/O

The kernel provides the first level of service for I/O system calls by routing data between processes and the appropriate file managers and device drivers. The kernel also allocates and initializes global static storage on behalf of file managers and device drivers.

The kernel maintains two important internal data structures: the device table and the path table. These tables reflect two other structures respectively: the device descriptor and the path descriptor.

When a path is opened, the kernel attempts to link to the device descriptor associated with the device name specified (or implied) in the pathlist. The device descriptor contains the names of the device driver and file manager for the device. The information in the device descriptor is saved by the kernel in the device table so that it can route subsequent system calls to these modules.

Paths maintain the status of I/O operations to devices and files. The kernel maintains these paths using the path table. Each time a path is opened, a path descriptor is created and an entry is added to the path table. When a path is closed, the path descriptor is de-allocated and its entry is deleted from the path table.

When an I$Attach system call is first performed on a new device descriptor, the kernel creates a new entry in the device table. Each entry in the table has specific information from the device descriptor concerning the appropriate file manager and driver. It also contains a pointer to the device driver static storage. For each device in the table, the kernel maintains a use count which indicates the current number of device users.

## *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 name of the module is used as the logical device name by the system and user (that is, it is the device name given in pathlists). Its format consists of a standard module header that has a type code of device descriptor (DEVIC). The remaining module header fields are shown in Figure 3-1 and described below.

**NOTE:** These fields are standard for all device descriptor modules. They are followed by a device specific initialization table. Refer to Appendix B of this manual for the initialization tables of each standard class of I/O devices (RBF, SCF, SBF).

| **Name** | **Description** |
|---|---|
| M$Port | **Port address**<br>The absolute physical address of the hardware controller. |
| M$Vector | **Interrupt vector number**<br>The interrupt vector associated with the port, used to initialize hardware and for installation on the IRQ poll table:<br><br>25-31 for an auto-vectored interrupt. Levels 1 - 7.<br>57-63 for 68070 on-chip auto-vectored interrupts. Levels 1 - 7.<br>64-255 for a vectored interrupt. |

| **Name** | **Description** |
|---|---|
| M$IRQLvl | **Interrupt level**<br>The device's physical interrupt level. It is *not* used by the kernel or file manager. The device driver may use it to mask off interrupts for the device when critical hardware manipulation occurs. |
| M$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$Mode | **Device mode capabilities**<br>This byte is used to validate a caller's access mode byte in I$Create or I$Open calls. If the bit is set, the device is capable of performing the corresponding function. If the Share_bit (single user bit) is set here, the device is non-sharable. This is useful for printers. |
| M$FMgr | **File manager name offset**<br>The offset to the name string of the file manager module for this device. |
| M$PDev | **Device driver name offset**<br>The offset to the name string of the device driver module for this device. |
| M$DevCon | **Device configuration**<br>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$DevCon is generally available to the driver only during the driver's INIT and TERM routines. Other routines in the driver (for example, Read) must first search the device table to locate the device descriptor before they can access this field.<br><br>Typically, this table is used for name string pointers, OEM global allocation pointers, or device-specific constants/flags. **NOTE:** These values, unlike the standard options, are not copied into the path descriptors options section. |
| M$Opt | **Table size**<br>This contains the size of the device's standard initialization table. Each file manager defines a ceiling on M$Opt. |

| **Name** | **Description** |
|---|---|
| M$DTyp | **Device type (first field of initialization table)**<br>The device's standard initialization table is defined by the file manager associated with the device, with the exception of the first byte (M$DTyp). The first byte indicates the class of the device (RBF, SCF, etc.). |

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.

**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.

| Offset | Name | Description |
|---|---|---|
| $30 | M$Port | Port Address |
| $34 | M$Vector | Trap Vector Number |
| $35 | M$IRQLvl | IRQ Interrupt Level |
| $36 | M$Prior | IRQ 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 |

***Figure 3-1: Additional Standard Header Fields For Device Descriptors***

You may wish to add additional devices to your system. If an identical device controller already exists, simply add the new hardware and another device descriptor. Device descriptors can be in ROM, in the boot file, or loaded into RAM from mass storage files while the system is running.

## *Path Descriptors*

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

Path descriptors have three sections:

- The first 30 bytes are defined universally for all file managers and device drivers.
- PD_FST is reserved for and defined by each type of file manager for file pointers, permanent variables, etc.
- PD_OPT is a 128-byte option area used for dynamically alterable operating parameters for the file or device. These variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module, and can be examined or altered later by user programs via GetStat and SetStat system calls. Not all options can be modified.

Refer to Appendix B for the current definitions of the path descriptor option area for each standard class of I/O devices (that is, RBF, SCF, SBF, and Pipes). The definitions are included in sys.l or usr.l, and are linked into programs that need them.

**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 the module with the relocatable libraries, sys.l, or usr.l.

| 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 |

***Figure 3-2: Universal Path Descriptor Definitions***

## File Managers

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

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

File managers are re-entrant. One file manager may be used for an entire class of devices having 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 included in a typical 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 hardcopy 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. These routines are called in system state. A sample listing of the beginning of a file manager module is listed in Figure 3-3.

When the individual file manager routines are called, 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. |

```
* 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
```

*Figure 3-3: Beginning Of A Sample File Manager Module*

## File Manager I/O Responsibilities

| Name | Description |
|---|---|
| Open | Opens a file on a particular device. This typically involves allocating any buffers required, initializing path descriptor variables, and parsing the path name. If the file manager controls multi-file devices (RBF, PIPEMAN), directory searching is performed to find the specified file. |
| Create | Performs the same function as Open. If the file manager controls multi-file devices (RBF, PIPEMAN), a new file is created. |
| Makdir | Creates a directory file on multi-file devices. Makdir is neither preceded by a Create nor followed by a Close. File managers that are incapable of supporting directories return with the carry bit set and an appropriate error code in register d1.w. |
| Chgdir | On multi-file devices, ChgDir searches for a directory file. If the directory is found, the address of the directory is saved in the caller's process descriptor at P$DIO. The kernel allocates a path descriptor so that the ChgDir function may save information about the directory file for later searching.<br><br>Open and 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 register d1.w. |
| Delete | Multi-file device managers usually do a directory search that is similar to Open and, once found, remove the file name from the directory. Any media that was in use by the file is returned to unused status.<br><br>File managers that do not support multi-file devices return an E_UNKSVC error. |
| Seek | File managers that support random access devices use Seek to position file pointers of the already open path to the specified byte. Typically, this is a logical movement and does not affect the physical device. No error is produced at the time of the Seek, if the position is beyond the current end of file.<br><br>File managers that do not support random access usually do nothing, but do not return an E_UNKSVC error. |

| Name | Description |
|---|---|
| Read | Read returns the number of bytes requested to the user's data buffer. If there is no data available, an EOF error is returned. Read must be capable of copying pure binary data. It generally does not perform editing on data. Usually, the file manager calls the device driver to actually read the data into a buffer. It then copies data from the buffer into the user's data area. This method helps keep file managers device independent. |
| Write | The Write request, like Read, must be capable of recording pure binary data without alteration. Usually, the Read and Write routines are nearly identical. The most notable difference is that 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.<br><br>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. |
| Readln | ReadLn differs from Read in two respects. First, ReadLn is expected to terminate when the first end-of-line character (carriage return) is encountered. Second, ReadLn performs any input editing that is appropriate for the device.<br><br>Specifically, the SCF File Manager performs editing that involves handling backspace, line deletion, echo, etc. |
| Writeln | Writeln is the counterpart of Readln. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing also is performed. After a carriage return, for example, SCF usually outputs a line feed character and nulls (if appropriate). |
| Getstat | The Getstat (Get Status) system call is a wild card call designed to provide the status of various features of a device (or file manager) that are not generally device independent.<br><br>The file manager may perform some specific function such as obtaining the size of a file. Status calls that are unknown by the file manager are passed to the driver to provide a further means of device independence. |

| Name | Description |
|---|---|
| Setstat | Setstat (Set Status) is the same as the Getstat function except that it is generally used to set the status of various features of a device (or file manager).<br><br>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 by the file manager are passed to the driver to provide a further means of device independence. For example, a SetStat call to format a disk track may behave differently on different types of disk controllers. |

Close

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. It may do specific end-of-file processing if necessary, such as writing end-of-file records on tapes.

## *Device Driver Modules*

Device driver modules perform basic low-level physical input/output functions. For example, a disk driver module's basic functions are 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. Because device driver modules are re-entrant, one copy of the module can simultaneously support multiple devices that use identical I/O controller hardware.

This section describes the function and general design of OS-9 device driver modules to aid programmers in modifying existing drivers or writing new ones. To present this information in an understandable manner, only basic drivers for character-oriented (SCF-type) and disk-oriented (RBF-type) devices are discussed. We recommend that you study this section in conjunction with the individual device-specific sections and sample device driver source listings included in the ***OS-9 Technical I/O Manual***.

### *Basic Functional Requirements of Drivers*

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

The name by which the device is known to the system is the name of the device descriptor module. OS-9 copies the initialization data of the device descriptor to the path descriptor data structure for easy access by the drivers.

A device driver is actually a package of seven subroutines that are called by a file manager in system state. Their functions are:

- Initialize the device controller hardware and related driver variables as required.
- Read a standard physical unit (a character or sector, depending on the device type).
- Write a standard physical unit (a character or sector, depending on the device type).
- Return a specified device status.
- Set a specified device status.
- De-initialize the device. It is assumed that the device will not be used again unless re-initialized.
- Process an error exception generated during driver execution.

The interrupt service subroutine is also part of the device driver, although it is not called by the file manager, but by the kernel's interrupt routine. It communicates with the driver's main section through the static storage and certain system calls.

## *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. Currently OS-9 does not make use of this, but future revisions will require all device drivers to be system state modules. A sample assembly language header is shown in Figure 3-4.

The execution offset in the module header (M$Exec) gives the address of an ***offset table***, which specifies the starting address of each of the seven driver subroutines relative to the base address of the module.

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

The driver subroutines are called by the associated file manager through the offset table. 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 registers. Other parameters that depend on the device type and subroutine called may also be passed. These are described in individual chapters concerning file managers in the ***OS-9 Technical I/O Manual***.

### *INITIALIZE and TERMINATE*

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

### *READ, WRITE, GETSTAT and SETSTAT*

| | |
|---|---|
| (a1) | address of the path descriptor |
| (a2) | address of the driver's static variable storage |
| (a4) | address of the process descriptor requesting the I/O function |
| (a5) | pointer to the calling process' register stack |
| (a6) | address of the system global variable storage area |

### *ERROR*

This entry point should be defined as the offset to error exception handling code or zero if no handler is available. This entry point is currently not used by the kernel. However, it will be accessed in future revisions.

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.

```
* 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    Error      Error handler routine offset (0=none)
```

*Figure 3-4: Sample Driver Module Header Format*

## Interrupts and DMA

Because OS-9 is a multi-tasking operating system, you obtain optimum system performance when all I/O devices are set up for interrupt-driven operation.

For character-oriented devices, set up the controller to generate an interrupt upon the receipt of an incoming character and at the completion of transmission of an out-going 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), set up the controller to generate an interrupt upon the completion of a block read or write operation. It is not necessary for the driver to buffer data because the driver is passed the address of a complete buffer. Direct Memory Access (DMA) transfers, if available, significantly improve data transfer speed.

Usually, the Init routine adds the relevant device interrupt service routine to the OS-9 interrupt polling system using the F$IRQ system call. The controller interrupts are enabled and disabled by the READ and WRITE routines as required. TERM disables the physical interrupts and then takes the device off the interrupt polling table.

The assignment of device intercept priority levels can have a significant impact on system operation. Generally, the smarter the device, the lower you can set its interrupt level. For example, a disk controller that buffers sectors can wait longer for service than a single-character buffered serial port. Assign the Clock tick device the highest possible level to keep system time-keeping interference at a minimum.

The following table shows how you can assign interrupt levels:

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

***End of Chapter 3***

## *NOTES*