CHAPTER 13

FILE MANAGERS

To avoid the unnecessary duplication of many paragraphs, this chapter assumes that you have already read the chapter on Device Drivers.

File managers are essential components of the OS-9 I/O system. An understanding of existing file managers allows the programmer to make the most effective use of the I/O system within an application. However, some applications are best served by writing a new file manager. This chapter aims to describe the purpose of a file manager, and its interface to the kernel and device drivers, to give the system programmer the essential information needed to write a file manager. The Microware RBF and SCF file managers are described as examples, and to help the application programmer make best use of the I/O system.

The descriptions and code fragments in this chapter assume that the file manager is written in 68000 assembly language. However, file managers may equally well be written in C. The chapter on "Microware C and Assembly Language" describes how this is done.

13.1 THE FUNCTION OF A FILE MANAGER

Under OS-9, a file manager performs the filing structure maintenance and data processing for a class of like devices. That is, it performs all the logical operations on the devices. The file manager does not know how to control the hardware. Instead, it makes calls to the appropriate device driver to perform low-level hardware operations. This division of functionality allows one file manager to be used to manage a wide range of devices of a similar type, including devices not existent when the file manager was written. For

example, the Microware Random Block File manager (RBF) provides a hierarchical filing structure on almost any rewritable block structured device, while the Microware Sequential Character File manager (SCF) is suitable for almost all terminals and printers. Other file managers are available from Microware and third-party suppliers.

In addition, the file manager writer is able to concentrate on the problems of logical data manipulation, while the device driver writer handles the complexities of interrupts and VLSI interface chips.

Because one file manager is designed to work with many - as yet unwritten - device drivers, the file manager writer is also responsible for producing the specifications for:

a) The functions and parameter conventions of the device driver routines (other than initialize and terminate).

b) The options section of the path descriptor and device descriptor.

c) The file manager storage in the path descriptor.

d) The file manager storage in the device static storage, including the drive tables (if the file manager supports multiple channels on one device).

While the kernel prevents concurrent accesses on the same path, it is normally the file manager's responsibility to prevent concurrent accesses to the same device. The kernel will queue an I/O call on a path on which there is currently an I/O call being executed by another process. That is, two processes have the same path (the same system path number) open (because one inherited it from the other, or they both inherited it from the same parent or other ancestor). One process has made an I/O call on the path, and been put to sleep during the call by the file manager or device driver (usually only the device driver will sleep, waiting for a device operation to complete). The second process makes a call on the same path. The kernel will queue (put to sleep) the second process until the I/O call of the first process finishes.

The kernel uses the **F$IOQu** system call, which puts the process to sleep, and links its process descriptor onto a linked list of process descriptors rooted in the process descriptor of the process using the path. This creates a "queue" of processes waiting for the process at the root of the queue to finish its I/O operation. In this way the kernel prevents the use of one path descriptor by two processes simultaneously. Note that no queuing is required if the first

I/O call does not sleep, as rescheduling is suspended while a system call is being executed, so the second process would not have the opportunity to make its I/O call until the first call finished.

However, the kernel does not check whether an I/O call on a *different* path is to the same device - perhaps even on the same file - as a currently executing call. This is because the kernel makes no assumption that the device cannot handle multiple requests simultaneously. While the handling of multiple hardware transactions concurrently is really a device driver function, most file managers assume that the device cannot handle more than one transaction at a time. Such file managers therefore prevent concurrent calls into a particular device driver incarnation, by queuing calls that they wish to make to the device driver until the device driver has finished executing any current request. A typical mechanism for doing this is described in the section on Resource Control.

13.2 FILE MANAGER ROUTINES

Each file manager module provides a set of routines to carry out I/O functions on a path. The file manager routines are only called by the kernel, in response to I/O system calls. Therefore the number, basic function, and parameter convention of these routines is fixed. However, because the kernel is not concerned with filing structures and data processing, the detail of the function of each routine may vary significantly between file managers.

The file manager routines correspond directly to the OS-9 I/O system calls:

System call	Function	Description
I$Create	Create	Create a new file, and open a path to it.
I$Open	Open	Open a path to an existing file or device.
I$MakDir	Make directory	Create a new directory.
I$ChgDir	Change directory	Change the current data and/or execution directory of the calling process.
I$Delete	Delete	Delete a file.
I$Seek	Seek	Set the file pointer of an open path (the position within the file for the start of the next read or write).
I$Read	Read	Read data without editing.
I$ReadLn	Read line	Read data, perhaps with line editing, ending on Carriage Return or other end-of-record character.
I$Write	Write	Write data without editing.

System call	Function	Description
I$WritLn	Write line	Write data, perhaps with line editing, ending on Carriage Return or other end-of-record character.
I$GetStt	Get Status	"Wild card" call to get information about a device or path.
I$SetStt	Set Status	"Wild card" call to send information to or request action on a device or path.
I$Close	Close	Close a path.

A file manager can choose not to implement one or more functions, returning (with or without error) directly to the kernel, or perhaps passing the call without interpretation to the device driver. This is particularly the case for the file-related functions - Make Directory, Change Directory, Delete, and Seek - as these are not appropriate to certain types of device (those that cannot support a filing system), and for unrecognized sub-functions of the Get Status and Set Status calls.

13.3 KERNEL ACCESS TO THE FILE MANAGER

The kernel accesses the file manager routines by means of a table of offsets to the routines, similar to that for device drivers. The offset to the table from the start of the module header is in the "execution offset" entry (**M$Exec**) of the module header. Unlike the device driver offset table, however, the entries are relative to the start of the table, not the start of the module. For example:

```
EntryTable  dc.w    Create-EntryTable   create a file
            dc.w    Open-EntryTable     open a file
            dc.w    MakDir-EntryTable   make a directory
            dc.w    ChgDir-EntryTable   change default directory
            dc.w    Delete-EntryTable   delete a file
            dc.w    Seek-EntryTable     set the file pointer
            dc.w    Read-EntryTable     read data
            dc.w    Write-EntryTable    write data
            dc.w    ReadLine-EntryTable read data with line editing
            dc.w    WriteLine-EntryTable    write data with line
*                                           editing
            dc.w    GetStat-EntryTable  get information
            dc.w    SetStat-EntryTable  send a command
            dc.w    Close-EntryTable    close a path
```

After each call to the file manager, if the I/O queue (**F$IOQu**) rooted in the process descriptor of the calling process is not empty, the kernel marks the calling process as "timed out", forcing a reschedule when the system call finishes. (In OS-9 version 2.2 the calling process was marked as timed out

even if the I/O queue of the calling process was empty.) The kernel also performs an "I/O unqueue" operation. That is, if the I/O queue of the calling process is not empty, the calling process is detached from the queue, and the first process in the queue is woken up. This allows other processes to get a chance at the path or device, rather than letting the same process make another request and "hog" the path or device. It also improves the throughput of the I/O device - often the bottle-neck in a system - by quickly giving time to a process that wants to use the device, rather than allowing the previous process to finish its time slice.

13.4 PARAMETER CONVENTION

The kernel calls all the file manager routines with the same parameters:

(a1) = Path Descriptor
(a4) = Calling process's Process Descriptor
(a5) = Caller's register stack frame
(a6) = System Globals

The kernel sets up the following path descriptor locations before calling the file manager:

PD_CPR	Process ID of the calling (current) process
PD_LProc	Same as **PD_CPR**
PD_RGS	Address of caller's register stack frame (same as **a5**)

Note that the file manager is always supplied a properly initialized path descriptor. The kernel allocates and initializes a new path descriptor before calling the Create, Open, Make Directory, Change Directory, and Delete routines of the file manager. The initialization of the path descriptor includes attaching the device (**I$Attach**), even if the pathlist does not start with a "/" (that is, the pathlist is relative to the data or execution directory). The kernel copies the device descriptor options section to the path descriptor options section.

Immediately after calling the Make Directory, Change Directory and Delete file manager routines, the kernel terminates the path by de-allocating the path descriptor and executing an **I$Detach** system call, because these calls do not return an open path to the calling program. The kernel also terminates the path after calling the Close routine of the file manager if the use count of the path is zero.

The kernel expects the file manager to preserve the **a5** and **a6** registers, and the high byte of the status register. The file manager may destroy the other data and address registers.

13.5 PATHLISTS

A pathlist is the principal parameter to the Create, Open, Make Directory, Change Directory, and Delete routines. If the pathlist begins with the '/' character, the kernel only interprets the first name element, assuming it to be a device name. Any character not permitted in file names (by the **F$PrsNam** system call) is taken to terminate the device name. Permitted characters are alphanumeric, '.', '_', and '$'. The kernel does not interpret the remainder of the pathlist, or any of the pathlist if it does not begin with the '/' character.

Therefore elements of pathlists and the element separators can follow almost any convention, according to the specification prepared by the file manager writer. For compatibility with UNIX, however, elements are usually separated by the '/' character. RBF, Pipeman, and NFM use this convention. In addition, file managers usually use the **F$PrsNam** system call to parse name elements.

Multiple pathlist elements usually refer to a directory hierarchy, but could be used for other purposes. In effect, the pathlist provides a mechanism for passing an ordered list of character string parameters to the file manager.

13.6 CREATE AND OPEN

For file managers without filing structure support (such as SCF) these calls are usually synonymous, and prepare for I/O on the device. Such file managers will normally give an error if the pathlist is not a simple device name. For file managers that do support a filing structure, Open should prepare for access to an existing file, while Create should create a new file (or give an error if a file of the same name already exists), and open it for access.

13.6.1 SCF

SCF treats Create and Open calls identically. SCF also "attaches" (**I$Attach**) the "echo device" specified in the device descriptor (if one is given), and saves its device table entry address in the path descriptor field **PD_DV2**. The echo device is the device used for output when data is written to the device on which the path was opened (the "primary device"), and is usually the same as

the primary device – that is, keyboard input from a terminal on a serial port is echoed back to the terminal on the same serial port. SCF allocates a buffer of 512 bytes (256 bytes prior to OS-9 version 2.3) for input line editing of subsequent Read Line calls.

SCF calls the device driver's Set Status routine with the **SS_Open** function code. The call is made only for the primary device, even if the echo device is not the same as the primary device.

13.6.2 RBF

RBF parses the pathlist as described above, skipping the device name if the pathlist starts with a '/' character. If the pathlist starts with the '/' character and there are no following name elements, RBF opens the root directory of the device (a Create request with such a pathlist is returned a "file exists" error – **E$CEF**). If there are following name elements, RBF looks in the root directory of the device for the first element. If the pathlist does not start with the '/' character, RBF looks for the first element in the current data or execution directory, depending on whether the "execute" bit (bit 2) of the requested modes byte is set. It then looks for the next element within that directory, and so on.

All elements in the pathlist other than the last must be directories, thus creating a tree structured directory system. In the Open call the last element may also be a directory, provided the "directory" bit (bit 7) of the modes parameter is set. The Create call cannot be used to create a directory – the Make Directory call must be used. The Create call fails with a "file exists" error – **E$CEF** – if the last element already exists in the directory.

At each stage RBF checks that the file or directory permissions contain the requested mode bits, either in the public field of the permissions, or – if the caller is in the same group as the file creator (or is a super user) – in the private field of the permissions. If not, RBF returns a "file not accessible" error – **E$FNA**.

In the Create call, RBF creates the file with the supplied permissions byte, which specifies read, write, and execute permissions for public and private access. Note that if the execute bit is set in the modes byte the pathlist is assumed to be relative to the current execution directory even if neither the public nor the private execute permission bit is set in the permissions byte. Conversely, the permissions byte may set public or private execute permission (or both) even if the modes byte does not have the execute bit set (causing the pathlist to be taken relative to the current data directory). If the

"initial size" bit (bit 5) is set in the modes byte, the file is created with the requested size, otherwise it is created empty (the size is zero).

Note that if an initial file size is requested, RBF will not create the file with more than one segment. If a segment as large as the requested size cannot be allocated, RBF allocates the largest segment it can. Therefore to guarantee that a file is created with the desired size, the Create call should be followed by a Set Status call with the **SS_Size** function code to set the file size.

When creating a file, RBF inserts a new entry in the parent directory, using the first available entry. That is, if a file has been deleted from the directory (the first byte of the file name field is zero), then RBF will use that entry for the new file. If all the entries in the directory are occupied, RBF extends the directory, as it would extend any file. Each directory entry is 32 bytes - 28 bytes for the name, followed by 4 bytes giving the Logical Sector Number (LSN) of the File Descriptor (FD) sector for the file (which RBF allocates and initializes when creating the file). Note that RBF only uses the last 3 bytes of the field, as RBF restricts LSNs to 24 bits. The name string is terminated by having bit 7 of the last character set. All unused bytes in the directory entry are set to zero.

RBF allocates a memory buffer equal to one sector in size for caching the file descriptor sector, and another sector buffer for managing read and write requests that start or end part of the way through a sector.

RBF calls the device driver Set Status routine with the **SS_Open** function code.

13.6.3 The File Descriptor Sector

Each file (including directories) has a File Descriptor sector, giving information about the file. The directory entry for the file contains the LSN of the File Descriptor sector. (The LSN of the File Descriptor sector of the root directory is in the **DD_DIR** field of LSN 0.) The File Descriptor sector contains the user ID and group of the creator (the "file owner"), the file permissions, the date the file was created, the date and time the file was last "modified" (opened for writing), and the file size.

The remainder of the sector contains the file segment list. Each entry in the list is 5 bytes, allowing up to 48 entries if the sector size is 256 bytes. (Prior to OS-9 version 2.4, a bug in RBF caused the last entry not to be used, so the file was limited to 47 segments.) The segment list describes the fragmentation of the file. The first entry points to the first part of the file,

the second entry points to the second part of the file, and so on, until the whole file has been located. The first 3 bytes of the entry give a 24-bit LSN, indicating the start of that fragment (segment). The other two bytes are a 16-bit number indicating the length of the segment - the number of contiguous sectors. The length of each segment will vary as required by RBF to build up a file containing the total number of sectors occupied by the file. In general, a file will be more fragmented if it has been extended since its initial creation, or if the disk free space is in many separate small fragments (as a result of many files being created and deleted). The File Descriptor structure is shown below:

Offset	Name	Size	Description
$000	FD_ATT	b	File attributes.
$001	FD_OWN	b 2	File owner - group (first byte) and user ID.
$003	FD_DAT	b 5	Date and time the file was last modified, as YYMMDDHHMM.
$008	FD_LNK	b	Number of links to the file. This field is always one. RBF does not currently support file links, although the Delete routine decrements this field and deletes the file if the field is now zero.
$009	FD_SIZ	l	File size (in bytes).
$00D	FD_Creat	b 3	Date the file was created, as YYMMDD.
$010	FD_SEG	...	The segment list.

Note that the file owner's group and user ID are held as byte values only, not as word values (as in the rest of the system). This is a historical legacy from the original OS-9 for the 6809, which did not have the concept of user groups. Therefore under OS-9/68000 the word field for the user ID was split into byte fields for group and user ID. When checking user and group numbers for file access, RBF uses only the low byte of the caller's group and user ID. For example, a user in group 256 will be able to access files belonging to group 0 (the super user group). For this reason it is not advisable to allocate group numbers and user IDs above 255.

The year numbers in the creation and modification fields are relative to 1900. Thus 1992 is expressed as 92 ($5C). The **FD_Creat** field contains only the date of creation of the file, not the time. The **FD_DAT** field contains the date and time the file was last "modified" (opened with the write bit set in the modes byte). When the file is first created this field is set to the date and time of creation. Note that the time is held only to the minute. For this reason programs that perform an action dependent on the time a file was last updated can only resolve the update to the minute. For example, the **make**

utility may cause a file to be recompiled even though it has not been changed, because the previous compilation happened within a minute of the last edit of the source file, so the "last modified" times of the source and Relocatable Object Files are the same.

As the segment size is held in a 16-bit number, the segment length cannot exceed 64k-1 (65535) sectors. This sets the first limit on the size of RBF files. For a sector size of 256 bytes, the file size is limited to 48*65535*256 = 768Mbytes.

RBF always keeps the filing system up to date on the disk. In particular, any changes to the File Descriptor sector or the Allocation Bit Map (described below) are immediately written to disk. While this makes RBF a little slower than the disk file managers in some other operating systems, it ensures that the filing system is very robust. The computer can be turned off or reset while files are open, and the filing system will not be corrupted in any way. If this happens, files that were being extended (writing at the end of the file) may appear longer than expected. This is because RBF always extends a file by at least the "minimum segment allocation" size given in the **PD_SAS** field of the device descriptor (to reduce file fragmentation), and then trims the file size back to the length actually used when the file is closed. Also, because RBF maintains a sector buffer for data being written to the file, the data most recently written to the file may not have been written to the disk.

13.6.4 The Allocation Bit Map

RBF manages the disk space using an Allocation Bit Map. This map uses a number of contiguous sectors, following the disk identification (ID) sector (LSN 0), and preceding the root directory. Each bit of each byte represents a "cluster" of sectors on the disk. Bit 7 of the first byte represents the first cluster (starting at LSN 0), and so on. A cluster is a group of contiguous sectors. All the clusters on a disk are the same size, but this may be different from the cluster size on another disk. The size of a cluster is set at format time, and recorded in the disk ID sector. It must be an integral power of 2 (1, 2, 4, 8 and so on). In this way RBF can manage a disk with a large number of sectors more quickly by grouping the sectors into clusters, so reducing the size of the bit map. If a bit is set in the bit map, it indicates that the cluster is in use (allocated to a file), or is defective (and so must not be allocated). The bit map is initially built by the **format** utility.

When RBF allocates sectors to a file it always allocates complete clusters. The first sector of the first cluster allocated to the file always contains the File Descriptor sector. Therefore if the cluster size is greater than 1, the first

entry in the segmentation list for the file will always start with the LSN immediately following the LSN of the File Descriptor itself.

When creating or extending a file, RBF determines the number of clusters needed. It then searches the bit map, looking for the first free block (bits are zero) large enough, and allocates the required number of clusters from the start of the block, by setting the bits in the bit map to ones. If there is no block large enough, RBF uses the largest free block that it found during the search as the first new segment of the file. If the file is now large enough to satisfy the caller's request, RBF terminates the allocation (even if the allocation is not as great as RBF had intended). Otherwise RBF searches again, allocating more clusters to more segments until enough space has been found in total (or the disk is full, in which case RBF de-allocates the sectors, and returns a "disk full" error - **E$FULL**).

RBF will not create a file segment whose clusters are not wholly represented within one bit map sector. This means that a segment cannot contain more clusters than 8 times the number of bytes in one sector. For 256 bytes per sector this is a limit of 2048 clusters per segment. This is the other limitation on file size under RBF. Therefore in the case of 256 byte sectors, if the cluster size is less than 32 the limit is the number of *clusters* per segment (2048), otherwise the limit is the number of *sectors* per segment (65535). For example, if the cluster size is 1, the maximum file size is 48*2048*256 = 24Mbytes.

Note that these calculations assume that disk fragmentation does not further limit the size of each segment (for example, two files being written simultaneously will "leap frog" each other in the allocation bit map). Also, a sector size larger than 256 bytes greatly reduces the problem - the segment list for the file is larger (because the File Descriptor sector is larger), each bit map sector is larger, and the sectors allocated to the file are larger.

The cluster size is set by the '-c' option of the **format** utility. Because a small cluster size may restrict the file size on a large disk, while a large cluster size will waste space (disk space is allocated to files in whole clusters), it is reasonable to set a cluster size that allows a file to be as large as the whole disk, not allowing for disk fragmentation. For example, a cluster size of 4 would be suitable for a 100Mbyte disk with a sector size of 256 bytes (giving a maximum file size of 96Mbytes). Bearing in mind that the limitation imposed by the rule that a segment cannot contain more than 65535 sectors, the maximum file size for a disk with 256 byte sectors is 768Mbytes, so it would normally not help to set a cluster size greater than 32 with a sector size of 256 bytes.

If, however, the sector size is 512 bytes, the file descriptor segment list can hold 99 entries, and a bit map sector corresponds to 4096 clusters. The file size limit imposed by the segment restriction of 65535 sectors is 99*65535*512 = 3168Mbytes, and the file size limit imposed by the bit map sector size for a cluster size of 1 is 99*4096*512 = 198Mbytes. Thus a cluster size of 1 would be acceptable for disks up to 200Mbytes, and a cluster size of 16 is the maximum that need normally be used.

13.6.5 Access to the Whole Disk

RBF provides a "whole device" feature in the Open call. If the pathlist consists simply of a device name followed by the '@' character ("commercial at sign"), RBF opens the whole device as if it were a file. The program can then seek, read, and write to the disk without regard for the filing structure. The file pointer of the "file" starts (value 0) from the first byte of LSN zero. For example, a seek to location 1024 followed by a read of 256 bytes would read a sector - LSN 4 - of a disk with 256 byte sectors. Reading and writing can use any block size, just as for a normal file, and the "file size" is the size of the whole disk. Therefore all programs that do not use the file descriptor or directory information of a file can be used without modification. For example:

```
$ dump /d0@
$ merge /r0@ -b100 >/dd/ramdisk_image
```

Users other than the super user group (group 0) are only allowed read access, and are only allowed to read up to the end of the allocation bit map. This protects against damage to the filing system, and prevents unauthorized access to files.

13.7 CHANGE DIRECTORY

This call is normally only implemented by file managers that support a hierarchical directory structure, such as RBF. Other file managers will return an error - SCF returns **E$BPNAM** ("bad path name").

The process descriptor has a 32 byte area (**P$DIO**) for use by the kernel and file manager in remembering which are the current execution and data directories. The first 16 bytes are used for the data directory, and the remaining 16 bytes for the execution directory. The kernel uses the first long word of each half to store the device table entry address of the device with the current directory, and reserves the following word. The remaining 10

bytes of each half are available to the file manager. RBF stores the sector number of the file descriptor sector of the current directory.

As mentioned above, the kernel allocates and initializes a path descriptor (including making the **I$Attach** system call) before calling the file manager. Effectively, the file manager is working with an open path. The kernel "closes" the path and de-allocates the path descriptor after calling the file manager. Note that the kernel does not call the Close routine of the file manager when doing this. Therefore the file manager must perform an implicit closure of the directory file before returning to the kernel.

After calling the file manager Change Directory routine, and providing no error was returned, the kernel increments the device use count in the device table. This is to "hold" the I/O sub-system in existence even though there may be no path open to the device. Note that the kernel does not decrement the use count of the device in the device table when the current directory is changed to another device, so the use count of a disk drive rises each time a "change directory" request is made on it. The **I$Detach** system call (as made by the **deiniz** utility) must be used to decrement the use count to zero if it is necessary to remove the disk drive from the device table.

The calling program passes a "modes" byte, which must have either the read or the execute bit set, or both. If the read bit is set, the current data directory is changed to the requested pathlist. If the execute bit is set, the current execution directory is changed. Thus both the current data and execution directories can be changed (to the same directory) with one call, by setting both mode bits. This protocol is followed by the kernel, and must also be followed by the file manager.

RBF opens the directory in the normal way (so checking that the caller has permission to access the directory for read or execute as requested), and then saves the LSN of the directory's file descriptor sector in one or both of the entries in the path descriptor, depending on the mode bits set. If the write bit was set in the modes byte (as is done in the C library function **chdir()**), RBF also attempts to update the "last modified" date and time of the directory (in its file descriptor sector), but ignores any error (for example, if the disk is write protected).

13.8 MAKE DIRECTORY

This call is normally only implemented by file managers that support a hierarchical directory structure, such as RBF. Other file managers will return an error - SCF returns **E$BPNAM** ("bad path name").

The kernel temporarily opens a path for the duration of this call, in a similar way to the Change Directory call. The kernel treats this call as if it were a Create call with the directory bit set in the permissions byte (which is otherwise illegal), followed by a Close call. The file manager is therefore called with a properly initialized path descriptor, and must perform an implicit close of the file it has just created before returning to the kernel. For its own purposes when "opening" the path, the kernel forces the permissions byte to be directory and write ($82). In OS-9 version 2.3 and later, the kernel also clears all bits in the supplied modes byte other than the read and execute flags (bits 0 and 2), and bitwise ORs the resulting modes byte into the permissions byte, thereby ensuring that any bits set in the modes byte are also set in the permissions byte.

However, the kernel does not modify the caller's stack frame, so the file manager is passed the modes and permissions as specified by the calling program. As with the Create and Open calls, the kernel uses the execute bit of the modes byte to determine whether the directory is to be created relative to the current data or execution directory (unless the pathlist starts with a device name).

RBF uses the execute bit of the modes byte in the same way. The new directory is created with file attributes as specified by the permissions byte supplied by the calling program. The directory bit is also set in the file attributes. In addition to creating the directory (in the same way as any file), RBF writes two directory entries to the new directory - "parent" and "self". The "parent" entry is first, and has the name '..'. Its File Descriptor LSN field contains the LSN of the File Descriptor of the directory that contains this directory. The "self" entry has the name '.'. Its File Descriptor LSN field contains the LSN of the newly created directory itself. This information allows RBF to support pathlists that move up the directory hierarchy, such as '../DEFS'.

A directory is therefore created with an initial file size of 64 bytes. RBF ignores a request to create a directory with a larger size (indicated by setting the "initial file size" bit - bit 5 - of the modes byte). However, RBF allocates a first segment to the directory whose size is one sector less than the minimum segment allocation size, in anticipation of many small extensions to the directory (as files are created in the directory). Similarly, if the directory later requires additional disk space for a new entry, the new segment size is determined by the minimum segment allocation size (**PD_SAS** in the path descriptor) - the extra sectors above the actual directory file size are not de-allocated.

13.9 DELETE

This call is normally only implemented by file managers that support a filing structure, such as RBF. It could be used for other purposes - for example, to delete a window in a window management system. It is essentially the converse of the Create call. File managers that do not support this call will normally return an error - SCF returns **E$BMODE** ("bad mode").

The kernel's behaviour is almost identical to its Make Directory handler, except that it uses the modes byte without modification, and copies the modes byte to simulate a permissions byte for the purposes of initializing the path descriptor. Again, the execute bit of the modes byte is used to determine whether the pathlist is relative to the current execution or data directory. The kernel calls the Delete routine of the file manager, and then de-allocates the path descriptor. The file manager's Delete routine must locate and delete the file.

RBF finds the file in the same way as in the Open call, except that it also checks that the calling program has write permission for the file. RBF then trims the file size back to zero, in a similar way to the Set Status call "set file size". This de-allocates all of the clusters allocated to the file, except the first cluster, which contains the File Descriptor sector. Finally, RBF de-allocates this last cluster, and marks the file as deleted in the parent directory, by overwriting the first character of the name with a byte of zero.

Normally, if the file size is decreased, resulting in a segment table entry no longer being required, the "number of sectors" field of the entry is set to zero. From OS-9 version 2.3 onwards, RBF does not clear the segment list in the File Descriptor sector to zeros when trimming the file in the Delete routine. This allows a file that has just been deleted to be recovered, provided no files have been created or extended in the meantime. The only information not available is the first character of the file name in the directory entry. Such an "undelete" utility is not provided as standard with OS-9.

RBF will not allow a directory to be deleted. The "set attributes" Set Status call (**SS_Attr**) must be used to remove the directory attribute of the file before deleting a directory file. RBF will only allow this to be done if all the entries in the directory have been deleted (other than the "self" and "parent" entries) - that is, the directory is empty. The **deldir** utility performs all three operations. It deletes all the files in a directory, removes the directory attribute from the directory, then deletes the file.

13.10 SEEK

This call requests a repositioning of the file pointer for subsequent reading or writing. It is therefore normally only implemented by file managers that support random access of data, such as RBF. It could be used for other purposes, such as setting the cursor position in a graphics display.

SCF does not support this call - it does nothing, but returns no error.

RBF allows any file pointer value to be set. The device is not accessed as part of the seek operation. If the file pointer is past the current end of the file, a subsequent Read or Read Line call will return an "end of file" error (**E$EOF**), while a subsequent Write or Write Line call will cause the file to be extended. The file is extended to the length given by the file pointer set by the Seek call, plus the length of the Write or Write Line call. This leaves a portion of the file unwritten - it will contain whatever data was previously in the allocated sectors.

13.11 READ AND WRITE

These are the "raw" data transfer calls, to get data from or send data to a device. In the general philosophy of OS-9 I/O, these routines transfer the requested number of bytes unless an error occurs, or the end of the file is reached when reading. Other input termination conditions are file manager dependent. The file manager may also implement some processing of the data. However, the philosophy of these calls is in contrast to the Read Line and Write Line calls, so the data processing is usually minimal.

13.11.1 RBF

RBF performs no data processing. The number of characters transferred is always the number requested, unless a device driver error occurs, or the end of the file is reached in a Read call, or the disk (or the file's segment list) is full during a Write call. If a Read call attempts to read past the end of the file, no error is returned provided one or more bytes were read. The file pointer is moved to the end of the file. A subsequent Read call - reading starting at the end of the file - is returned an "end of file" error (**E$EOF**). For both Read and Write calls, the file pointer is advanced by the number of bytes read or written.

When a Write call writes past the end of the file, RBF automatically extends the file to accommodate the new length. When extending a file, RBF checks whether the current last cluster in the file already has sufficient room to

extend the file. If not, RBF checks whether the last segment in the file has sufficient room (in case it was pre-extended by an earlier write). If not, RBF must allocate additional space for the file, as described above.

13.11.2 SCF

In the Read call, SCF echoes characters read (if echo is enabled in the path descriptor **PD_EKO** field) to the attached echo device (usually the same as the primary device). SCF terminates the call with an "end of file" error (**E$EOF**) if the end-of-file character in the path descriptor (**PD_EOF**) is not zero, and matches the first character read. SCF also terminates the input prematurely if an input character is not zero, and matches the end-of-record character in the path descriptor options section (**PD_EOR**), or if the device driver returns an error.

In the Write call, SCF implements its "page pause" feature. That is, if the "page pause" flag is set in the path descriptor (**PD_PAU** is not zero), SCF will pause before sending a Carriage Return character if the total number of Carriage Return characters sent since SCF last read a character on this path is equal to the "number of lines per page" field (**PD_PAG**). Therefore "page pause" must be turned off in the path descriptor (**PD_PAU** set to zero) if binary data is to be sent that may include Carriage Return characters (byte value $0D). Otherwise the output will be paused once the number of Carriage Return characters sent equals the "page length" field (**PD_PAG**), until a character is received from the device.

13.12 READ LINE AND WRITE LINE

The only strict difference in the OS-9 I/O philosophy in file manager operation between these routines and the Read and Write routines is that in addition to the reasons for termination of the Read and Write calls (given above), these routines terminate if a Carriage Return character is encountered. However, even this feature is a function of the file manager only. In addition, it is intended that file managers may implement more data editing in these calls than in the Read and Write calls.

Because the calls should terminate if a Carriage Return character is transferred, the actual number of characters transferred may be less than the number requested. Note that the Carriage Return character is transferred, and is included in the count of characters transferred. If no Carriage Return character is encountered, the transfer will terminate once

the requested number of characters has been transferred, just as with the Read and Write calls.

13.12.1 RBF

RBF behaves exactly as described above for the Read and Write calls (with no data processing), other than terminating the request when a Carriage Return character is read or written.

13.12.2 SCF

SCF behaves as described for the Read and Write calls, with some additional features. In the Write Line call, characters are converted to upper case if that option is enabled in the path descriptor (the **PD_UPC** field is not zero), and a Line Feed character (byte value $0A) is automatically output after the Carriage Return character (if any) if the **PD_ALF** field is not zero. The SCF end of line pause is implemented. That is, if the device driver sets the "pause" flag in the Device Static storage because the "pause" character was received (non-zero, and matching the value in **PD_PSC**), SCF pauses output before outputting the Carriage Return character, until a character (other than the "pause" character) is received.

SCF also implements its end of page pause feature (as described above), and tab expansion - tab characters are expanded to spaces. That is, if a non-zero character matching the **PD_Tab** field of the path descriptor (usually $09) is to be transmitted, SCF instead transmits space characters (byte value $20) to bring the total number of characters sent since the last Carriage Return up to an integral multiple of the tab length field (**PD_Tabs**).

In the Read Line call SCF terminates when the character read matches the "end of record" character (**PD_EOR**), rather than the Carriage Return character. Normally the "end of record" character is set to be the same as the Carriage Return character. Note that input does *not* terminate when the requested number of characters has been input. SCF continues to input characters until the "end of record" character is received, discarding any characters that exceed the number requested. Input also terminates if the device driver returns an error, or if the "end of file" character (**PD_EOF**) is received and is the first character in the buffer. (Note that other characters can be input, provided they are deleted before the "end of file" character is entered.)

Note also that the "end of record" character is echoed without converting it to a Carriage Return character, and that the "automatic line feed" feature of

SCF is implemented only if a Carriage Return character is output, not an "end of record" character.

Input is through the SCF editing buffer, and so is restricted to 512 bytes (256 prior to OS-9 version 2.3), including the "end of record" character. The SCF input line editing features are operative during a Read Line call. The following list shows the line editing keys, giving the path descriptor field containing the key character value, and the standard setting for that field.

PD_field	Standard	Action
PD_BSP	^H	Delete the character to the left of the cursor.
PD_DEL	^X	Delete all characters (delete line).
PD_RPR	^D	Redisplay all characters (reprint line). This is useful if the display has become corrupted, perhaps due to limitations of the display terminal.
PD_DUP	^A	Redisplay from the current position in the input buffer to the end of the line. This causes SCF to display characters from the input buffer, starting at the current cursor position, and stopping at the first "end of record" character encountered, or at the last character ever entered into the buffer. This allows an entered line to be repeated. It also allows a simple form of editing of a previous line, by typing in and so overwriting characters in the buffer, and then redisplaying the remainder of the buffer (and then perhaps backspacing over some characters, and overwriting those).

13.13 GET STATUS AND SET STATUS

These are "wild card" routines, and so their function is very much file manager dependent. Most file managers implement the **SS_Opt** function of the Set Status call, setting new path descriptor options. The file manager may restrict which fields of the path descriptor options section can be modified in this way.

File managers will normally pass all calls on to the driver, even if they have recognized and handled the call (unless the file manager itself generated an error). If the file manager has handled the call, and the driver returns an "unknown service request" error (**E$UnkSvc**), the file manager should ignore the error. This permits drivers to choose to act on calls already handled by the file manager (such as a change of the path descriptor options section), or reject the call by returning the **E$UnkSvc** error.

File managers may internally generate calls to the device driver. For example, by convention file managers make an **SS_Open** Set Status call

when openeing or creating a file, and an **SS_Close** Set Status call when the last image of a path is closed. Note that if an internally generated call might also be made from a program, the file manager will need to pass any parameters by putting them in the caller's register stack frame (saving and afterwards restoring what was there before!), because the device driver cannot know whether the call was from a program, or was internally generated by the file manager.

SCF recognizes no calls other than the **SS_Opt** Set Status call. RBF implements several calls. They are described in detail in the OS-9 Technical Manual, and are also briefly listed below:

☐ Get Status

Function	Description
SS_Ready	Test for data available (always true).
SS_Size	Get file size.
SS_Pos	Get current file pointer.
SS_EOF	Test for file pointer at end of file.
SS_FD	Read part or all of the File Descriptor sector of the file.
SS_FDInf	Read part or all of a File Descriptor sector, specifying the LSN of the File Descriptor sector (usually from reading the directory entry of a file). This permits a File Descriptor sector to be read without opening the file.

☐ Set Status

Function	Description
SS_Opt	Update the options section of the path descriptor from the caller's buffer.
SS_Size	Set a new size for the file - causes the file to be extended or truncated (trimmed).
SS_FD	Update the File Descriptor sector of the file from the caller's buffer. The segment list cannot be altered, and only two other fields can be altered: the owner's group and user ID (can only be altered by a super user), and the "last modified" date and time. Prior to OS-9 version 2.3, the date of creation could also be altered.
SS_Ticks	Set the maximum wait for a record to become unlocked. If the caller is subsequently put to sleep by RBF because it attempts to read a record on this path that is currently held by another process, RBF uses this as the parameter to the sleep call. A value of zero (the default) will cause an indefinite wait for the record to be released. A timeout while waiting causes RBF to return an **E$Lock** error to the caller.

Function	Description
SS_Lock	Request RBF to lock part or all of a file. This will lock a record without the need for the calling process to read the record. A further call to this function will release any previously locked record. Hence a call with a record length of zero removes any lock held by this process on the file.
SS_Attr	Change the attributes (permissions) of the file. The calling process must be a member of the same group as the owner of the file, or be a super user. The directory bit of a directory file can only be cleared if the directory has no entries still in use (other than "self" and "parent"). The directory bit of the root directory cannot be cleared.

13.14 CLOSE

The Close routine is essentially the converse of the Open and Create routines. The file manager is requested to ensure that any resources allocated for the management of the path are de-allocated, and that all information about the file (if a filing system is supported) is up to date on the medium. However, the Close system call (**I$Close**) is also the converse of the Duplicate Path system call (**I$Dup**). Therefore a Close request to the file manager may not cause the termination of a path, as there may be other duplications (or "images") still in existence.

Path images are counted in the path descriptor word field **PD_COUNT**. The kernel also maintains the byte field **PD_CNT**, but this is for historical compatibility only. The file manager should terminate the path and de-allocate associated resources only when the Close routine is called with the **PD_COUNT** field at zero. When this occurs, a file manager will also typically generate a **SS_Close** Set Status call to the device driver.

13.15 CALLING THE DEVICE DRIVER

The file manager calls the device driver to carry out physical device operations, and to pass on Get Status and Set Status calls. The section on Device Drivers gives details of the device driver routines, and the calling conventions used by SCF and RBF. Note that the file manager must not call the Initialization and Termination routines of the device driver - these are called only by the kernel.

A device driver is a separate OS-9 memory module, and the file manager writer cannot know the address of the device driver at compile time. To call a device driver routine the file manager must calculate the routine address, using the device driver module address in the device table entry, and the

offset to the required routine (from the routine offset table in the device driver). The offsets within the table have been symbolically defined by Microware in the file 'DEFS/sysio.a'.

For example, to call the driver Read routine:

```
movea.l PD_DEV(a1),a0    get device table entry address
movea.l V$STAT(a0),a2    get device static storage
movea.l V$DRVR(a0),a0    get driver module address
move.l  M$Exec(a0),d0    get offset to routine offset table
move.w  D$READ(a0,d0.l),d0  get offset to read routine
jsr     0(a0,d0.w)       call the routine
```

The file manager will normally save the processor registers that it wishes to have preserved before calling the device driver. This avoids the need for the device driver writer to know which registers the file manager wishes preserved.

13.16 RESOURCE CONTROL

Paths and devices are system resources. Catastrophic results could occur if two processes were allowed concurrent access - system memory structures could be corrupted, and device operations confused. In general this cannot occur, because process rescheduling does not occur while a process is executing in system state, so the system call can finish its operations without worrying that it may be scheduled out, and another process scheduled in which will want to use the same resource.

However, it is possible for an operating system routine to explicitly go to sleep (**F$Sleep** or **F$Event**). This is normal practice in interrupt-driven device drivers. In this case another process may become the current process, and may attempt access to the same system resource.

The kernel controls concurrent accesses to the same path, using the **PD_CPR** field of the path descriptor. If this field is non-zero when a process makes an I/O request on the path, the kernel assumes it to be the ID of a process currently executing an I/O operation on the path, and I/O queues the calling process on that process (**F$IOQu** system call). Otherwise it puts the ID of the calling process in the **PD_CPR** field, so holding the path for the calling process. On return from calling the file manager, the kernel releases the path by clearing the **PD_CPR** field, and waking up the first process in the queue of processes queued on the calling process (if any).

However, the kernel does not implement any control of concurrent accesses to devices, or to "channels" within devices. This function is left for the file

manager or device driver. The file manager writer must decide what control of concurrent accesses the file manager will provide, and the device driver must implement any remaining functionality. SCF and RBF use a common mechanism, which removes the need for the device drivers to perform any control over concurrent accesses. They use the **V_BUSY** field of the device static storage to prevent concurrent accesses to the whole device, in the same way the kernel uses **PD_CPR** to prevent concurrent accesses to the path descriptor.

When the file manager wishes to acquire the device (for example, before calling the device driver), it checks the **V_BUSY** field. If it is zero, the device is free - the file manager copies the process ID of the calling process to this field, and makes its call to the device driver. On return, the file manager clears the **V_BUSY** field, and wakes up the first process (if any) queued on the current process. If **V_BUSY** is not zero, however, the file manager assumes it is the process ID of a process currently making a call into the device driver, and I/O queues the current process onto the process that is holding the device (**F$IOQu** system call). This puts the current process to sleep. On wakeup, the file manager again tries to acquire the device, unless it decides that a fatal condition has occurred.

It is up to the file manager writer to decide when and how to acquire control of a device - it can be applied to the device as a whole, or to a "channel" on the device, or it can even be left entirely to the device driver. RBF and SCF behave somewhat differently. RBF acquires the device just before calling the driver, and releases it on return from the driver (unless it is manipulating the allocation bit map, in which case it hangs on to the device until it has finished the allocation bit map function). By contrast, SCF acquires the device (and the associated echo device, if any) at the start of the file manager routine, and does not release it (knowing that the kernel will perform an I/O unqueue operation when SCF returns to the kernel). This keeps text lines indivisible if multiple processes are writing to the same device.

Because only one field - **V_BUSY** - is used for device allocation, these file managers prevent concurrent read and write requests. This is why output cannot occur to the screen while a process is taking in input from the keyboard. This is not a fixed requirement, however. Provided the device driver can handle concurrent read and write operations, the file manager can allocate the device for read and write operations separately, using separate fields (defined by the file manager writer) in the device static storage. The same applies to the allocation of multiple channels within a device. In the most liberal case, the file manager will use separate allocation fields for read and write operations on each channel, leaving the device driver with the

responsibility to ensure that it does not cause conflicts of use of the device static storage, or of access to the physical device interface.

There is no simple system call to "I/O unqueue" a process waiting on the current process. The file manager must check the I/O queue fields of the current process's process descriptor, unlink the current process from the queue, and wake up the first t process in the queue. If there were multiple processes in the original I/O queue, the remainder of the queue is now rooted in the process descriptor for the process that has been woken:

```
            move.w   P$IOQN(a4),d0    get ID of process queued on this
            beq.s    Done             ..none; no action
            clr.w    P$IOQN(a4)       clear the "next" link
            moveq    #S$Wake,d1       wake up the queued process
            os9      F$Send           by sending the wakeup signal
            os9      F$GProcP         get proc desc ptr of queued process
            clr.w    P$IOQP(a1)       clear the "previous" link
Done
```

However, as described for SCF above, the file manager does not need to perform an "I/O unqueue" operation when it has finished with the device or channel, because the kernel always performs such an operation after calling the file manager. Therefore the kernel will wake up the first process queued on the current process, whether the reason for queuing was because the process wanted to use the same path, or the same device. RBF performs an unqueue operation as soon as its call into the device driver has finished only because it aims to allow fair usage of the device by multiple processes concurrently.

Note that SCF does not use this device acquisition technique in a Get Status call. Therefore the **V_BUSY** field is not set, and if the device driver sleeps within the Get Status call, SCF may call any of the driver's routines as the result of another I/O call (from another process on another path to the same device).

13.17 A SKELETON FILE MANAGER

As with device drivers, part of the problem in writing a file manager is knowing where to start. This section shows a skeleton file manager in 68000 assembly language. The chapter on "Microware C and Assembly Language" shows how this can be adapted to form the core of a file manager written in C.

```
* Skeleton file manager
Typ_Lang    set     (FlMgr<<8)+Objct    module type and language
Att_Revs    set     ((ReEnt+SupStat)<<8)+0  module attributes and
*                                           revision
Edition     set     1               software edition number
            psect   skelmgr,Typ_Lang,Att_Revs,Edition,0,EntryTable
            use     /dd/DEFS/oskdefs.d

* Routine offset table:
EntryTable  dc.w    Create-EntryTable   create
            dc.w    Open-EntryTable     open
            dc.w    MakDir-EntryTable   make directory
            dc.w    ChgDir-EntryTable   change directory
            dc.w    Delete-EntryTable   delete
            dc.w    Seek-EntryTable     seek
            dc.w    Read-EntryTable     read
            dc.w    Write-EntryTable    write
            dc.w    ReadLn-EntryTable   read line
            dc.w    WriteLn-EntryTable  write line
            dc.w    GetStat-EntryTable  get status
            dc.w    SetStat-EntryTable  set status
            dc.w    Close-EntryTable    close

* Create
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Create      movea.l R$a0(a5),a0     get ptr to pathlist
            move.w  #E$BMode,d1     error - can't create
            ori     #Carry,ccr
            rts

* Open
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Open        movea.l R$a0(a5),a0     get ptr to pathlist
            move.w  #E$BMode,d1     error - can't open
            ori     #Carry,ccr
            rts
```

```
* Make directory
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
MakDir      movea.l R$a0(a5),a0     get ptr to pathlist
            move.w  #E$BMode,d1     error - can't make directory
            ori     #Carry,ccr
            rts

* Change directory
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
ChgDir      movea.l R$a0(a5),a0     get ptr to pathlist
            move.w  #E$BMode,d1     error - can't change directory
            ori     #Carry,ccr
            rts

* Delete
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Delete      movea.l R$a0(a5),a0     get ptr to pathlist
            move.w  #E$BMode,d1     error - can't delete
            ori     #Carry,ccr
            rts

* Seek
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr NOT (a5/a6)
*
Seek        move.l  R$d1(a5),d0     get desired position
            rts                     no action - carry is clear
```

```
* Read
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Read        move.l  R$d1(a5),d0     get (max) number of bytes to read
            movea.l R$a0(a5),a0     get ptr to buffer
            clr.l   R$d1(a5)        no bytes read
            rts                     no action - carry is clear

* Write
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Write       move.l  R$d1(a5),d0     get (max) number of bytes to write
            movea.l R$a0(a5),a0     get ptr to buffer
            clr.l   R$d1(a5)        no bytes written
            rts                     no action - carry is clear

* Read line
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
ReadLn      move.l  R$d1(a5),d0     get (max) number of bytes to read
            movea.l R$a0(a5),a0     get ptr to buffer
            clr.l   R$d1(a5)        no bytes read
            rts                     no action - carry is clear

* Write line
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
WriteLn     move.l  R$d1(a5),d0     get (max) number of bytes to write
            movea.l R$a0(a5),a0     get ptr to buffer
            clr.l   R$d1(a5)        no bytes written
            rts                     no action - carry is clear
```

```
* Get status
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
GetStat     move.w  R$d1+2(a5),d0   get function code
            move.w  E$UnkSvc,d1     unknown function
            ori     #Carry,ccr      show error
            rts

* Set status
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
SetStat     move.w  R$d1+2(a5),d0   get function code
            move.w  E$UnkSvc,d1     unknown function
            ori     #Carry,ccr      show error
            rts

* Close
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a4,ccr (NOT a5/a6)
*
Close       tst.w   PD_COUNT(a1)    last duplication is closing?
            bne.s   Close10         ..no
            nop                     no action
Close10     rts                     no error - carry is clear

            ends                    end of code
```