CHAPTER 7

THE OS-9 I/O SYSTEM

The OS-9 input/output system has several unique features, making it very flexible and easy to customize. The I/O (input/output) system is tree structured. All I/O system calls go to the kernel, which routes the call to the appropriate file manager module handling that class of device. To perform physical device operations, the file manager calls the device driver module for the specific interface device.

There can be any number of file managers (or none), and for each file manager there can be any number of device drivers. Each file manager handles a particular class of devices. For example, the Random Block File manager (**RBF**) handles randomly accessible block structured devices such as hard and floppy disks, while the Sequential Character File manager (**SCF**) handles character stream devices such as video terminals and printers.

In order to maintain the broad applicability of a file manager, it deals only with logical data operations - it has no understanding of how the data is physically transferred. The physical transfer of data is performed - on the request of the file manager - by the device driver that has been specifically written for a particular I/O interface device, such as a floppy disk controller chip, a serial communications chip, or an intelligent network controller board.

Each device is described by a special data module known as a device descriptor. The device is known by the name of the device descriptor module, preceded by a '/'. For example, the device descriptor module **term** describes the device '/term'. The device descriptor contains the names of the file manager and device driver modules to use to manage the device, essential information about the device - such as the address of the interface - and a set of options fields for controlling the device behaviour.

This approach has two important benefits. It allows the same device driver to be used for any number of I/O interfaces that use the same interface chip, by having a separate device descriptor for each interface, giving a different port address and interrupt vector. Also, multiple device descriptors with different names can be created for the same interface, but with different options settings, or even with a different file manager or device driver name.

For example, two device descriptors could refer to the same serial port, one with options appropriate for its use in communicating with a terminal, another for communicating with a printer. The two device descriptors are "aliases" for the same device. Or, the two device descriptors could specify different device drivers, one for asynchronous communication, another for HDLC synchronous communication. (In the latter example, the device descriptors are not considered simply aliases for the same device, because different device drivers are specified).

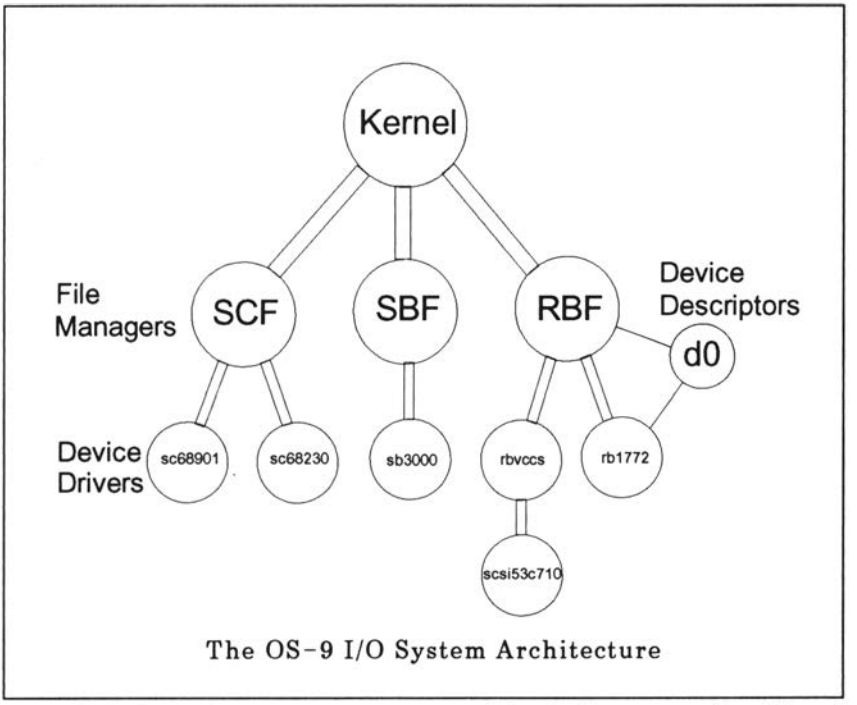
One of the special features of the OS-9 I/O system is the dynamic initialization and termination of I/O sub-systems. Under OS-9, a device does not need to be explicitly initialized by the user. The kernel will automatically initialize the device (if it has not already been initialized) when a path is opened on the device. This feature is described in more detail below.

7.1 I/O SUB-SYSTEMS AND DEVICES

An "I/O sub-system" comprises a file manager module, a device driver module, a device descriptor module, and a device static storage memory area. The I/O sub-system is held together by the device table entry, which contains the addresses of each of these items. (This simple view is made slightly more complex by the possibility of "alias" device descriptors).

A "device" is a physical data store or data conduit. It is difficult to separate the I/O interface on the computer (such as a serial communications chip) from the external data store (such as a disk drive). Some devices have no data store - they exist only as data conduits. Examples are interface chips for serial communications and networking, ADC (analogue to digital converter) chips, and graphics display circuits. Whether or not the device has a data store, it must have an interface on the computer - a microprocessor cannot directly handle any external objects.

The interface is an electronic circuit that appears to the microprocessor as a set of memory locations, and provides the means for the microprocessor to handle the device data, status, and control functions. The interface may be a



The OS-9 I/O System Architecture

simple digital circuit, or one or more specialized chips, or an "intelligent" circuit with its own microprocessor.

The OS-9 concept of a device includes abstract devices that have no interface or data store - they exist only as a function of software. Examples are pipes (sequential memory buffers) and RAM disks (a simulation of a disk drive using computer memory). Programs and the kernel make no distinction in their use of real and abstract devices.

I/O sub-systems are dynamically created and dismantled. When a path is opened on a device, the kernel implicitly executes an **I$Attach** system call. This system call (which is part of the kernel) links to the device descriptor module, and then searches the device table for an entry with the same device descriptor address. If a matching entry is not found, the I/O sub-system does not exist, and must be created. The kernel:

1) Gets the file manager and device driver names from the device descriptor.

2) Links to the file manager and device driver modules.
3) Builds an image of the desired device table entry.
4) Sets the device use count in the image to 1.
5) Allocates the device static storage.
6) Calls the initialization routine of the driver.
7) If the driver returned no error, copies the device table entry image to the device table. Otherwise, "detaches" the device (see below).

If a matching entry is found, however, **I$Attach** performs only steps 1 and 2, and increments the device use count.

Note that the **I$Attach** system call can also be made explicitly - this is what the **iniz** utility does. This ensures that the device is initialized (if the I/O sub-system did not already exist), and prevents the device from being terminated even if there are no paths open on it. The most common example of the use of this feature is with RAM disks. The RAM disk device driver uses an area of memory to simulate a disk drive, which can be used to store temporary files or copies of commonly required files, to speed up access to these files. Normally this memory is dynamically allocated from system memory when the I/O sub-system is initialized, and de-allocated when the I/O sub-system is terminated. The **iniz** utility is used to ensure that the RAM disk remains in existence even if no paths are open to files on the RAM disk:

```
$ iniz /r0
```

If this were not done, the following sequence of operations would give no error, but an unexpected result:

```
$ copy /h0/startup /r0/startup
$ dir /r0
```

The **copy** would open a path to the RAM disk, causing it to be initialized, copy the file to the RAM disk, and then close the path, causing it to be terminated. The **dir** causes the RAM disk to be initialized again, and reads the root directory of the freshly initialized RAM disk - the file has apparently disappeared!

The complement to the **I$Attach** system call is the **I$Detach** system call. The kernel implicitly makes this system call when a path is closed with no remaining duplications (so the path descriptor is about to be de-allocated). An **I$Detach** system call on an I/O sub-system with a device table use count

of one causes the kernel to dismantle the I/O sub-system and delete the device table entry. The kernel:

1) Calls the device driver termination routine (ignoring any returned error).
2) De-allocates the device static storage.
3) Unlinks from the device descriptor, file manager and device driver.
4) Deletes the device table entry.

If the use count was not at one, **I$Detach** unlinks from the device descriptor, file manager, and device driver, and decrements the device use count.

Note that the **I$Detach** system call can also be made explicitly - this is what the **deiniz** utility does. This allows the user to terminate an I/O sub-system that is being held in existence even though there are no paths open on it because a previous explicit **I$Attach** call was made on the device (such as by the **iniz** utility).

A more detailed description of the operation of **I$Attach** and **I$Detach** is given below, in the section on The I/O System Calls.

For the system calls **I$MakDir** (make directory), **I$ChgDir** (change directory), and **I$Delete** (delete file) the kernel opens a path, calls the appropriate file manager function, and then closes the path. This is important for two reasons:

a) All the file manager functions, including those above, are called with an open path (an initialized path descriptor exists).
b) The device is guaranteed to be initialized when a call is made to any file manager function.

The kernel also increments the device use count before closing the path in the **I$ChgDir** system call. This is to prevent the I/O sub-system from being terminated if there are no paths currently open on the device on which the default directory resides. Therefore a user wishing to delete an I/O sub-system (that has no paths open on it) must call **deiniz** as many times as **iniz**, **chd**, and **chx** together were called on that device. This is commonly experienced with RAM disks. The **devs** utility can be used to show the

current use count on all devices. (The display from **devs** refers to the use count as "links").

Because the **I$Attach** and **I$Detach** system calls can be made explicitly, there may be no path open on the device when the call is made. Therefore the kernel calls the initialization and termination functions of the device driver directly, without calling the file manager. This is to maintain the philosophy that the file manager functions are always called with an open path. The initialization and termination functions are the only device driver functions called by the kernel. All other device driver functions are called only by the file manager.

7.2 FILE MANAGERS AND DEVICE DRIVERS

Microware have separated the code components of an I/O sub-system into a file manager and a device driver. This is a convenience - the split in functionality can be at any level. For example, the pipe device driver does nothing. The operation of pipes cannot vary from system to system, because they are simply memory buffers, so the pipe file manager can contain all the functional code without fear that this will restrict the portability of the I/O sub-system to other computers. Because many device drivers may be written to work with one file manager, the functionality of the device driver and the interface between the file manager and the device driver is a convention defined by the writer of the file manager.

Normally, however, the file manager contains all the code for the logical manipulation of the data for devices of a particular type. For example, **RBF** contains all the functionality for handling a hierarchical filing system. The device driver has only the task of carrying out physical operations on the device (at the request of the file manager).

The separation into two modules has these benefits:

a) Multiple device drivers (for different devices of a similar type) can use the same file manager, saving on development effort (and memory). Existing file managers can be used wherever possible, independent of the actual hardware on a particular system.

b) The device driver writer does not have to understand how the filing system works.

c) The device driver writer has the minimum task to perform - he need only provide low-level physical control of the device.

d) The device driver writer has a limited (and therefore more easily learned and understood) programming environment.

These advantages simplify and speed up the porting of OS-9 onto new hardware. It should not be thought, however, that the existence of the file manager level makes it impossible for the device driver writer to include special functionality in the device driver. The "get status" and "set status" system calls (described below) can be used to send requests directly to the device driver - the file manager passes on the call without interpretation - so the device driver can implement any number of special features appropriate to the particular device or application.

7.3 DEVICE DESCRIPTORS

Each I/O sub-system is described by a small data module known as a device descriptor. Multiple device descriptors (of different names) may exist to describe the same device, specifying different optional properties or just a different name. Paths to more than one such "alias" can be open simultaneously.

A device is known by the name of its device descriptor module preceded by the '/' character. The "alias" feature of the I/O system means that the same device may be known by more than one name.

A device descriptor contains a standard section and an options section. The standard section follows after the header parity word of the module header. It is defined in the file 'DEFS/module.a'. In the following description, the offsets are relative to the start of the module header:

Offset	Name	Size	Description
$030	M$Port	1	The device "port address" - the memory address of the registers of the interface chip used to control the device. The kernel only uses this field to check whether a device descriptor is only an alias of another device descriptor already in the device table, and to initialize the **V_PORT** field of the device static storage. The use of this field in actually accessing the chip is a function of the device driver only. This field is intended to allow a device driver module to be used on any number of interfaces that use the same type of chip.

Offset	Name	Size	Description
$034	M$Vector	b	The interrupt vector to be used by the device. This field is not used by the kernel - it is for the use of the device driver when installing an interrupt handler. As with **M$Port**, this field is intended to allow a device driver module to be used on any number of interfaces that use the same type of chip. The value set in this field must conform to the requirements of the interface chip. If the chip supports a programmed vector number, this field can be set to a unique number for each chip, so the interrupt handler of the device driver does not need to poll to see which chip interrupted. Note that some chips generate more than one vector (relative to a base value), depending on the cause of the interrupt. In this case, this field should be set with the base value to be programmed into the chip. The chip, or its supporting circuitry, may not support normal vectoring - the hardware is configured to request autovectoring from the processor. In this case, this field must be set to the appropriate autovector - the interrupt level generated by the chip, plus 24.
$035	M$IRQLvl	b	The interrupt level of the device. This is usually a hardwired feature of the circuit incorporating the interface chip. Sometimes it is a link option on the board, and some chips or supporting circuitry support a programmable interrupt level. If the interrupt level is hardwired, this field must be set to that value (1 to 7). If it is a link option, this field must match the link setting. If it is a programmable setting or link option, use the philosophy described in the chapter on Device Drivers when deciding on the interrupt level to use.
$036	M$Prior	b	The interrupt software polling priority. If the device has been assigned a unique vector number, this field should be zero. The kernel will give an error if a device driver tries to install an interrupt handler on a vector if an entry already exists for that vector and the specified polling priority of the new or existing handler is zero. Some devices or device drivers absolutely require this restriction, because for those devices the vector number returned by the device on interrupt is the only information that distinguishes which device is generating the interrupt. If there is more than one device installed on the same vector the kernel creates a linked list of interrupt table entries in polling priority order (low priority values first). This is the order in which the kernel will call the interrupt handlers on that vector until one of them indicates that it has handled the interrupt.

Offset	Name	Size	Description
$037	M$Mode	b	The device capabilities. This is a byte of bit flags, enabling use of the device as follows: Bit 0 read 1 write 2 execute 5 supports "initial file size" 6 supports "non-sharable" files 7 supports directories When a path is opened on a device, the kernel checks that the access mode requested in the open call is compatible with the capabilities of the device.
$038	M$FMgr	w	Offset to a string giving the name of the file manager to use.
$03A	M$PDev	w	Offset to a string giving the name of the device driver to use.
$03C	M$DevCon	w	Offset to an optional table of extra information about the device. If this field is zero, no such table exists. The structure and use of such a table is defined by the device driver writer.
$046	M$Opt	w	Size of the options section.
$048			The options section of the device descriptor.

The options section contains information about the configuration of the device. The structure of the options section is defined by the file manager writer, although specific device drivers may define additional locations to configure special devices (in general, this is not recommended). The options section of the device descriptor (up to a maximum of 128 bytes) is copied to the options section of the path descriptor whenever a path descriptor is created by the kernel.

7.4 PATHS, PATHLISTS, AND FILES

A "path" is a logical conduit for data, commands, and status between a program and a device, or a data structure within a device. A program would not normally access an I/O interface directly (although it is perfectly possible to do so), because this would bypass all the resource management, file handling, and interrupt handling benefits of the operating system. Instead, the program opens a path to the device (or data structure within the device) by using a system call. The operating system returns a path number, which the program uses to identify the path in subsequent operations of read, write,

status, and control. When the program has finished with the device it makes a system call to close the path, closing the logical conduit.

The operating system manages the path through the path descriptor memory structure, which it allocates when a path is opened, and de-allocates when the path is closed. The kernel automatically closes any paths that a process has open when the process dies.

A "file" is a data structure within a device that has a data store. The concept of a file allows one device to be used to store multiple sets of data, and (on most devices) for the data to be modified, extended, or truncated. File management is a function of the particular file manager used with the device, and so will vary between devices. In general, file managers are written to give as similar a programmer's view of files as possible, to make programs more portable.

The best known use of files is on a disk drive. A disk drive is a "random access device" - the computer can read data blocks from all over the disk in any order without a great delay. This makes it feasible to create, modify, extend, truncate, and delete files by allocating space to each file as necessary. To keep track of the files the operating system must maintain one or more "directories" on the disk. A directory is a file that contains a list of file names and positions of the files on the disk. Because a directory is a file, the files in a directory may themselves be directories, creating a so-called tree structure or hierarchy of directories. There must be at least one directory on the disk. This directory is known as the root directory (because it is the root of the tree).

A "pathlist" is a character string identifying a device and/or a file, used in the system call to open a path. A pathlist may have multiple name elements, separated by separator characters. In its simplest form a pathlist is just a device name:

```
/d0
```

or a file name:

```
fred
```

If a device supports files, the device name is taken to refer to the root directory of the device. For example:

```
$ dir /d0
```

will display the root directory of the device '/d0', while:

```
$ list /d0/fred
```

will list the file 'fred' in the root directory of the device '/d0'.

If the pathlist does not start with a device name, it is taken to be relative to the current data directory of the process, or - if the path is opened with the execute mode set - relative to the current execution directory of the process. Because some file managers support directories - which may be hierarchical, or only a root directory - there must be some way of expressing the route through the directory tree to the file required. This is done by stringing the names of the directory files together to make the pathlist, in the order in which the route must be followed.

Under OS-9, the convention is that the name elements are separated by the '/' character, (but bear in mind that this is a function of the file manager, not the kernel). For example, if the device '/d0' has in its root directory a directory called 'GEORGE', and that directory has within it a directory called 'JIM', and within the directory 'JIM' is a file 'henry', then the following command line would be used to list the file 'henry':

```
$ list /d0/GEORGE/JIM/henry
```

Notice that by convention directories are given names in upper case. This is for convenience - it makes it obvious which files are directories when a directory is listed. The kernel uses the system call "parse name" (**F$PrsNam**) to check device names (and all module names). This call ignores letter case. File managers (including **RBF**) usually also use this system call for the other elements of the path list, so file names are not sensitive to letter case (unlike UNIX). For example, the command line shown above would produce the same result if entered as:

```
$ list /d0/george/jim/henry
```

Again, bear in mind that the kernel only parses the device name, stopping at the first character that is not a legal character within a name. The **F$PrsNam** system call - used to check module and device names, and the names of files being created - places the following restrictions on names:

a) Legal characters are numbers, letters, the underscore character '_', the period character '.', and the dollar character '$'.

b) The name must contain at least one number, letter, or underscore.

The **F$CmpNam** system call used to compare the name of a file being opened ignores letter case, and implements wild carding of names. The '*' character matches any number (including zero) of characters, up to the next

occurrence in the target string of the character following the '*' in the match string, or to the end of the target string if the '*' is the last character in the match string. The '?' character matches the next character in the target string whatever it is.

Match string	Matches
f*d	fred, fold, folded, fd
fr*	fred, fr, fritter
f?r	fur, far, for
f*d?d	folded, fielded

7.5 PERMISSIONS, ATTRIBUTES, AND MODES

Devices and files have associated flags to restrict access. These flags are known as "permissions" or "attributes" (there is no difference). When a program opens a path to a device or file, it specifies the "mode" in which it wishes to access the path. The "mode" is a set of flags indicating the type of operations the program wishes to subsequently perform on the path. The operating system checks that the requested mode matches the available permissions of the device or file, and returns an error (such as **E$FNA** - file not accessible) if they do not. The permissions and mode flags are bit flags within a byte or word field. The permissions field may have separate sub-fields for user, group, and public permissions.

When a path is opened the kernel checks the requested mode against the device permissions as part of the **I$Attach** system call. The **I$Attach** system call checks that all of the mode flags that are set are matched by the equivalent permissions flags in the device descriptor module (except the "sharable" flag, which **I$Attach** ignores). It is the responsibility of the file manager to check the requested mode against the individual file permissions. RBF checks that the requested mode is valid for each directory/file in a pathlist. Note that even if a pathlist does not start with a device name, but is relative to the current execution or data directory, the kernel still performs an **I$Attach** system call for the device, and so checks the requested mode against the device permissions. If the mode specifies execute access, the kernel checks the device of the current execution directory, otherwise it checks the device of the current data directory.

The basic permission (and mode) flags are read, write, and execute. For example, when used in a permissions field, the "read" flag indicates that read operations are permitted on the device or file. When used in a mode field, the

"read" flag indicates that the program wishes to be able to make system calls to read data from the path. The mode flags - used in a system call to open a path (including creating a new file) - are:

Bit	Meaning when set
0	Read
1	Write
2	Execute
3	Not used
4	Not used
5	Initial file size is specified (when creating a file)
6	Non-sharable
7	Directory file

Bit 5 - used in a call to create a file - is not used by the kernel, and indicates to the file manager that the program is explicitly giving the size of the file to create. Otherwise the initial size depends on the file manager. For example, RBF will create a file of zero length, while the pipe file manager **pipeman** will allocate a pipe of about 90 bytes.

Bit 6 indicates that the calling program wants to be the only process using the file or device. The **I$Attach** system call ignores this flag, but if this flag is set the kernel returns an error when trying to open a path if another path is already open on the device. Notice that the kernel does not perform the reverse check - if this flag is not set, the kernel will allow a path to be opened on a device even if the device has already got another path open on it that was opened with this flag set. Also, the kernel skips this check altogether if the mode or the device permissions include bit 7 - the directory flag.

The permissions flags for a device are held in the **M$Mode** field of the device descriptor module header. Note that these are the permissions available for opening paths on the devices, as opposed to the module access permissions **M$Accs** in the module header, which give the permissions available for linking to a module. The device permissions flags have exactly the same format as the mode flags. If a flag - such as the "initial size" flag - is set, this permits the corresponding mode to be used. If the "non-sharable" bit is set, the kernel will not allow a path to be opened on the device if another path is already open on the device, so only one path can be open on the device at any one time.

The permissions flags for files depend on the file manager. RBF keeps a byte field of permissions with each file:

Bit	Meaning when set
0	Owner or group read
1	Owner or group write
2	Owner or group execute
3	Public read
4	Public write
5	Public execute
6	Non-sharable
7	Directory

The use of single flags for owner and group permissions is a historical legacy of OS-9/6809, which does not have the concept of user groups.

7.6 THE I/O SYSTEM CALLS

The I/O system calls are a special subset of the OS-9 system calls. They provide the facilities for data transfer, and control and status of devices and files. While the other system calls all have assembly language symbolic names beginning with the characters **F$**, the I/O system call names start with the characters **I$**. The kernel manages I/O calls by using the device static storage, device table, and path descriptor memory structures.

OS-9 has a unified, device independent I/O system. Therefore it has a general purpose set of I/O system calls. It is the job of the file manager to produce an effect in response to an I/O call that is as consistent as possible with the OS-9 I/O system philosophy. Because particular devices usually have some special properties that could not reasonably be covered by a generalized set of system calls, two of the calls - **I$GetStt** and **I$SetStt** - are "wild card" calls whose effects vary from file manager to file manager, and device to device. Even with these calls, the device driver writer should try to maintain the same effect for all devices of the same type.

During I/O system calls, when the kernel is making a call to the file manager or device driver it disables the processor data cache(s), unless the compatibility flags in the System Globals indicate that the data caches should not be disabled during I/O accesses (bit 7 of **D_Compat2**), or they indicate

that all the data caches are coherent (**D_SnoopD** is non-zero). When the call to the file manager or device driver is complete, the kernel flushes and enables the data cache(s) it had previously disabled.

In OS-9 version 2.2, after each call to a file manager the kernel would always force a process reschedule by setting the "timed out" flag in the state field of the caller's process descriptor, thus terminating the process's time slice. From OS-9 version 2.3 onwards this is only done if there was another process "I/O queued" on the current process - that is, another process is requesting the resource this process has just finished with. The aim is to maximize I/O usage, as I/O is often the bottleneck in system performance.

For calls made from user state on an already open path (**I$Seek**, **I$Read**, **I$ReadLn**, **I$Write**, **I$WritLn**, **I$GetStt**, **I$SetStt**, **I$Close**, and **I$SGetSt**), the kernel converts the caller's local path number to a system path number through the path number conversion table in the caller's process descriptor. The exception is the call **I$SGetSt**, as this call is explicitly made with a system path number. Calls made in user state that open a new path return a local path number, and store the system path number in the caller's process descriptor path number conversion table.

Calls made in system state on an already open path expect a system path number, and perform no path number conversion. Similarly, calls that open a new path return a system path number, and do not update the path number conversion table.

The following descriptions of the I/O system calls concentrate on the behaviour of the kernel. Further detail of the behaviour of the **RBF** and **SCF** file managers is given in the section on File Managers.

7.6.1 I$Attach: Add Device to Device Table

The **I$Attach** system call takes a device name string and a set of mode flags, and ensures that the device is installed in the device table and initialized.

Note that this call is made implicitly by the kernel whenever a path is opened. If the pathlist does not start with a device name, the kernel uses the device table entry address stored in the caller's process descriptor to get the address of the device descriptor on which the current directory is located, and performs an **I$Attach** on that device (the current execution directory if the mode flags include the execute flag, otherwise the current data directory).

I$Attach performs the following sequence of operations:

a) Skip a leading '/' character if present.

b) Link to the device descriptor module of the given name.

c) Link to the device driver and file manager modules whose names are in the device descriptor.

d) Search the device table for an entry for the same device descriptor, or an entry with a different device descriptor specifying the same device driver and port address (an alias). If an entry for the same device descriptor address is found:
   - Check the device static storage address in the existing device table entry. If it is zero, the I/O sub-system is being dismantled - the device driver's terminate routine is currently being executed (it must have gone to sleep).
   - In that case, I/O queue on the process that is terminating the I/O sub-system - its process ID is in the "use count" field of the device table entry. (This prevents a call being made on a device that is in the processing of being terminated.) On wakeup, check again (the device table entry will have been deleted, unless the process was woken for another reason).

e) If an existing entry for the device descriptor was not found, find an empty entry and build an image of the new entry (in private memory).
   - If the new entry was found to be an alias for an existing entry, copy the address of the device static storage from the existing entry to the image of the new entry.
   - Otherwise, allocate and initialize the device static storage, and call the initialization routine of the device driver.
   - In either case, then copy the image of the device table entry to the new entry, and set the use count in the new entry to one.

f) If an existing entry was found, increment the use count (unless it is at the limiting value for a word field, 65535).

g) Check that all the mode flags set in the supplied mode are matched by flags set in the device permissions in the device descriptor. If not, return an error **E$BMode** (but do not detach the device).

Note that the device descriptor, device driver, and file manager modules are simply linked to. The kernel does not automatically load these modules if

they are not present in the module directory. Therefore the modules must either be in ROM or the boot file, or they must be explicitly loaded before the device is used. The 'startup' file is a convenient place to load additional I/O modules that are regularly required on a particular system.

7.6.2 I$Detach: Remove Device from Device Table

The **I$Detach** system call is the complement to **I$Attach**. It is used to remove a device from the device table when the device is no longer in use. The kernel makes this call implicitly whenever it terminates a path - that is, whenever the use count of a path descriptor is decremented to zero, because all duplications of the path have been closed.

I$Detach performs the following operations:

a) Decrement the use count of the device table entry.

b) If the use count is now zero:
   - Get the address of the device static storage from the device table entry, and clear the device static storage field in the device table entry as an indication that the device is being terminated.
   - Look through the device table to see if there is another entry using the same device static storage address. If not, copy the caller's process ID to the "use count" entry of the device table entry, call the termination routine of the device driver, and de-allocate the device static storage.
   - In either case, save the device descriptor address from the device table entry, and clear the device descriptor address and use count fields of the device table entry to zeros, to indicate it is free.

c) Unlink from the file manager, device driver, and device descriptor modules.

7.6.3 I$Dup: Duplicate a Path

The **I$Dup** system call takes the path number of an already open path, and returns a new local path number that accesses the same path. The path use count fields (**PD_COUNT** and **PD_CNT**) in the path descriptor are incremented. (The word field **PD_COUNT** is incremented, and the low byte copied to the byte field **PD_CNT**. If the result in **PD_CNT** is zero, it is set to one). In common with the calls that open a new path (**I$Open** and **I$Create**), **I$Dup** uses the first free entry in the process's path number

conversion table. That is, the lowest available local path number is used. This call is used primarily to redirect the standard input, standard output, and standard error paths (paths 0, 1, and 2 respectively) when forking a child.

By duplicating a path, the process can save a copy of its own standard path, close the standard path, open the desired new path - which will take the path number of the closed standard path, being the first free local path number - and fork the child process. The child process inherits the redirected path. The parent can now close the newly opened standard path, duplicate the saved path again - which will be duplicated to the standard path just closed, being the first available - and close the first duplication. **shell** uses this technique for implementing its redirection features.

The kernel uses path duplication when asked to fork a process. It duplicates the requested number of paths (usually three) from the parent to the new child. This is how a child process "inherits" the standard paths of its parent.

Path duplication is a simple function. Owner permissions do not need to be checked, as the process clearly must already have the necessary permissions to have opened the path. Apart from finding a new local path number for the calling process (or the child, in the case of a fork), the kernel simply increments the use count fields of the path descriptor used to manage the path.

7.6.4 I$Create: Create a File

The **I$Create** system call creates a new file and opens a path to it. File managers that do not support a filing system - such as the Sequential Character File manager (**SCF**) used for character stream devices like terminals and printers - normally treat this just like the **I$Open** system call. **I$Create** takes a pathlist giving the name of the new file, a set of permissions flags that determines the permissions of the new file, and a set of mode flags that determines the mode of the path opened to the file.

A directory cannot be created by this call (the "directory" flag of the permissions must not be set). The **I$MakDir** system call must be used to create a directory.

The kernel treats this call in exactly the same way as an **I$Open** call. The distinction - creating a new file - is made only by the file manager. **RBF** gives an error if a file of the same name already exists (rather than overwriting the existing file).

7.6.5 I$Open: Open a Path

The **I$Open** system call takes a pathlist giving the name of the file or device to open, and a set of mode flags indicating the desired modes of access of the path.

The kernel creates and clears a path descriptor, and allocates a system path number. It initializes the **PD_COUNT** and **PD_CNT** fields of the path descriptor to one, and saves the requested access modes in the field **PD_MOD**. The **PD_USER** field is set to the group and user numbers of the calling process. The kernel then makes an **I$Attach** system call for the device on which the path is being opened, and saves the device table entry address in the **PD_DEV** field of the path descriptor. If the pathlist does not start with a device name, the kernel makes the **I$Attach** call for the device whose device table entry address is stored in the "current data directory" field of the process descriptor, unless the "execute" flag is set in the requested access modes, in which case the "current execution directory" entry is used.

If either the requested mode or the device permissions have the non-sharable flag set, and the requested mode does not have the directory flag set, the kernel checks whether a path is already open on the device (using the linked list of path descriptors whose root pointer is in the device static storage). If so, the kernel "detaches" the device, de-allocates the path descriptor, and returns an error **E$Share** (non-sharable device is in use).

Otherwise, the kernel links the new path descriptor at the head of the linked list of path descriptors open on this device (rooted in the device static storage field **V_Paths**), and copies the options section of the device descriptor to the options section of the path descriptor. This completes the initialization of the path descriptor.

The kernel then calls the file manager. The kernel first checks whether another process is already making a file manager call on the path - the **PD_CPR** (process ID of process using the path) field in the path descriptor is not zero. If so, it "I/O queues" (**F$IOQu** system call) the calling process onto the process that is currently calling the file manager on this path. This puts the calling process to sleep. When it is woken from the I/O queue, the kernel tries again, unless the process has received a signal (other than the wakeup signal that was used to wake it from the I/O queue), in which case the kernel returns the signal code as an error code. (Of course, in the case of an open or create call the path cannot be in use by another process, as it has just been opened, but this same sequence is used for all calls by the kernel to a file manager).

The process ID of the calling process is then copied to the **PD_CPR** field of the path descriptor, indicating that there is currently a call by this process on this path into the file manager, and the kernel calls the appropriate file manager function (in this case the "open" function). On return from the file manager, the kernel "I/O unqueues" the path. It checks whether there is a process I/O queued on the current process (the calling process). If so, it clears the link to that process in the process descriptor of the current process (**P$IOQN** field), and wakes up that process by sending it a "wakeup" signal (signal code **S$Wake**). The kernel then sets the "timed out" flag in the process state flags of the process descriptor of the current process, causing reschedule when the current process next returns to user state.

As mentioned above, if the file manager supports directories, opening a path with a pathlist consisting only of the name of the device opens a path to the root directory of the device:

```
path_num = open("/d0", S_IDIR|S_IREAD);
```

RBF implements a special feature that allows a program to open a path to the whole of a disk, as if it were a file. This feature is requested by appending the '@' character to the device name:

```
path_num = open("/d0@", S_IREAD);
```

A process whose group number is zero can read and write any part of the disk in this way. Other processes cannot write to the disk, and can only read the first few sectors (the disk identification sector and the allocation bitmap sectors). Note that a process has a group number of zero if it was forked by a member of the super user group (group zero), or if it has changed its group number to zero using the **F$SUser** system call (only permitted if the program module was created by a super user).

7.6.6 I$MakDir: Create a New Directory

The parameters to the **I$MakDir** system call are the pathlist of the directory to create, the permissions of the new directory file, and the access mode for opening the path while the file is being created. Like the **I$Create** system call, **I$MakDir** is a request to the file manager to create a new file, but in this case although the kernel opens a path for the benefit of the file manager, it closes the path before returning to the calling program. The file permissions passed by the calling program are not used by the kernel (although they may be used by the file manager). The "write" and "execute" flags are added to the "read" and "execute" flags of the access modes passed by the calling program, to form the access modes used to open the path.

7.6.7 I$ChgDir: Change Current Directory

The **I$ChgDir** system call is used to change the current data and/or execution directories. Like the **I$MakDir** system call, this call temporarily opens a path, calls the appropriate file manager functions, and then closes the path. The parameters are the pathlist of the directory, and the access modes for opening the directory. The kernel adds the "directory" flag to the access modes before opening the path.

If the file manager function is successful, the kernel saves the address of the device table entry for the device on which the directory was opened, in the **P$DIO** field of the caller's process descriptor. If the access modes have the "read" or "write" flag set, the device table entry address is saved to the "current data directory" portion of this field (the first long word of the first half). If the access modes have the "execute" flag set, the device table entry address is saved to the "current execution directory" portion of this field (the first long word of the second half). Flags of both types may be set, in which case both entries are updated.

Before closing the path, the kernel increments the use count of the device table entry for the device on which the directory exists. This prevents the I/O sub-system being deleted by the **I$Detach** call used in closing the path, in case there are no other paths open on the device.

7.6.8 I$Delete: Delete a File

The **I$Delete** system call requests the deletion of a file on a device that supports a filing system. This is another system call that temporarily opens a path, calls the file manager, and closes the path. The parameters are the pathlist of the directory, and the access modes for opening the path. The file manager will also normally insist that the caller has write permission on the file to be deleted. Also, if the file is a directory, a file manager will insist that the directory is empty. In fact, **RBF** does not permit the deletion of a directory. The file attributes must first be changed to make the file an ordinary file, and RBF will only permit this if the directory is empty.

7.6.9 I$Seek: Set the File Pointer

The **I$Seek** system call is made on an open path, and is intended to reposition the current file pointer of a file (that is, the position from which the next read or write will transfer data). The kernel passes this call directly to the file manager.

7.6.10 I$Read: Read Data

The **I$Read** system call is intended to read data from a path without editing or interpretation by the file manager. It is made on an open path, with parameters giving the address of the memory buffer to read to, and the (maximum) number of bytes to read. If the call is made from user state, the kernel checks (using the **F$ChkMem** system call) before calling the file manager that the process has permission to write the requested number of bytes to the indicated memory buffer, and (provided no error is returned from the file manager) adds the number of bytes read to the **P$RBytes** field of the process descriptor. (If the field thereby exceeds the maximum value that can be stored in a long word - $FFFFFFFF - the kernel sets it to $FFFFFFFF).

7.6.11 I$Write: Write Data

The **I$Write** system call is intended to write data to a path without editing or interpretation by the file manager. It is made on an open path, with parameters giving the address of the memory buffer to read from, and the (maximum) number of bytes to write. If the call is made from user state, the kernel checks (using the **F$ChkMem** system call) before calling the file manager that the process has permission to read the requested number of bytes from the indicated memory buffer, and (provided no error is returned from the file manager) adds the number of bytes written to the **P$WBytes** field of the process descriptor. (If the field thereby exceeds the maximum value that can be stored in a long word - $FFFFFFFF - the kernel sets it to $FFFFFFFF).

7.6.12 I$ReadLn: Read Line

The kernel treats the **I$ReadLn** system call exactly the same as an **I$Read** call. However, the intention is that the file manager will end the input when a CR (Carriage Return) control character is read (character code 13), if this occurs before the requested byte count is reached. The file manager may also perform additional data manipulation. For example, **SCF** implements a simple set of line editing functions.

7.6.13 I$WritLn: Write Line

The kernel treats the **I$WritLn** system call exactly the same as an **I$Write** call. However, the intention is that the file manager will end the output when a CR (Carriage Return) control character is written (character code

13), if this occurs before the requested byte count is reached. The file manager may also perform additional data manipulation. For example, **SCF** implements line feed after carriage return, end of line and page pause, and tab expansion.

7.6.14 I$GetStt: Get Status

The **I$GetStt** "get status" system call is a "wild card" call. In combination with the **I$SetStt** system call, this call is intended to provide access to all of the features of the I/O system that cannot be accessed by the other calls. **I$GetStt** is intended to get status about the path, file, or device, while **I$SetStt** is intended to exercise control or change the state of the path, file, or device. An **I$GetStt** or **I$SetStt** call is made on an already open path. In addition to the path number, the caller passes a function code indicating which "get status" or "set status" function is to be executed, together with parameters appropriate to that function.

The kernel implements two "get status" functions itself. After executing the relevant function the kernel also passes the call to the file manager's "get status" routine. Similarly, the file manager will normally pass a "get status" call on to the device driver, even if the file manager has recognized the function code and executed the appropriate function. The kernel or file manager ignores (no error is returned to the caller) an "unknown service request" error (**E$UnkSvc**) returned by the file manager or device driver respectively in response to a call that it has itself recognized. Any other error is returned to the caller.

If the kernel does not recognize the function code, it passes the call directly to the file manager. Similarly, if the file manager does not recognize the function code, it will normally pass the call directly to the device driver. This allows the file manager writer to invent new function codes for functions specific to the class of devices supported by the file manager, and the device driver writer to invent codes for functions specific to a particular device (or mode of operation of the device).

Microware have defined many function codes, covering all the special functions of the file managers and device drivers they have written. The function codes (which all start with the characters **SS_**) are defined in the file 'DEFS/funcs.a'.

The two "get status" function codes recognized by the kernel are:

Code	Name	Description
$0000	SS_Opt	Copy the options section of the path descriptor to the caller's buffer.
$000E	SS_DevNm	Copy the device name (without a leading '/') from the device descriptor to the caller's buffer.

In both functions the kernel checks (using the **F$ChkMem** system call) that the indicated buffer is permitted to the calling process.

7.6.15 I$SetStt: Set Status

The **I$SetStt** "set status" system call is a "wild card" call, complementing the **I$GetStt** system call. It is intended to allow commands and parameters to be sent to a device and its device driver. The kernel does not implement any "set status" calls itself. It passes the calls directly to the file manager.

7.6.16 I$Close: Close a Path

The **I$Close** system call closes an open path. The kernel decrements the **PD_COUNT** use count field of the path descriptor, and copies the low byte to the **PD_CNT** field (if that byte is zero, the kernel copies the high byte of **PD_COUNT** to **PD_CNT**, to ensure that **PD_CNT** only goes to zero if **PD_COUNT** is zero). Provided the **PD_CPR** field is zero, indicating there is not currently a call on the path into the file manager, the kernel calls the "close" function of the file manager. This implies that the file manager is not always called for the closure of every duplication of a path, but it will at least be called for the closure of the last duplication (because there cannot then be any other call currently executing on the path).

If the **PD_COUNT** use count field is now zero, the kernel calls the **I$Detach** function and de-allocates the path descriptor.

7.6.17 I$SGetSt: Get Status on System Path

The **I$GetStt** system call, if called from user state, takes a local path number, so a program cannot get the status of paths other than its own. This is a good security measure, but restricts the facilities of programs used to report the status of other processes, such as the **procs** utility.

The **I$SGetSt** system call therefore provides a means for a program to request information about the paths of other processes, by supplying a

system path number rather than a local path number. The calling process must know the system path number for the path it wants information about. It can find this out by requesting a copy of the target process's process descriptor (using the **F$GPrDsc** system call), and inspecting the process's path number conversion table.

To maintain system security, this system call is restricted to those "get status" functions that the kernel implements itself (get path options, and get device name). The call is only permitted if the calling process is a member of the super user group (group zero), or is the same group and user as the requested path. Furthermore, unlike the **I$GetStt** system call, **I$SGetSt** does not normally pass the call on to the file manager.

The kernel implements a special option, using an options field in the extended header of the kernel module. If bit 7 of this field (the byte at offset $84 from the start of the module header) is set, then the kernel does pass the calls it recognizes on to the file manager after carrying out its own function (and returns no error if the file manager returns the error **E$UnkSvc**).

7.7 PATH DESCRIPTOR OPTIONS

The second half of the path descriptor is the "options section". The kernel copies the device descriptor options table to the path descriptor options section. The structure of the options in the device descriptor and path descriptor are therefore the same. File managers also commonly write additional information about the path or file at the end of the options section, so that the program can inspect this information using the **SS_Opt** function of the **I$GetStt** "get status" system call.

The structure of the options section is defined by the file manager writer. Only the first field is common to all options sections. This is a byte field **PD_DTP**, giving the "device type". This is a code number indicating the nature of the device, and the structure of the options section. Its purpose is to allow programs to determine whether they are dealing with an appropriate device, and to determine the structure of the options section. For example, the **tmode** utility checks whether the path uses either the **SCF** or the **GFM** file manager (type 0 or 11 respectively), and gives an error if not. Microware has defined the following device type codes. The symbolic name also indicates which file manager the code is intended for).

Code	Name	Description
0	DT_SCF	Sequential character device (terminal or printer).
1	DT_RBF	Random block device (disk drive).
2	DT_Pipe	Pipes.
3	DT_SBF	Sequential block device (tape drive).
4	DT_NFM	Microware protocol network device.
5	DT_CDFM	Compact disc drive (CD-I).
6	DT_UCM	User interface communications device (CD-I).
7	DT_SOCK	Logical socket communications device (ISP).
8	DT_PTTY	Pseudo-keyboard device (ISP).
9	DT_INET	Internet protocol networking device (ISP).
10	DT_NRF	Non-volatile memory store (CD-I).
11	DT_GFM	Graphics display device (CD-I).

A program can alter the fields in the options section (at the discretion of the file manager) by using the **SS_Opt** function of the **I$SetStt** "set status" system call. In this way a program can dynamically modify the handling of a device. For example, a screen editor will use this mechanism to disable echoing of input characters. The **tmode** utility uses this capability to alter the options on the standard input, output, or error path.

Some options section parameters are used by the file manager, and so the result of changing them is defined in the file manager documentation. Others are used by the device driver, so the effect of changing them may vary from system to system. For example, some serial port device drivers will re-initialize the device if a change is made to the character format or baud rate values in the options section, while others will not. (The recent device drivers from Microware support this feature, but early ones did not).

As mentioned above, the structure of the remainder of the options section depends on the file manager. The options structures for the **RBF**, **SCF** and **SBF** file managers are described below. The offsets shown are relative to the beginning of the path descriptor, and so start at 128. If the symbolic names are used to access the options section of a device descriptor, an adjustment must be applied, because the options section of a device descriptor starts 72 bytes from the start of the module header. The adjustment can conveniently be symbolically expressed as:

```
M$DTyp-PD_OPT
```

For example, to access the baud rate code field of an **SCF** device descriptor, assuming that the **a1** register contains the address of the device descriptor module header:

```
move.b PD_BAU-PD_OPT+M$DTyp(a1),d0
```

7.7.1 RBF Options Section

Offset	Name	Size	Description
$080	PD_DTP	b	Device type (1 for RBF).
$081	PD_DRV	b	Logical drive number (base 0) - used by RBF as an index into the drive tables in the device static storage.
$082	PD_STP	b	Drive step rate - code depends on the device driver.
$083	PD_TYP	b	Disk type: Bit Description (when set) 0 *before OS-9 version 2.4 - 8" disk (else 5.25")* 1:4 Disk size: 1 8" 2 5.25" 3 3.5" 5 Track 0 is double density 6 Removable (hard disks only) 7 Hard disk (else floppy disk)
$084	PD_DNS	b	Disk density: Bit Description (when set) 0 Double density (MFM) 1 Double track density (96tpi) 2 Quad track density 3 Octal track density
$086	PD_CYL	w	Number of cylinders available for data (different from PD_TotCyls if partitioning is used, or some cylinders are reserved for defect handling).
$088	PD_SID	b	Number of surfaces (sides) available for data (tracks per cylinder).
$089	PD_VFY	b	Verify after write is disabled if this field is not zero.
$08A	PD_SCT	w	Sectors per track (other than track zero).
$08C	PD_TOS	w	Sectors on track zero (surface 0 of cylinder 0).
$08E	PD_SAS	w	Segment allocation size - the minimum number of sectors RBF will allocate when extending a file, to minimize fragmentation.
$090	PD_ILV	b	Physical sector interleave factor, for formatting.
$091	PD_TFM	b	DMA mode to use - device driver dependent.
$092	PD_TOffs	b	First cylinder to use (one for Microware's Universal format, zero otherwise).

$093	PD_SOffs	b	Lowest physical sector number on each track (zero or one).
$094	PD_SSize	w	Logical block size, used by RBF. Prior to OS-9 2.4, RBF only supported a value of 256. Now any power of 2 (starting at 256) is supported.
$096	PD_Cntl	w	Options control word: Bit Description (when set) 0 Do not allow formatting 1 Disable multi-sector requests from RBF 2 Device ID will not change 3 Driver supports **SS_DSize** Get Status call 4 Driver and device can format individual tracks
$098	PD_Trys	b	*(sic)* number of retries by driver on data error. 0 => use driver default 1 => one try only (no retries)
$099	PD_LUN	b	Physical drive number (SCSI LUN).
$09A	PD_WPC	w	First cylinder to use write precompensation (to disable write precompensation, set this field equal to the number of cylinders).
$09C	PD_RWR	w	First cylinder to use reduced write current (rarely used).
$09E	PD_Park	w	Cylinder to park heads on (rarely used).
$0A0	PD_LSNOffs	l	Logical sector number offset for driver to add to RBF requests – used to create partitions.
$0A4	PD_TotCyls	w	Total number of cylinders on disk.
$0A6	PD_CtrlrID	b	Target controller ID (for SCSI).
$0A7	PD_Rate	b	Data transfer rate and rotational speed (for floppy disks): Bit Description 0:3 Rotational speed (rpm) 0 300 1 360 2 600 4:7 Data transfer rate (k bits/sec) 0 125 1 250 2 300 3 500 4 1000 5 2000 6 5000

$0A8	PD_Scs1Opt	1	SCSI options: Bit Description (when set) 0 Host is permitted to assert ATN 1 Driver and interface support target mode 2 Target supports synchronous transfers 3 Check parity on receive
$0AC	PD_MaxCnt	1	Maximum number of bytes driver and interface can transfer in one request. RBF will not ask to transfer more bytes than this. If there is no limit, set this field to $FFFFFFFF.

The following fields are written by RBF to the path descriptor:

$0B5	PD_ATT	b	Attributes (permissions) of the file accessed by this path.
$0B6	PD_FD	1	Logical Sector Number (LSN) of the file descriptor sector of this file.
$0BA	PD_DFD	1	LSN of the parent directory of this file.
$0BE	PD_DCP	1	Position of the directory entry for this file in the parent directory file.
$0C2	PD_DVT	1	Copy of the device table entry address for this device.
$0C8	PD_SctSiz	1	Copy of the sector size used by RBF on this device.
$0E0	PD_NAME	b 32	Name of this file (not the full pathlist) as a null-terminated ASCII string (bit 7 of the last character is not set, unlike the name string in the directory entry).

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section - this is a function of the kernel. However, a "set status" call with the same function code only modifies the first 11 fields, up to and including **PD_SAS** (this is a function of **RBF**).

If the device driver support the **SS_DSize** and **SS_VarSect** Get Status calls to determine the disk and sector sizes, and the device can inform the driver of the relevant values, the following fields can be zero: **PD_CYL**, **PD_SID**, **PD_SCT**, **PD_T0S**, **PD_SSize**, and **PD_TotCyls**.

7.7.2 SCF Options Section

Many of the fields in the **SCF** descriptor options are flags controlling the line editing behaviour of **SCF**. The field description for such flags indicates the behaviour if the field is non-zero. A more detailed description of the behaviour of **SCF** in response to these flags is given in the chapter on File Managers. Several of the other fields are key codes for input editing, pause, flow control, and signal generation. Each feature can be disabled by setting the key code field to zero.

Offset	Name	Size	Description
$080	PD_DTP	b	Device type (0 for SCF).
$081	PD_UPC	b	Flag: force upper case on receive and transmit.
$082	PD_BSO	b	Flag: to erase a character, transmit [BS][SP][BS], else transmit [BS] (where [BS] is the code in the **PD_BSE** field).
$083	PD_DLO	b	Flag: delete line (in response to **PD_DEL**) by erasing the characters, else start new line (appropriate to teletypes).
$084	PD_EKO	b	Flag: echo received characters back to device (normal terminal operation).
$085	PD_ALF	b	Flag: add [LF] after transmitting [CR] ("automatic line feed generation").
$086	PD_NUL	b	Number of [NUL] characters to send after [CR] (normally zero - set non-zero for slow devices that do not support flow control handshaking, such as teletypes).
$087	PD_PAU	b	Flag: pause after transmitting a page of lines (number of lines given by **PD_PAG**) since the last pause or input.
$088	PD_PAG	b	Length of page in lines, including any top and bottom margins.
$089	PD_BSP	b	Key code: "delete character" - usually [BS] $08, sometimes [DEL] $7F.
$08A	PD_DEL	b	Key code: "delete line" - usually [^X] $18. Causes all characters on the current input line to be erased, and the input buffer pointer to be reset.
$08B	PD_EOR	b	Key code: "end of input line" - usually [CR] $0D.
$08C	PD_EOF	b	Key code: "end of file" - usually [ESC] $1B.
$08D	PD_RPR	b	Key code: "reprint current input line" - usually [^D] $04. (Used for devices that cannot erase characters, such as teletypes).
$08E	PD_DUP	b	Key code: "redisplay to end of line" - usually [^A] $01. Causes all characters from the current buffer position to the character before the first [CR] character in the buffer to be displayed as if typed in (allows commands to be repeated, with some editing).
$08F	PD_PSC	b	Key code: "pause at end of next output line" - usually [^W] $17.
$090	PD_INT	b	Key code: "generate interrupt signal" (send **S$Intrpt** to the last process that used the device) - usually [^C] $03.
$091	PD_QUT	b	Key code: "generate quit signal" (send **S$Abort** to the last process that used the device) - usually [^E] $05.

Offset	Name	Size	Description
$092	PD_BSE	b	Character code used to erase a character (see **PD_BSO**) - usually [BS] $08.
$093	PD_OVF		Character to send on input line buffer full - usually the bell character [BEL] $07.
$094	PD_PAR	b	Character format flags (serial communications): Bit Description when set 0 Generate/check parity bit 1 Even parity (else odd) 2:3 Bits per character = 8-field 4:5 Stop bits = field/2 + 1
$095	PD_BAU	b	Baud rate code: Code Baud rate 0 50 1 75 2 110 3 134.5 4 150 5 300 6 600 7 1200 8 1800 9 2000 10 2400 11 3600 12 4800 13 7200 14 9600 15 19200 16 38400 $FF use external clock source
$096	PD_D2P	w	Offset to name string of device for output (echo device). Usually the same as this device (primary device).
$098	PD_XON	b	Flow control "start" character - usually [^Q] $11.
$099	PD_XOFF	b	Flow control "stop" character - usually [^S] $13.
$09A	PD_Tab	b	Tab character, recognized and expanded to spaces by SCF during line output (**I$WritLn**) - usually [^I] $09.
$09B	PD_Tabs	b	Tab position spacing (see **PD_Tab**) - usually 4.
The following fields are written by SCF to the path descriptor:			
$09C	PD_TBL	l	Copy of the device table entry address for this device.
$0A0	PD_Col	w	Column number for next character in line output (used for tabbing).
$0A2	PD_ERR	b	Bit pattern for most recent input character error - format is device driver dependent.

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section - this is a function of the kernel. However, a "set status" call with the same function code only modifies the fields up to and including **PD_Tabs** (this is a function of **SCF**).

7.7.3 SBF Options Sections

The descriptor options structure for the **SBF** file manager is not defined in the file 'DEFS/io.a'. The version of OS-9 supplied at the time of writing only includes the C language header file 'DEFS/sbf.h'. Therefore the symbolic names shown below are those of the C structure 'sbf_opt' in that file. However, Microware has also defined the corresponding assembly language symbols. These are listed in Appendix B.

SBF implements the concept of multiple buffers, so that tape data transfer can continue while the controlling process fills (or empties) the next (or previous) buffer. For systems where the hard disk and the tape drive are on the same interface (typically SCSI), this is usually of no benefit, and the **pd_numblk** field can be set to zero to conserve memory.

Offset	Name	Size	Description
$080	pd_dtp	b	Device type (3 for SBF).
$081	pd_tdrv	b	Logical drive number (base 0) - used by SBF as an index into the drive tables in the device static storage.
$083	pd_numblk	b	(Maximum) number of buffers to allocate. If this field is zero, SBF buffers are not used (transfer is directly to/from the program's buffer).
$084	pd_blksiz	l	Size of each SBF buffer.
$088	pd_prior	w	Process priority for the background process that manages continuing transfer using the buffers.
$08A	pd_flags	w	Device capability flags - the high byte is for use by the file manager, the low byte is for use by the device driver.
$08C	pd_dmamode	w	DMA mode to use - device driver dependent.
$08E	pd_scsiid	b	Target controller ID (for SCSI).
$08F	pd_scsilun	b	Physical drive number (SCSI LUN).
$090	pd_scsiopt	l	SCSI options: Bit Description (when set) 0 Host is permitted to assert ATN 1 Driver and interface support target mode 2 Target supports synchronous transfers 3 Check parity on receive

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section - this is a function of the kernel. **SBF** does not modify any fields for a "set status" call with the same function, but returns no error (unless the device driver generates an error other than **E$UnkSvc**).

7.8 MAKING A NEW DEVICE DESCRIPTOR

Device descriptors are special modules containing data about a particular device. The information is in binary form. The usual way of creating device descriptor modules is by assembling and linking an assembly language file. Microware have provided source files to help the user create new device descriptors. The main work is done by files in the 'IO' and 'DEFS' directories. Which file is used depends on the file manager to be used by the device, because that determines the structure of the options section:

File Manager	File
RBF	IO/rbfdesc.a
SCF	IO/scfdesc.a
SBF	DEFS/sbfdesc.d

These files contain default options section values, which may be overridden. The programmer does not modify these files - they are general purpose. Instead, the programmer creates a separate source file that "includes" the general purpose file at assembly time. Such a source file is created for each device, and normally has the same name as the target device descriptor (with a '.a' extension, being an assembly language source file).

Again, Microware have provided a number of such files (in the 'IO' directory), covering the common device names (for example, 'term.a', 't1.a', 't2.a', 'd0.a', 'd1.a', 'h0.a', 'h0fmt.a'). As with the main descriptor files, the programmer does not normally modify these files (although he may create new ones, using an existing one as a template). Each of these files calls an assembly language macro which the programmer must provide. It is in this macro that the programmer gives the basic information about the device (port address, interrupt vector and level, and device driver name), and overrides the default options values as desired.

For convenience the macros for all the devices in a system are usually contained in one file, called 'systype.d'. This file also usually contains definitions about the system as a whole, such as the memory map of the system. 'systype.d' may be in the 'DEFS' directory, or it may be in a separate

"system" directory from which device descriptors and other operating system components are created for a system. (The latter approach allows multiple target systems to be supported on one development system).

The file 'IO/t1.a' is an example of a descriptor file for an **SCF** device '/t1':

```
nam     T1
ttl     T1 device descriptor module
use     defsfile
use     ../IO/scfdesc.a
T1
ends
```

This file pulls in two other assembly language source files: 'defsfile' (in the same directory as 'systype.d', from where the assembly is performed), and 'IO/scfdesc.a', which does the main work. It also calls the macro 'T1', which must be defined in 'systype.d'. A file for a descriptor for the device '/t2' is almost identical:

```
nam     T2
ttl     T2 device descriptor module
use     defsfile
use     ../IO/scfdesc.a
T2
ends
```

The file 'defsfile' does nothing except pull in two other files:

```
use     ../DEFS/oskdefs.d
use     systype.d
```

Or, if 'systype.d' is in the 'DEFS' directory:

```
use     ../DEFS/oskdefs.d
use     ../DEFS/systype.d
```

The file 'DEFS/oskdefs.d' is supplied by Microware, and includes definitions that cannot be used from a library (due to restrictions of the assembler and linker), such as the module type codes used with the **psect** directive.

The 'T1' macro in 'systype.d' will be similar to this:

```
T1          macro
port        set     $00FFC000
vector      set     27              autovector level 3
IRQLev      set     3
IRQPri      set     5
parity      set     $20             8 bits, no parity, two stop bits
baud        set     14              9600 baud
            SCFDesc port,vector,IRQLev,IRQPri,parity,baud,sc6850
* Default descriptor values can be changed here:
pagpause    equ     OFF
DevCon      equ     0       needed from OS-9 Version 2.4 onwards
            endm
```

The macro 'SCFDesc' is defined in the file 'IO/scfdesc.a'. This macro defines symbolic values for use by the main body of the file, from the parameters passed to the macro. The parameters to the macro are the port address, the interrupt vector, the interrupt level, the character format pattern (for the field **PD_PAR** in the path descriptor), the baud rate code (for the field **PD_BAU**), and the device driver name ('sc6850' in this example).

The file 'scfdesc.a' creates the options section using the 'dc.x' pseudo-operator. For example, the end-of-file field is created by:

```
            dc.b    C$EOF
```

The symbolic values used in 'scfdesc.a' are defined in the library 'LIB/sys.l', from the source file 'DEFS/io.a'. If these symbols are not defined within the 'T1' macro the assembler will generate external references for them in the ROF 't1.r', which will be resolved from 'LIB/sys.l' at link time. If one or more symbols (of the correct names!) are defined in 'T1', then the assembler resolves the references at assembly time, and does not generate corresponding external references. So a statement such as:

```
C$EOF       equ     $1A
```

within the macro 'T1' will override the default value ($1B) for the end-of-file character. The file 'io.a' also defines the symbols 'OFF' and 'ON' (as 0 and 1 respectively), for convenience. For example, the "line feed after carriage return" feature can be disabled by the line:

```
autolf      equ     OFF
```

in the 'T1' macro. Refer to the file 'DEFS/io.a' for a complete list of the symbols used in 'scfdesc.a', and their default values.

The symbol **DevCon** must either be set to zero, or it must be the offset to a table of additional configuration information following the options section. In the source file, this is achieved by placing the table - with the label 'DevCon'

- after the call to the macro 'SCFDesc'. The structure of the additional information is defined by the device driver writer. The above 'T1' macro modified to have such a table might be:

```
T1          macro
port        set     $00FFC000
vector      set     27                  autovector level 3
IRQLev      set     3
IRQPri      set     5
parity      set     $20                 8 bits, no parity, two stop
bits
baud        set     14                  9600 baud
            SCFDesc port,vector,IRQLev,IRQPri,parity,baud,sc6850
* Default descriptor values can be changed here:
pagpause    equ     OFF
DevCon      dc.w    $3456
            dc.b    $78,$9A
            endm
```

To make the device descriptor module from the source files:

```
$ r68 ../IO/t1.a -o=RELS/t1.r
$ l68 RELS/t1.r -l=../LIB/sys.l -O=OBJS/t1
```

If the assembly is done from the 'IO' directory itself, rather than the 'DEFS' directory, or a separate "system" directory, the assembler command line would be:

```
$ r68 t1.a -o=RELS/t1.r
```

If the output file is to go directly to the 'BOOTOBJS' directory within the execution directory, rather than a local directory, the linker command line would be:

```
$ l68 RELS/t1.r -l=../LIB/sys.l -o=BOOTOBJS/t1
```

The lowercase '-o' option causes the output from the linker to be relative to the current execution directory, while the upper case '-O' option causes the output to be relative to the current data directory.

Making **RBF** device descriptors using the file 'IO/rbfdesc.a' is similar, but there are subtle differences. 'IO/rbfdesc.a' defines default values locally, rather than producing external references to be resolved from a library. Therefore to change a default value the symbolic definition must be overridden using the **set** pseudo-operator. For example:

```
SctTrk      set     9               sectors per track
```

These redefinitions must follow the call to the 'RBFDesc' macro defined in 'IO/rbfdesc.a', in order to replace the default definitions. Also, one of the parameters to the 'RBFDesc' macro is a conditional assembly symbol, indicating the disk format, or the nearest to the desired format. 'IO/rbfdesc.a'

uses this symbol with conditional assembly to define default options values appropriate to the desired disk format. Refer to the file 'IO/rbfdesc.a' for a list of the symbols used for the descriptor fields and for the conditional assembly. For example, a macro to create the floppy disk device descriptor 'd0' might be:

```
DiskD0      macro
port        set     $00FFC040
vector      set     64              first normal vector
IRQLev      set     2
IRQPri      set     0       must be the only interface on this vector
            RBFDesc port,vector,IRQLev,IRQPri,rbteac,dd580
* Default descriptor values can be changed here:
SOffs       set     1
DevCon      dc.b    "scsi5380",0
            endm
```

In this example the device driver to use is **rbteac** and the disk format conditional assembly symbol is 'dd580'. The "first sector on the track" symbol 'SOffs' is changed from the default value of 0 to a value of 1.

This is an example of a device descriptor for the Microware SCSI Device Driver System, which uses an additional "low-level" (or "physical") driver (actually a subroutine module) to control the SCSI interface, while the "high-level" (or "logical") device drivers interpret the file manager requests and convert them to SCSI commands. This allows multiple devices, even on different file managers, to work through the same interface. The **M$DevCon** field of the device descriptor is set to the value 'DevCon', which is an offset to the name of the low-level driver module.

A typical device descriptor source file for the device '/d0' would be 'IO/d0.a':

```
            nam     D0
            ttl     D0 device descriptor module
            use     defsfile
            use     ../IO/rbfdesc.a
            DiskD0
            ends
```

assembled and linked by:

```
$ r68 ../IO/d0.a -o=RELS/d0.r
$ l68 RELS/d0.r -l=../LIB/sys.l -O=OBJS/d0
```

It is customary to also produce a "default device" ('/dd') device descriptor for each **RBF** device, using an additional linker command line such as:

```
$ l68 RELS/d0.r -l=../LIB/sys.l -O=OBJS/dd.d0 -n=dd
```

This will produce a file 'OBJS/dd.d0' containing a module called **dd** – all the other fields will be the same as the device descriptor module **d0**.

The file 'DEFS/sbfdesc.d' used to create **SBF** device descriptors is similar to 'IO/rbfdesc.a', in that it uses locally defined default values that can be overridden by the **set** pseudo-operator. It does not use conditional assembly to set default groups of values, and so is rather simpler than 'IO/rbfdesc.a'. Refer to the file 'DEFS/sbfdesc.d' for the symbolic names and default values. A typical **SBF** descriptor macro for a SCSI tape drive (and using no **SBF** buffering) would be:

```
MT0         macro
port        set     $00FFC040
vector      set     64                  normal vector
IRQLev      set     2
IRQPri      set     0       must be the only interface on this vector
            SBFDesc port,vector,IRQLev,IRQPri,sbteac
* Default descriptor values can be changed here:
NumBlks     set     0                   unbuffered operation
ScsiID      set     3                   tape drive SCSI controller ID
DevCon      dc.b    "scsi5380",0
            endm
```

using the source file 'IO/mt0.a':

```
            nam     MT0
            ttl     MT0 device descriptor module
            use     defsfile
            use     ../DEFS/sbfdesc.d
            MT0
            ends
```

and assembled and linked by:

```
$ r68 ../IO/mt0.a -o=RELS/mt0.r
$ l68 RELS/mt0.r -l=../LIB/sys.l -O=OBJS/mt0
```

7.9 SPECIAL FEATURES

The I/O system has many features that are unique to a particular file manager or device driver. This section highlights a few of the more important special features created by Microware.

7.9.1 RBF Disk Caching

Disk caching uses computer memory to temporarily store data read from disk, or yet to be written to disk, with the aim of speeding up disk file operations. First implemented in OS-9 version 2.4, the disk caching capability in **RBF** is a simple sector-orientated caching without write-behind. Because **RBF** is managing the filing system, the caching functions are able to be somewhat "intelligent", preferentially caching sectors

that are more likely to be needed again. Large block transfers are not cached. The performance benefit of this disk caching varies according to the application, and the allocated cache buffer size. Disk caching is by default disabled, and is enabled using the **diskcache** utility.

7.9.2 SCSI Device Driver System

The Small Computer Systems Interface (SCSI) provides a means of accessing up to 7 controllers through a single interface, with each controller handling up to 8 drives. The drives may be of different types – disk drives, tape drives, printers, and so on. This conflicts somewhat with the simple tree structure of the OS-9 I/O system, as several file managers may be acting through one interface, which must be controlled by one device driver.

The problem is resolved very elegantly using a two level device driver approach. The device drivers known to the kernel and the file managers are "high level" (or "logical") device drivers, each handling one type of controller. That is, they understand the requests from the file manager, and how to build SCSI commands for the controller they have been written for, but they do not know how to transact these commands over the SCSI interface. To do this they call an additional module, known as a "low level" (or "physical") device driver (actually a module of type "subroutine"). The high level drivers link to this module as part of their initialization routine (and unlink from it on termination), so they can call the functions within the low level driver.

SCSI also provides for multiple commands to be transacted concurrently over the interface (known as disconnect/reselect). To implement this feature the low level driver needs its own static storage, in order to keep track of multiple commands together. It does this by means of a data module which it creates in memory when its initialization routine is called from the initialization routine of the high level driver. If the data module already exists (so this is not the first high level driver to be initialized), the low level driver simply links to it. The low level driver returns the address of the data module to the high level driver, which passes this address back to the low level driver when calling the "transact SCSI command" routine of the low level driver.

By dynamically building the name for the data module using the address of the SCSI interface in ASCII hexadecimal, the low level driver allows for multiple SCSI interfaces in the same system.

7.9.3 Ethernet support

The Internet Support Package (ISP) from Microware provides the standard 'TCP/IP', 'telnet', and 'ftp' facilities commonly used over Ethernet networks. ISP uses separate "protocol modules" rather than building the protocol interpretation into the file manager. This allows for the easy addition of new protocols in the future, from Microware or other sources.

7.9.4 The X Window System

The X Window System (often referred to as "X Windows") graphical user interface (GUI) package is also available from Microware. Developed at MIT, X is a very sophisticated package, and is the only GUI available for a wide range of operating systems. It is able to work across a network (such as Ethernet), so that the display terminal can be remote from the computer.