CHAPTER 12

DEVICE DRIVERS

This section is intended to dispel the mystery surrounding device drivers. It explains the purpose of a device driver within OS-9, describes the operating system environment the device driver works in, and shows typical algorithms for particular device types. Particular attention is given to the use of interrupts, as interrupt-driven devices are central to the proper functioning of a multi-tasking computer (under any operating system).

The descriptions and code fragments in this section assume that the device driver is written in 68000 assembly language. However, device drivers can equally well be written in C. The section on Microware C and Assembly Language describes how this is done.

12.1 THE FUNCTION OF A DEVICE DRIVER

A device driver is one part of an OS-9 I/O sub-system. All I/O system calls go initially to the kernel. However, the kernel has no understanding of the filing structure of a device, or of the hardware used to control the device. The job of the kernel is simply to set up the software environment, such as allocating a path descriptor, or locating an existing path descriptor if the path is already open. The kernel must then call the file manager or the device driver to carry out operations on the device. Similarly, the file manager understands the data structure of the device, but it does not know how to handle the hardware. If the file manager wishes to perform device operations - such as a data transfer - it must call the device driver.

In summary, the OS-9 I/O sub-system philosophy is to split the I/O operations as follows:

- The kernel allocates and de-allocates path descriptors, device table entries, and device static storages.
- The file manager handles the filing structure and any data editing.
- The device driver carries out physical device operations.

The device driver is therefore generally only concerned with performing low level physical device operations, without any understanding of why these are be carried out. However, this is not a strict requirement of OS-9. Device drivers may perform data interpretation, or other higher-level functions. For example, serial port drivers generally recognize certain special characters, such as the abort and interrupt keys. This is necessary because these keys must be acted upon as soon as they are received - that is, within the interrupt service routine. The file manager would only "see" the keys when a process subsequently performs a read request, which may be much later (or never!).

Similarly, a serial port driver for communications work might also incorporate a communications protocol because a received packet must be acknowledged within a very short time of reception, or because the file manager being used does not understand the protocol. It is sufficient that the combination of the file manager and the device driver provides all the data manipulation and hardware control functions.

The I/O system is normally a simple tree structure. There is one kernel, which can call multiple file managers. Each file manager can call multiple device drivers. Although it is conceivable that different file managers could call the same driver, in practice this is rarely done, because each file manager will normally have a different calling convention, and different structures for the path descriptor and device static storage. The **RBF** and **PCF** file managers are an example of this technique. **PCF** carefully uses the same path descriptor and device static storage structures as have been defined by the writer of **RBF**, so that **PCF** can implement an MS-DOS filing system using existing **RBF** device drivers.

12.2 DEVICE STATIC STORAGE

When considering the operation of an I/O sub-system within a multi-tasking operating system it is important to distinguish between "logical" paths and "physical" devices. A path is an operating system construct to enable a program to make system calls for data transfer and control on a device. The operating system can create more paths (given enough memory), up to some large limit (65535 under OS-9). A device is (normally) a construct to allow

the operating system to control a physical hardware object, such as a disk drive or a serial port. The operating system can "create" new devices, but usually only on a one-to-one correspondence with the hardware objects. Multiple paths may be open on a single device, but a path cannot be open on multiple devices. Note that this abstracted description relates to the kernel's view of devices. At a lower level (for example, in a device driver), the definition may become blurred. For example, a pipe does not relate to any hardware object - it uses a memory buffer only. Also, a device driver could use multiple I/O chips to provide one "device" with a complex function.

Because at any one time a device may have multiple paths open to it, or no paths, the path descriptor is not a suitable place to store variables for the control of the device. There must be one data structure in memory for each device. This is the purpose of the device static storage. Thus, the path descriptor is used to control the logical path, while the device static storage is used to control the physical device. The device driver is usually mainly concerned with the device static storage, as the device driver has the job of controlling the device. However, it may make use of some of the fields in the path descriptor, as these may provide information about the current configuration required for this particular I/O call.

Conversely, the file manager mainly makes use of the path descriptor for variables storage, as the file manager has the job of managing the logical path - for example, input line editing for a "read line" system call. But it may make use of some device static storage fields. For example, RBF maintains some disk structure information in the device static storage, as it is needed by all path-related functions using the device.

The kernel is responsible for allocating and de-allocating the device static storage for an I/O sub-system. The kernel allocates a new device static storage when a new device descriptor is installed in the device table (by the **I$Attach** system call, made implicitly when a path is opened) if either:

- The port address in the device descriptor is different from any other in the device table.

Or:

- Another device table entry has the same port address in its device descriptor, but a different device driver.

That is, the kernel considers this to be a new device if there is no device already in the device table using the same port address and device driver. Thus the kernel ensures that there is a separate device static storage allocated for each device currently in existence in the device table. Note that

by specifying a different device driver or by adjusting the port address in the device descriptor the programmer can coerce the kernel into allocating a separate device static storage for what may in fact by the same I/O interface. This is used in the SCSI driver system, where multiple drivers use the same I/O interface, and in the drivers for dual serial port chips, where separate "incarnations" of the driver must be created for each channel of the chip (because the SCF file manager does not support multi-channel drivers).

Conversely, the kernel permits multiple device table entries (for different device descriptors) that refer to the same device (they use the same device driver and port address). These "alias" device descriptors may be used to manage separate "channels" on the same device - such as multiple floppy disk drives attached to one controller - or to select different configurations for the device.

The device static storage comprises three or four parts. The first part is defined by the kernel - its size and usage is the same for all devices. Note that some fields are not used by the kernel - they have been defined for the convenience of the file manager and device driver writers, because they are required by many file managers and device drivers. This is an example of how Microware has provided a comfortable environment for the device driver writer, to try and limit the extent to which the programmer must learn about the operating system before writing a device driver. (Nonetheless, it is strongly recommended that you learn as much as possible about the operating system before writing a device driver.)

Following the kernel section, the file manager may (and usually does) require its own storage. The size and usage depends on the file manager, and is defined by the file manager writer. This part will be the same for all devices controlled through the same file manager. If the file manager supports multi-channel devices (for example, four disk drives attached to one interface) it will also usually require an area of storage for each channel, known as a drive or channel table. Therefore the third part of the device static storage - which exists only if the file manager supports multi-channel devices - contains the drive tables. This is simply an array of structures (drive tables), one for each channel, usually indexed by a logical drive or channel number (base zero). Typically the file manager will be capable of supporting a large number of channels, but on any given device the actual number will be far fewer. Therefore to conserve memory the size of the array is determined by the device driver (which knows how many channels the hardware will support) rather than the file manager.

The last part (highest memory address) of the device static storage is the storage area required by the device driver. Its size and usage is determined by the device driver writer. It effectively comprises the static variables of the device driver. It is defined using the normal statements for static variable definition in the source files of the device driver, whereas the kernel and file manager parts are defined in separate source files, as described below. The total size of the device static storage for the kernel to allocate is taken from the memory size entry in the device driver module header (**M$Mem**). It is the sum of the kernel, file manager, and device driver requirements. The size is in the device driver module header because the device driver is at the bottom of the tree, so the total size is only known when the device driver is created. It is calculated by the linker when creating the device driver module.

To simplify this operation, Microware provides pre-prepared definitions of the kernel and file manager storage, already assembled, to give the static storage (**vsect**) definitions required to reserve this storage. They need only be included at link time when creating the device driver. For example, when linking an SCF device driver:

```
$ 168 ../LIB/scfstat.l RELS/sc6850.r -l=../LIB/sys.l
-O=OBJS/sc6850
```

and when linking an RBF device driver for a device supporting two drives:

```
$ 168 ../LIB/drvs2.l RELS/rb1772.r -l=../LIB/sys.l
-O=OBJS/rb1772
```

Note that 'drvs2.l' is simply a merging of three ROFs:

```
$ chd /dd/LIB; merge rbfstat.r drvstat.r drvstat.r
>drvs2.l
```

'drvs2.l' reserves storage for the kernel and RBF, and for two RBF drive tables (as this device driver supports two drives on one interface).

The file 'LIB/scfstat.l' is an assembly of the file 'DEFS/scfstat.a', and is used for device drivers that work with the SCF file manager. The file 'LIB/rbfstat.r' is an assembly of the file 'DEFS/rbfstat.a', and is used for device drivers that work with the RBF file manager. It defines the storage required by the kernel and RBF. An RBF device driver must also reserve device static storage for the RBF drive tables, using the file 'LIB/drvstat.r' (which is an assembly of the file 'DEFS/drvstat.a') once for each drive to be supported. The files 'LIB/drvs1.l', 'LIB/drvs2.l', and 'LIB/drvs4.l' contain 'LIB/rbfstat.r' followed by one, two, or four copies (respectively) of 'LIB/drvstat.r', for device drivers supporting one, two, or four drives. By merging 'LIB/rbfstat.r' and multiple copies of 'LIB/drvstat.r' you can create versions for any number of disk drives.

Note that the copy of 'LIB/rbfstat.r' must come first in the "merge", as the definitions for the kernel and file manager storage must precede the drive tables. Similarly, in the linker command line the appropriate static storage definition file (such as 'LIB/scfstat.l') must precede the driver ROF (or ROFs), as all of the kernel and file manager storage must precede the driver storage in the device static storage memory. The linker builds the final static storage definitions from the static storage definitions in the ROFs strictly in the order that the ROFs appear on the linker command line. This ensures that kernel references into the device static storage are correct, even though the kernel does not know the structure (or even the existence) of the file manager and driver parts of the device static storage. Similarly, file manager references are correct, even though the file manager does not know the structure of the driver part.

As described above, the source files for these storage definitions are in the 'DEFS' directory (typically '/dd/DEFS'), as is a "make" file ('DEFS/makefile') to assemble and merge them. The equivalent files for the SBF file manager are 'LIB/sbfstat.r' (for the kernel and file manager definitions), and 'LIB/sbfdrvtb.r' (for one drive table), although full source code is not provided with OS-9. The SBF static storage structures are fully defined in 'DEFS/sbfdev.d', and in the files listed in Appendix B.

After allocating a new device static storage the kernel clears it to zeros. This is a very convenient software flag mechanism. If the initialization routine of the device driver aborts due to an error, the termination routine is always called by the kernel. The termination routine can "clear up" what resource allocation or device initialization the initialization routine managed to do before aborting, by looking to see if initialization flags are non-zero. For example, a field might be used to store the address of an allocated memory buffer, and will only be non-zero if the memory was actually allocated. However, the kernel does not support full C-like static storage initialization of the device static storage.

The device driver is not restricted to using only the device static storage for its variables and buffers. It can allocate additional memory as required using any of the available memory allocation mechanisms, including the general memory allocation system call (**F$SRqMem**), coloured memory, and data modules. Unlike the management of a program's memory allocations, the kernel does not keep track of memory allocated by a device driver (or any operating system component). It is the responsibility of the termination routine of the device driver to ensure that all such memory is de-allocated. The same philosophy applies to other resources that the device driver may allocate, such as creating an event or opening a path.

12.3 PATH DESCRIPTOR

A path descriptor is a memory structure (always 256 bytes) used by OS-9 to manage a path. Because the path concept is concerned with the logical manipulation of data (data editing, filing structure, and so on), the path descriptor variables in the first 128 bytes are mainly used by the kernel and the file manager, not by the device driver. The second half of the path descriptor is the "options section". It contains a copy of the options section of the device descriptor on which the path was opened. The layout of the options section is defined by the file manager writer, although the device driver writer may define additional fields in the device descriptor (but outside of the options section proper), pointed to by the offset value in the **M$DevCon** field of the device descriptor extended module header.

The options section contains parameters used to select optional behaviour of the file manager and device driver. For example, an SCF options section contains all the line editing key codes, and other special characters, while an RBF options section contains disk format parameters. The file manager may also dynamically write additional fields at the end of the path descriptor options section that are not defined in the device descriptor options section. These fields are for the information of a program, which can read all 128 bytes of the path descriptor options section using a Get Status request with the function code **SS_Opt**. For example, RBF puts a copy of the file name in the **PD_NAME** field of the options section.

The options sections of the Microware file managers are described in the OS-9 Technical Manual. The following paragraphs do not repeat those descriptions. Instead, they attempt to clarify certain areas that have caused difficulty to users in the past.

12.3.1 RBF Path Descriptor

RBF is not concerned with the physical layout of the disk (cylinders, surfaces, physical sector numbering[14]). It uses a logical sector numbering convention in which sector 0 is the first sector on the disk, and the other sectors are numbered sequentially. Some controllers (such as SCSI controllers) use the same convention, so the device driver does not need to translate. Otherwise,

[14] Disk drive terminology is often confused. A disk drive will have one or more disks on the same spindle. Each disk has one or two data surfaces. Data is read and written in concentric rings on each surface, using a read-write head on each surface. Each ring on each surface is a track, while the rings on all surfaces at the same radius are known as a cylinder. Therefore the total number of tracks equals the number of cylinders multiplied by the number of surfaces. Within each track the data is subdivided into equal sectors, numbered from zero or one upwards on each track.

the device driver must use the fields in the options section of the path descriptor to convert the logical sector number (LSN) to a cylinder number, surface (or head) number, and sector number. The calculation can be somewhat complex. The following fields of the options section are relevant:

PD_DRV	Logical drive number. This is the number used (base zero) by RBF to index into the drive tables. It may also be used by the device driver as the physical drive number, if the drives are numbered sequentially from zero upwards (but see **PD_LUN**).
PD_CYL	Number of cylinders available for data (for LSN validity check, and partitioning).
PD_SID	Number of data surfaces (tracks per cylinder).
PD_SCT	Number of sectors on each track, except track zero (cylinder zero, surface zero).
PD_T0S	Number of sectors on track zero.
PD_TOffs	First physical cylinder (*not track*) to use. After calculating the cylinder number (base zero) from the LSN, this value must be added to the cylinder number to form the true physical cylinder to access. This feature is used to skip cylinder zero on the Microware Universal floppy disk format, as different controllers place different restrictions on the format used on track zero.
PD_SOffs	First sector number on each track (zero or one). After calculating the sector number (base zero) within the track from the LSN, this value must be added to the sector number to form the true sector number to access. This feature is used because certain disk formats number the sectors on a track from zero, while others number the sectors from one.
PD_LUN	Physical drive number. If the interface is connected to multiple controllers (as with SCSI), then this is the drive number on that controller. This field may be equal to **PD_DRV** if only one controller is supported by the interface, and the drives are numbered sequentially from zero. This is typically the case for a simple floppy disk controller.
PD_LSNOffs	Offset for logical sector numbers. The driver must add this value to the LSN supplied by RBF before using the LSN in a controller command, or converting it to physical parameters. This allows support for partitioning on a hard disk.
PD_TotCyls	Total physical cylinders on the disk (for formatting, and LSN validity check). This value is usually equal to **PD_CYL** plus **PD_TOffs** (or the sum of these for all partitions) plus any allowance for cylinders reserved for automatic defect handling by the disk controller.
PD_CtrlrID	Controller number. This field is only used if the interface can be connected to multiple controllers (as with SCSI).

In addition to the parameters described above, other format variables are specified in the path descriptor options section:

PD_TYP — Disk type flags. If bit 7 is set, the disk is a hard disk, otherwise it is a floppy disk. This bit is only of importance for controllers that support both hard and floppy disk drives. Prior to OS-9 version 2.4, bit 0 was set for an 8" disk, and reset for a 5¼" (or 3½") disk. An 8" disk requires a rotational rate of 360rpm, and a data rate (MFM) of 500kbps, while a 5¼" disk requires a rotational rate of 300rpm, and a data rate (MFM) of 250kbps. Note that as far as the floppy disk controller and device driver are concerned, there is no difference between a 5¼" disk and a 3½" disk. As more disk formats were developed this became restrictive, and from OS-9 version 2.4 onwards bit zero is not used. Bits 1:4 define the disk size:

Value	Disk size
1	8" disk
2	5¼" disk
3	3½" disk

while the rotational speed and data rate are defined in the new field **PD_Rate**. If bits 1 to 4 of **PD_TYP** are zero, the driver knows that the descriptor is from before OS-9 version 2.4, and so bit zero of **PD_TYP** is used, and **PD_Rate** is not defined.
Two further bits are defined. If bit 5 is set, track zero (cylinder zero, surface zero) is double density, otherwise it is single density. This allows the support of old formats that have single density (FM) on track zero, and double density (MFM) on other tracks. Finally, for a hard disk (bit 7 is set), if bit 6 is set, the hard disk is removable.

PD_DNS — Data density flags – if bit 0 is set, the disk is double density (MFM encoding), otherwise it is single density (FM encoding). Single density is used only rarely today, in support of old formats on products using historical standards.

PD_Rate — This field is defined for OS-9 version 2.4 onwards. Bits 0:3 specify the rotational speed:

Value	Speed
0	300rpm (3½" or 5¼")
1	360rpm (8", or PC-AT 5¼")
2	600rpm

and bits 4:7 specify the data rate:

Value	Data rate
0	125kbps (3½" or 5¼" single density only)
1	250kbps (3½" or 5¼" double density, or 8" single density)
2	300kbps (ditto, but rotating at 360rpm)
3	500kbps (high density, or 8" double density)
4	1000kbps
5	2000kbps
6	5000kbps

Combinations of these fields allow the support of all commonly used floppy disk formats. However, not all controllers and disk drives will support all rotational speeds and data rates.

PD_SSize — Number of bytes per sector. This is the block size RBF will assume when requesting data transfers. Prior to OS-9 version 2.4 only a value of 256 was permitted. From OS-9 version 2.4 onwards any value that is a power of 2, from 256 to 32768, is permitted. Also, when opening a file, and before

performing any data transfers or allocating any data buffers, RBF will make a Get Status call to the driver with the function code **SS_VarSect**. This gives the driver the opportunity to check or alter the value in **PD_SSize** to correspond to the medium in use. For example, a device descriptor for a SCSI hard disk drive may have zero in this field. The device driver updates the field with the actual disk sector size returned from the drive controller in response to the SCSI READ CAPACITY command.
If the driver returns no error in response to the **SS_VarSect** call, RBF assumes the value in **PD_SSize** is the correct sector size to use. If the driver returns the error **E$UnkSvc** (unknown request), RBF assumes a default sector size of 256 bytes. (Any other error code will cause RBF to abort the opening of the file with an error.) Having determined the sector size, RBF writes it to the path descriptor options section field **PD_SctSiz** (a long word). It is this value that RBF uses for subsequent operations, and it can be read by a program, using the Get Status request **SS_Opt** to return a copy of the options section.

Microware supports a range of floppy disk formats. Although the preferred distribution format is the Universal format (which does not use cylinder zero), this is a recent standard, and many OS-9 systems use other - older, or more conventional, or higher data density - formats. The user may therefore have a number of different "alias" device descriptors for the same floppy disk drive, specifying different format parameters. The Microware-defined format codes depend on the density of track zero - single density (FM) or double density (MFM) - and the number of the first sector on each track (sector offset). In addition, the Universal format (code 38U0) does not make use of cylinder zero - the cylinder offset is one. The commonly used formats are:

Format code	Track 0 density	Sector offset	Cylinder offset
3803	FM	0	0
3807	MFM	0	0
38W7	MFM	1	0
38U0	MFM	1	1

The format codes shown above are for 3½" disks. The initial "3" of the format code is changed to "5" for 5¼" disks. All of these formats use double density on all tracks (other than track zero for 3803 format), 80 cylinders, and 16 sectors per track (10 sectors on track zero for 3803 format).

A device driver written to support a wide range of formats will need to take account of all of the above parameters when initializing the disk controller, and when performing data transfers. There is the risk for a removable disk

(such as a floppy disk) that the user will insert a disk of a different format. In this case the user will normally access the disk using a different device descriptor, with the appropriate format parameters. The device driver must check at each transfer (or perhaps only when a path is opened - RBF makes the Set Status call **SS_Open** to the driver) whether the format parameters have changed, requiring a re-initialization of the disk controller.

One simple way of doing this is to keep a record (in the device static storage) of the address of the device descriptor last used to initialize the controller. The address of the device descriptor can be taken from the device table entry. The address of the device table entry for this device is in the path descriptor location **PD_DEV** (it is set up by the kernel), and the field **V$DESC** in the device table entry contains the address of the device descriptor. If the current device descriptor address is different from the one last used to initialize the controller, a re-initialization is required. This is not foolproof - the user might change the parameters in an already loaded device descriptor, or unlink a device descriptor and load a new one at the same address - but it is simple and effective. The alternative is to check each one of the format parameters against the values last used to initialize the controller.

Certain fields of the options section are used to control the behaviour of RBF and the device driver:

PD_VFY	Disable verify after write. If this field is non-zero the device driver should not perform a verify (read) of each sector after writing it. Verify-after-write is only used with disk structures that do not support error detection and correction - usually floppy disks. Other device drivers (for example, for SCSI hard disk drives) will ignore this field.
PD_SAS	Minimum segment allocation size. When RBF is asked to extend a file (for example, by a write at the end of the file), if the extension is shorter than this value (in sectors) RBF will allocate this many sectors to the file. When the file is closed, RBF trims back the file to its true length (provided the file pointer is at the end of the file). This reduces the fragmentation problem caused by two files "leap-frogging" each other as they are written to.
PD_Cntl	A word of bit flags, having the following effects when set: 0 - enable formatting and writing to sector zero. If this bit is not set, the driver should return an error **E$Format** if it is requested to format the disk, or to write to sector zero. Hard disk device descriptors are usually format protected in this way, with a special device descriptor being loaded in order to format the disk, or set a new boot file (the **os9gen** utility writes the address of the boot file to sector zero). 1 - enable multi-sector transfers. If this bit is not set, RBF will only request the device driver to transfer one sector with each request. This bit should be reset for controllers that can only transfer one sector at a time.

3 - the device driver can determine the disk capacity. If this bit is set, the device driver supports the **SS_DSize** Get Status call, returning the disk capacity in sectors.

PD_MaxCnt — Maximum transfer size. RBF will not ask the device driver to transfer more than this number of bytes in one request. This may be set to a limit imposed by a DMA controller, for example. If the controller cannot transfer more than a certain number of sectors in one request, either this field must be set to that number of sectors multiplied by the sector size, or the device driver must be able to divide up a large request into manageable pieces.

The RBF path descriptor variables section (the first 128 bytes of the path descriptor) contains three fields of interest to the device driver writer:

PD_DEV — Address of the device table entry for the device on which the path was opened. The device driver can use this pointer to get the address of the device descriptor (field **V$DESC** in the device table entry).

PD_DTB — Address of the drive table. RBF multiplies the logical drive number (**PD_DRV**) by the size of one drive table, and adds it to the address of the first drive table in the device static storage, to form this address. The device driver must copy the first 22 bytes of LSN zero to the drive table at this address whenever LSN zero is read or written.

PD_BUF — Buffer address. When RBF calls the read or write routines of the device driver, this field contains the address of the memory to read to or write from.

12.3.2 SCF Path Descriptor

The SCF path descriptor options section is mainly composed of special key codes for line editing and keyboard signals. If a key code is set to zero, an incoming character is not checked against the key code. This permits line editing functions to be disabled. Note that the **xmode** utility allows the user to modify the options section of an SCF device descriptor in memory, while the tmode utility modifies the path descriptor options section for path 0, 1, or 2 (standard input, standard output, and standard error) of its inherited paths. Certain other options fields modify the line editing behaviour of SCF. Of these, the most commonly used are:

PD_EKO — Enable echo. If this field is non-zero, SCF echoes each character as it is read during a "read" (**I$Read**) or "read line" (**I$ReadLn**) request.

PD_ALF — Automatic line feed. If this field is non-zero, SCF outputs a line feed character ($0A) after each carriage return character ($0D) in a "write line" (**I$WritLn**) request.

PD_PAU — End of page pause. SCF counts carriage return characters ($0D) as they are written by "write" and "write line" requests. It resets the count if any "read" or "read line" request is made. If this field is not zero, when the count

reaches the value in **PD_PAG** SCF does not output the carriage return character until a character has been received. Note that this applies even to characters written by "write" requests, so it is important to clear this field when sending binary data.

PD_EOR — End of record character (usually [CR]). If this field is non-zero, SCF compares every incoming character with this field, and terminates a "read" or "read line" request when a character matches this field. Note that if echo is enabled (see **PD_EKO**) the carriage return character ($0D) is echoed in response to this character being received.

PD_EOF — End of file character (usually [ESC]). If this field is non-zero, and a matching character is read as the first character of a "read" or "read line" request, SCF aborts the request with an "end of file" error (**E$EOF**). For a "read line" request, the end of file condition is reported if the first character in the edit buffer matches this field, even if other characters have previously been entered and then erased.

PD_PSC — End of line pause key code (usually [^W]). SCF copies this field to the **V_PCHR** field of the device static storage. The "data received" interrupt service routine of the device driver should compare each incoming character with the **V_PCHR** field (if non-zero). If a match is found, the device driver sets the **V_PAUS** field of the device static storage. When SCF is about to write a carriage return character from a "write" or "write line" request, it checks the **V_PAUS** field. If non-zero, SCF waits for a character to be received (other than a match for **PD_PSC**) before outputting the carriage return.

PD_INT — Interrupt key code (usually [^C]). SCF copies this field to the **V_INTR** field of the device static storage. The device driver "data received" interrupt service routine should check each incoming character against this field (if non-zero). If a match is found, the driver sends an interrupt signal (3 - **S$Intrpt**) to the last process to use the device. The process ID of this last process is copied by the kernel to the field **V_LPRC** in the device static storage. (**V_LPRC** is set to zero by SCF if the process has died, and it has no parent. If it has a parent with a path open on the same device, the parent's ID is copied to **V_LPRC**). When SCF sees this character as part of a "read line" request, it acts as if the "delete line" key code had been received.

PD_QUT — Abort (or quit) key code (usually [^E]). Similar to **PD_INT**, SCF copies this field to the **V_QUIT** field of the device static storage, and the device driver sends an abort signal (2 - **S$Abort**) if a matching character is received.

Two fields - **PD_XON** and **PD_XOFF** - are used for software flow control. They usually have values $11 and $13 respectively ([^Q] and [^S]) - the ASCII XON and XOFF characters. SCF copies these fields to the **V_XON** and **V_XOFF** fields of the device static storage, and the device driver uses these values (if non-zero) as the character codes to restart and stop transmission (respectively) in both directions. That is, if an XOFF character is received, the driver suspends transmission until an XON character is received. Conversely, if the driver's receive buffer is becoming full (it has reached a "high water mark"), it sends an XOFF character, and then sends an XON

character when the buffer has emptied by some pre-determined amount (it is reduced to a "low water mark").

Lastly, the device driver uses two fields to determine the desired configuration of a serial port (if that is what is being controlled):

PD_PAR — Character format. The bits of this field are used as follows:

0:1	parity generated and expected:
0	none
1	even
3	odd

2:3	bits per character:
0	8
1	7
2	6
3	5

4:5	number of stop bits:
0	1
1	1.5
2	2

Not all devices will be able to support all character formats.

Typically a device driver will configure the device to automatically pause transmission if the CTS handshake input is negated (relying on the fact that in most circuit designs this input will float asserted if not connected), and will assert the RTS (or DTR) handshake output. The driver will also configure the device to generate an interrupt when the DCD handshake input changes state (relying on the fact that in most circuit designs this input will float securely to either the asserted or negated condition if not connected). However, some device drivers (not Microware's) also use bits 6 and 7 to control the use of the hardware handshake lines:

6	set to disable hardware handshake (RTS/CTS)
7	set to disable recognition of DCD input

PD_BAU — Baud rate code - see the table of baud rates below.

Code	Rate	Code	Rate
0	50	9	2000
1	75	10	2400
2	110	11	3600
3	134.5	12	4800
4	150	13	7200
5	300	14	9600
6	600	15	19200
7	1200	16	38400
8	1800	255	external

An "external" baud rate is set in hardware, and is not controllable by the device driver. Not all devices will be able to support all baud rates. Note that

no means is given of separately defining the baud rate for receive and transmit.

The device driver for a serial port will use these values to configure the interface during the driver's initialization routine. However, a program may wish to dynamically change the configuration on an open path. The program can modify these fields in the path descriptor options section using the Set Status system call with the **SS_Opt** function code (the **_ss_opt()** C library function). SCF will copy the new values to the path descriptor options section, and then pass the call on to the driver. On receiving this call the driver should check whether the device configuration fields in the path descriptor have changed since the last time the interface was initialized. If so, the driver should re-initialize the interface.

If the device driver is controlling an "intelligent" communications board, the board may also support the detection of the flow control and signal characters. For this type of device, the driver should also check to see whether these fields have changed. If any of the configuration fields have changed, the driver should re-initialize the board.

While this dynamic re-configuration capability of the device driver is desirable, early Microware example SCF device drivers did not provide this feature. As a result there are many SCF device drivers in existence that take no notice of changes to the device configuration fields of the path descriptor.

If a device descriptor or path descriptor options section specifies a device configuration that the device (or the device driver) does not support, the device driver should return a "bad mode" error (**E$BMode**).

12.4 SYMBOLIC DEFINITIONS

Microware have provided symbolic definitions in both C and assembly language for the structures and constants likely to be used by a device driver. These files are all in the 'DEFS' directory - it is strongly recommended that you study all of these files carefully before writing a device driver. The assembly language files are pre-assembled to make the library 'LIB/sys.l'.

Therefore a device driver written in assembly language does not need to pull in (**use** assembler directive) any of these files - the references are resolved at link time. Device drivers written in C will need to **#include** the relevant files. A typical list for an RBF device driver might be:

rbf.h	Path descriptor options section structure.
MACHINE/reg.h	Processor register definitions.
procid.h	Process descriptor structure.
path.h	Path descriptor variables section format.
module.h	Module header structures (including device descriptor).
errno.h	Error codes.
signal.h	Signal codes.
sg_codes.h	Set Status and Get Status function codes.

Note that the order of the **#include** statements for these files is important, as some files declare structures that are used in other files. The only operating system structure for which there is not a proper C definitions file is the System Globals. The file 'DEFS/setsys.h' does give the offsets within the System Globals structure to each field, but the System Globals is not defined as a structure, and the fields are not "typed".

In this book the symbolic names for the OS-9 error codes are sometimes given using the assembly language definitions in the file 'DEFS/funcs.a', and sometimes given using the C language definitions in the file 'DEFS/errno.h'. The symbol names are the same in both sets of definitions, except that the C definitions start with **E_** and use upper case only, while the assembly language definitions start with **E$** and use both upper and lower case. For example, the C symbol for the "not ready" error (code 246) is **E_NOTRDY**, while the assembly language symbol is **E$NotRdy**.

12.5 REGISTER USAGE

OS-9 was originally written completely in assembly language, although parts are now written in C. Therefore parameters are passed to and returned from the device driver in processor registers. In the following descriptions, as elsewhere in this book, parentheses around a register name mean "points to", and a suffix of ".b", ".w", or ".l" gives the size of the object in the register as byte, word (16 bits), or long (32 bits). Where the object is smaller than the register containing it, the object is always in the low order bits of the register, starting with bit zero.

Because the initialization and termination routines of the device driver are called directly by the kernel (as part of the **I$Attach** and **I$Detach** system calls), the calling convention to these routines is defined by the kernel, and is therefore the same for all drivers:

(a1) Device Descriptor module
(a2) Device Static Storage
(a4) Process Descriptor of calling process
(a6) System Globals

The calling conventions for the other routines (usually read, write, get status, and set status) are determined by the file manager, and may vary, especially for the read and write routines. Usually, however, the following conventions are adhered to:

(a1) Path Descriptor
(a2) Device Static Storage
(a4) Process Descriptor of calling process
(a5) Caller's register stack frame
(a6) System Globals

The other registers may contain other parameters. The return convention for read and write varies according to the file manager. The error return convention for all of the functions is (usually) the same as that used throughout the operating system: the carry flag of the Condition Codes register is set if there was an error, in which case **d1.w** contains the appropriate OS-9 error code.

The kernel saves all the registers it uses before calling the initialization and termination routines of the device driver, except for the **a6** register when calling the termination routine. Therefore the driver need only preserve the stack pointer and the high byte of the status register (and **a6** in the termination routine). In general, file managers also save all the processor registers they use before calling the device driver functions. This is true of SCF and RBF. However, other file managers may only save certain registers, in order to speed up calls to the device driver, so it is important to check the documentation on the file manager.

12.6 DEVICE DRIVER ROUTINES

A device driver is a separate OS-9 module, so the addresses of its routines are not known to the kernel and file manager. However, the kernel and file manager need to be able to call the device driver routines. To achieve this, the "execution entry offset" (**M$Exec**) in the module header of the device driver gives an offset from the start of the module to a table of offsets from the start of the module to each of the routines.

Two routines are absolutely required - initialization and termination - as these are called by the kernel when an I/O sub-system is created and deleted.

Any other routines are only called by the file manager, and their presence or absence is a matter for the file manager specification. File managers can specify any number of device driver routines for any purpose. Conventionally, however, the file manager requires four routines, making six in total. The code fragment below shows a typical **psect** statement and routine offset table for a device driver:

```
            use     /dd/DEFS/oskdefs.d
Typ_Lang    equ     (Drivr<<8)+Objct    module type and language
Att_Revs    equ     ((ReEnt+SupStat)<<8)+0  attributes and revision
*                                       number
Edition     equ     1                   software edition number
            psect   sc68681,Typ_Lang,Att_Revs,Edition,0,EntryTable

EntryTable  dc.w    Init                initialize
            dc.w    Read                input data
            dc.w    Write               output data
            dc.w    GetStat             wildcard call (I$GetStt)
            dc.w    SetStat             wildcard call (I$SetStt)
            dc.w    Term                terminate
            dc.w    0                   exception handler (see below)
```

Notice that the last parameter to the **psect** statement is the label of the routine offset table. It is from this statement that the linker takes the value to put in the "execution entry offset" field of the module header. Note also that the positions of the initialize and terminate routine offsets within the table are fixed, as these routines are called by the kernel. Therefore if the file manager does not require one or more of the read, write, get status, or set status routines these entries must still exist (replacing the routine label with zero), and if the file manager needs additional routines their offsets must be added to the end of the table shown above.

Microware have indicated that future versions of the kernel may implement an additional "exception handler" routine, using a seventh table entry as shown above. This entry point will be called if a hardware exception (such as bus error) occurs during driver execution.

12.6.1 Initialize

This routine is called directly by the kernel. It is only called when the I/O sub-system is being created - that is, a new device static storage has been allocated. It is *not* called on the first usage of each channel on a

multi-channel device. The kernel calls the initialization routine as part of the **I$Attach** system call. This call can either be made explicitly by a program (such as the **iniz** utility), or implicitly whenever a path is opened on the device. The initialization routine may be called more than once as the I/O sub-system is terminated and then re-created, but there will always be an intervening call to the termination routine as part of the termination of the I/O sub-system.

Some I/O devices must only be initialized once after reset - a repetition of the reset would cause problems. There is no operating system mechanism to determine whether this is the first time this I/O sub-system has been brought into being since reset. If it is important to know this, the initialization routine can use a data module. It attempts to create a data module whose name is constructed from the device port address. If there is no error, the data module did not already exist (otherwise a "known module" error **E_KWNMOD** would be returned), so this is the first time the I/O sub-system is being created.

This data module mechanism is also useful for sharing hardware with one or more other drivers. Common variables (such as the current state of write-only registers) can be held in the data module. Also, if the module does not already exist on initialization the driver knows it is the first user and must initialize the hardware. If the initialization and termination routines maintain a use count in the data module, the termination routine can know that it is the last user, and must terminate the hardware.

The initialization routine has a number of responsibilities. It must:

a) Initialize the device static storage as needed. Usually the driver only initializes its own fields of the device static storage, but it may initialize other fields to pass device information to the file manager. For example, an RBF driver sets the field **V_NDRV** to the number of drives supported (which must be no larger than the number of structures in the drive table), and the **DD_TOT** field of each drive table structure to a non-zero value (to permit RBF to read LSN zero).

b) Initialize the hardware, ready for subsequent calls to the other routines, such as read and write. A device driver expecting unrequested data to be received (such as an asynchronous serial port) must also set up the device ready for data to be received. For example, the interface chip would

be set up to generate interrupts when characters are received.

c) Install the interrupt service routine in the polling table (**F$IRQ** system call) if the device is to be interrupt driven. If the device driver is to receive interrupts on more than one vector, it will need to install multiple interrupt service routines. OS-9 places no limit on the number of interrupt service routines one driver can install.

The initialization routine is passed the address of the device descriptor module, not the address of the path descriptor (there may be no open path if an explicit **I$Attach** system call is being made). However, the Microware definitions in the 'LIB/sys.l' library only include definitions for the offsets into the options section of the path descriptor - there are no symbolic definitions for accessing the options section of the device descriptor. As the two options sections have (by definition) the same structure, the programmer can use the same symbols - with a constant offset - to access the options section of the device descriptor. For example:

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

will access the baud rate code in the device descriptor (assuming the **a1** register is pointing to the device descriptor), while:

```
move.b  PD_BAU(a1),d0
```

will access the baud rate code in the path descriptor (assuming the **a1** register is pointing to the path descriptor). This works because **M$DTyp** is the offset from the start of the device descriptor to the first entry in the options section, while **PD_OPT** is the offset from the start of the path descriptor to the start of the options section.

The device driver should not sleep as part of the initialization routine. The kernel does not build the device table entry until the initialization routine returns, so a concurrent I/O call from another process on the same device would cause a recursive call to the **I$Attach** system call and the initialization routine of the device driver. Also note that the **V_BUSY** field of the device static storage is not set to the process ID of the calling process at this time (see below), as this is a function of the file manager.

12.6.2 Terminate

The termination routine is essentially the converse of the initialization routine. It is called directly by the kernel as part of the dismantling of an I/O sub-system, from within the **I$Detach** system call. The kernel will only call the termination routine when the device use count (in the device table entry)

has been decremented to zero. That is, there are no paths open on the device, and any explicit calls to **I$Attach** (and **I$ChgDir**) have been complemented by an equal number of explicit calls to **I$Detach**. The kernel will always de-allocate the device static storage after calling the termination routine, and (in all versions of the kernel to date) ignores any error returned by the termination routine.

The termination routine must:

a) Wait for any "write-behind" activity to finish. For example, characters may be waiting in a buffer to be transmitted out of a serial port under interrupt, perhaps paused by software or hardware handshake.

b) Shut down the hardware. In particular, the hardware must be disabled from generating any interrupts or other autonomous behaviour.

c) De-allocate any resources allocated by the device driver. Examples are buffer memory allocated, data modules created or linked to, paths opened, and events created or linked to.

d) Remove the driver from the interrupt polling table, using the **F$IRQ** system call. Each interrupt service routine that the driver installed must be un-installed.

Note that if the initialization routine returns an error to the kernel, the **I$Attach** system call will call the termination routine before de-allocating the device static storage.

12.6.3 Read

As mentioned above, the read routine (if it exists) is only called by the file manager. Therefore the purpose of the routine and the parameter convention used when calling it are determined by the file manager writer. In general it is used to get data from the device. As an illustration, for this and the other routines the purpose and parameter convention are shown for the SCF and RBF file managers. These two file managers adhere to the general parameter convention described above, so only the additional parameters are described below.

☐ The Sequential Character File Manager (SCF)

Purpose: read one character.
Parameter convention:
Passed: nothing
Returns: **d0.b** = character read

SCF drivers usually maintain a circular input buffer in the device static storage (or dynamically allocated in the initialization routine) filled under interrupt. The interrupt service routine for the "data received" interrupt takes the character from the chip and puts it in the circular buffer. The driver read routine takes a character from this buffer, waiting (by sleeping) if the buffer is empty. The interrupt service routine is responsible for waking up the driver when a character is received, and for detecting and acting on certain special characters - flow control (XON and XOFF), "interrupt" (usually [^C]), "quit" (usually [^E]), and "end-of-line pause" (usually [^W]).

The "data received" interrupt service routine is also responsible for sending the "pause" flow control character (XOFF) when the buffer is becoming full - usually at a "high water mark" of three quarters full. Conversely, the read routine is responsible for sending the "restart" flow control character (XON) if a "pause" had been requested and the buffer is now sufficiently empty - usually at a "low water mark" of one quarter full.

Characters may be received with errors. For example, parity checking may be enabled for an asynchronous serial port, and a character may arrive with incorrect parity. As this error status is normally supplied by the interface chip with each character, the "data received" interrupt service routine must save the error status as it reads each character from the chip. The standard Microware drivers simply bitwise OR the error status of each character into the **V_ERR** field of the device static storage. The interrupt service routine also sets a bit in this field if the input buffer overflows, so one or more characters are lost. The read routine checks this field when returning a character - if it is not zero, the routine clears the field and returns a "read" error (**E$Read**).

Under this scheme the calling program is not able to determine which character was in error. This is not important when simply reading from a keyboard, but may be unsatisfactory for communications applications. A device driver for such an application might maintain a second circular buffer containing a status byte for each received character. When SCF requests a character for which the status is not zero, the driver returns a "read" error (**E$Read**), and saves the status in the device static storage field **V_ERR**. Such a driver could also support the Get Status call function **SS_ELog** (read

error log), returning a copy of the latest saved error status, permitting the program to determine the type of error.

If no characters are available in the input buffer, the read routine must sleep. It is then woken by the interrupt service routine when a character arrives. This is described in detail below in the discussion of interrupts. The read routine may also be woken from its sleep by a signal from another process (sent to the process that called the driver), or by a "quit" or "interrupt" signal sent to the process by the interrupt service routine on receipt of one of the special key codes. The read routine must decide whether to go back to sleep and wait for a character, or to abort the read with an error. A typical device driver for terminals and printers will abort only if the signal received was a "deadly" signal. Prior to OS-9 version 2.4 the deadly signals were signal 0 (the kill signal - **S$Kill**), signal 2 (the quit signal - S$**Abort**), and signal 3 (the interrupt signal - **S$Intrpt**). From OS-9 version 2.4 onwards all signals below 32 are considered deadly, except signal 1 (the wakeup signal - **S$Wake**).

The use of an input buffer filled under interrupt provides a "type ahead" capability. That is, provided the device is active (a path is open to the device, or the device has been explicitly initialized), characters can be received in advance of any read request from a program. This allows a user to type in a command in advance of the previous command completing. More importantly, it reduces the real-time response requirement of a program that is receiving data. Typically an SCF device driver has an input buffer of 80 characters. Thus a program can delay 80 character times (about 80ms at 9600 baud) before reading the data without losing any data.

There is no requirement under OS-9 that device drivers must be interrupt driven. The read routine of an SCF device driver could simply poll the status register of the interface chip until a character had been received, and then return that character to SCF. However, this would destroy the multi-tasking capability of the operating system, as rescheduling does not take place while a system call is executing - the system call must go to sleep or exit to allow a reschedule.

An alternative approach is to poll the status register and, if no character is available, to sleep for one tick. Sleeping for one tick requests a reschedule, but the process remains active. This allows another active process (if there is one, and it is of sufficient priority - see the chapter on Multi-tasking) to become the current process, otherwise the driver (or rather, the process calling the driver) remains the current process, and continues to poll the status register.

Clearly, the most efficient technique – both in terms of processor time usage, and of speed of response to a received character – is to use interrupts. However, the above description shows that OS-9 does not force any particular style of operation on the device driver.

☐ The Random Block File Manager (RBF)

Purpose: read one or more consecutive sectors from a disk.
Parameter convention:
Passed: **d0.l** = number of sectors
d2.l = starting Logical Sector Number (base 0)
PD_BUF(a1) = memory to read to
Returns: nothing

Note: prior to OS-9 version 2.4 the number of sectors was in **d0.b** only, and could not exceed 255. Now the number of sectors is only limited by the **PD_MaxCnt** field of the path descriptor, which sets a limit on the total number of bytes RBF may ask the driver to transfer. The same change applies to the write routine.

RBF is not concerned with the physical disk structure. It uses a Logical Sector Numbering scheme (base zero). The device driver must (if necessary) convert this to physical disk parameters, as described above in the section on the Path Descriptor.

RBF is also not concerned with retries. If there is an error on reading, it is up to the driver to decide whether to try again to read the sector (or sectors). If the driver returns an error to RBF, then RBF will consider it to be an unrecoverable error, and abort the filing operation, which may cause some damage to the disk structure. The disk controller may do retries itself, in which case the device driver will not itself implement any retries. SCSI hard and floppy disk controllers typically operate in this way. For simple floppy disk controllers the device driver may retry several times, occasionally restoring (seeking to cylinder zero) and re-seeking, in case the problem is a head misalignment.

On reading (or writing) LSN zero the driver also has the responsibility to copy the first 22 bytes into the first part of the appropriate drive table structure in the device static storage. (Note that RBF has pre-calculated the address of the drive table structure for this drive, and placed it in the path descriptor field **PD_DTB**.) This 22 byte structure contains information about the disk structure, both logical and physical (refer to the OS-9 Technical Manual section on the RBF Drive Table). A device driver may elect to use the physical format fields of this structure in place of some of the fields of the

path descriptor when calculating cylinder, surface, and sector numbers. This allows a driver to dynamically adapt to different disk formats (although of course it does assume that the driver can read LSN zero). The useful fields are:

DD_TOT	Total data sectors on disk (maximum LSN plus 1) (for LSN validity checking). RBF will not issue a request for an LSN greater than or equal to this value. Therefore the initialization routine of the device driver must set this field non-zero in each drive table structure, to allow RBF to read LSN zero.
DD_TKS	Sectors per track. (The field **DD_SPT** contains the same value as a word rather than a byte).
DD_FMT	Disk format flags. The bits have the following meanings when set: 0 double sided disk 1 double density disk 2 double track density disk

For example, if bit 1 of the path descriptor field **PD_DNS** is set, indicating that the drive is double track density, but bit 1 of **DD_FMT** read from LSN zero is not set, indicating a single track density disk, then the driver knows that it must instruct the drive controller to "double step", that is, to move the head by two steps for each cylinder number (because the drive supports cylinders twice as closely packed radially as the disk has on it). Similarly, even though the drive supports double sided floppy disks (**PD_SID** in the path descriptor is 2), if bit 0 of **DD_FMT** is not set the driver knows that the disk is single sided, and so adjusts its calculation of the physical parameters from the LSN.

12.6.4 Write

The write routine of the device driver is usually very much the complement of the read routine, the only difference being the direction of data transfer – the write routine (generally) sends data to the device. Much of the code for the read and write routines is shared in most device drivers. As in the read routine, the specific function of the write routine is defined by the file manager specification. The basic requirements of an SCF device driver and an RBF device driver are shown below as examples.

☐ The Sequential Character File Manager (SCF)

Purpose: write one character.
Parameter convention:
Passed: **d0.b** = character to write
Returns: nothing

SCF drivers usually maintain a circular output buffer in the device static storage (or dynamically allocated in the initialization routine) filled by the write routine, and emptied under interrupt. The interrupt service routine for the "transmitter ready" interrupt takes the character from the circular buffer and puts it in the chip. The driver write routine waits (by sleeping) if the buffer is full, and is woken by the interrupt service routine when the buffer has emptied a little.

Once the buffer has been completely emptied the interrupt service routine must disable further "transmitter ready" interrupts from the chip. Therefore the write routine must check whether the "transmitter ready" interrupts are disabled, and if so it must write the character directly to the chip (rather than putting it in the buffer), and enable the interrupts. This starts a stream of interrupts, each one being serviced by putting the next character into the chip. The stream is only stopped when the buffer becomes empty – the calling program has no more characters to send, or is supplying them at a rate below the transmission rate of the chip.

The "data received" interrupt service routine or the read routine may wish to send a flow control character (XON or XOFF), which must take precedence over any characters waiting in the output buffer. If "transmitter ready" interrupts are disabled, the flow control character is put directly in the chip, and the interrupts are enabled (starting a transmission stream that may be only one character long). Otherwise a flag is set in the device static storage. The flag is checked by the "transmitter ready" interrupt service routine at the next interrupt, and the flow control character is sent instead of taking the next character from the output buffer.

If the output buffer is full when SCF calls the write routine to send a character, the write routine must sleep. It is then woken by the interrupt service routine when space becomes available in the output buffer. Typically the interrupt service routine will not wake the write routine when just one space is available, but waits until the buffer has subsided to a low water mark, typically 10 characters left to send. This reduces the number of sleep/wakeup cycles, so reducing the processor load. This is described in detail below in the discussion of interrupts.

The use of an output buffer provides a "write behind" mode of operation. That is, provided there is sufficient space in the output buffer (typically 140 bytes in size), a program making a write request is returned to immediately, and continues with further operations while the data is transmitted at the rate permitted by the interface chip. This can be very important in preventing unacceptable delays – for example, when a real-time process prints an error or status message. However, under certain circumstances it may cause problems, as a program may need to know when a packet of data has completed transmission. In such applications the driver might be modified to provide an additional Set Status function to send a signal when the output buffer becomes empty.

The write routine may also be woken from its sleep by a signal from another process (sent to the process that called the driver), or by a "quit" or "interrupt" signal sent to the process by the "data received" interrupt service routine on receipt of one of the special key codes. The write routine must decide whether to go back to sleep and wait for a character, or to abort the write with an error. As with the read routine, a typical device driver for terminals and printers will abort if the signal received was a "deadly" signal.

SCF provides a read-write lockout. That is, even if the read routine goes to sleep (allowing another process to execute and make system calls), the write routine will not be called until the read routine has woken up and exited. A process making a system call that requires a write call to this device will be "I/O queued" until the read request finishes. The same mechanism applies if a read request is made while a write request is in progress. This mechanism is explained in detail in the section on Resource Control in the chapter on File Managers. It greatly simplifies the job of the device driver – which can use common device static storage locations for read and write calls – as well as ensuring that a message cannot appear on a display while a program is waiting for input (which would lose the waiting program's prompt).

☐ The Random Block File Manager (RBF)

Purpose: write one or more consecutive sectors to a disk.
Parameter convention:
Passed: **d0.l** = number of sectors
d2.l = starting Logical Sector Number (base 0)
PD_BUF(a1) = memory to write from
Returns: nothing

This routine is very much the complement of the read routine, and most RBF device drivers will use the same subroutines for most of both operations. As with the read routine, the write routine must translate the RBF logical

parameters to the appropriate physical parameters for the disk controller, initialize the controller whenever the format parameters have changed, and copy the first 22 bytes of the data to the drive table whenever LSN zero is being written (provided the write operation is successful).

In addition to writing the data, a device driver for a controller that does not support error detection and correction (EDC) should verify that the write operation was successful, by reading the sector that has just been written. Some controllers implement a "verify" command that reads the sector to check it, but without returning the data to the interface. In the absence of such a facility the device driver must read the sector to a local buffer in the device static storage (or dynamically allocated in the initialization routine). The controller will generate an error condition if the sector was not written successfully. The device driver can then retry the write operation a few times, eventually returning a "write" error (**E$Write**) to RBF if the sector cannot be written successfully.

For maximum confidence of data integrity the driver can compare the data read back by the verify operation with the data in the write buffer. This will reveal any errors in the transfer of data between the interface and the controller.

If the **PD_VFY** field of the path descriptor is not zero the driver should not perform the verify operation. Verifying after each sector is written is very time consuming, because the controller must wait for the disk to rotate a complete revolution before reading the sector that was just written. Therefore some programs - such as the **copy** and **backup** utilities - set the **PD_VFY** field non-zero while writing large blocks of data, to speed up data transfers.

RBF provides the same read-write lockout as described above for SCF. This greatly reduces the complexity of the device driver, and is appropriate because most block-structured devices cannot support concurrent read and write operations. The lockout is for the whole device. Therefore RBF will not call the driver to read or write on this or another drive on the same interface while a previous read or write request is not yet complete.

12.6.5 Get Status and Set Status

These are "wild card" routines. That is, they are a mechanism to permit any function to be implemented. By convention, the Get Status routine is used to request information from the device or driver, while the Set Status routine is used to request device or driver operations.

In general the file manager will perform the same device lockout for these routines as for the read and write routines, so it is permissible for the device driver to sleep as part of one of these calls. However, SCF does not implement device lockout in its Get Status routine. Therefore SCF device drivers must not sleep in a Get Status routine, or else the driver must implement the device lockout itself. In practice, if the SCF device driver writer wishes to add extra functionality to get information from the device, and the function may need to sleep, a Set Status call should be used rather than a Get Status call, to overcome this problem.

As with all I/O system calls, the Get Status (**I$GetStt**) and Set Status (**I$SetStt**) system calls go first to the kernel. The specific function required is indicated by a function code parameter to the system call (in the **d1.w** register). The kernel checks this code to see if it is known to the kernel. If so, the kernel executes the desired function. In either case, the kernel then calls the Get Status (or Set Status) routine of the file manager, passing it the same function code. If the file manager returns an "unknown request" error (**E$UnkSvc**) for a function code that the kernel recognized, the kernel returns no error to the calling program. Otherwise the kernel returns the error returned by the file manager. Of course, if the kernel recognizes the function code, but experiences an error in executing the appropriate function, it does not call the file manager, but returns the error to the calling program.

Typically the file manager will behave like the kernel – that is, it checks the code, executes the appropriate function if it recognized the code, and then calls the Get Status (or Set Status) routine of the device driver. If the driver returns an "unknown request" error for a function code that the file manager recognized, the file manager returns no error to the kernel.

This mechanism whereby the call is passed from the kernel to the file manager, and from the file manager to the device driver, allows each of the three modules to implement any number of functions that may be unknown to the other two. And because even recognized calls are still passed down the tree, a call that requires action by two modules can be implemented. For example, the **SS_Opt** Set Status function to alter the options section of the path descriptor is acted on by the file manager, but because it is also afterwards passed to the device driver, the driver can use the new parameters in the options section to reconfigure the interface chip.

The kernel recognizes no Set Status function codes, and only two Get Status functions: **SS_Opt** (return a copy of the path descriptor options section, 128 bytes), and **SS_DevNm** (return a copy of the device descriptor module

name). These requests are made by the C library functions **_gs_opt()** and **_gs_devn()** respectively.

In addition to calls from a program, the file manager may generate Get Status or Set Status calls to the device driver. This is an alternative to defining additional routines in the driver, and has the advantage that the file manager writer can add more such calls in a later release of the file manager without the need to change the device driver - the driver will automatically return an "unknown request" error to the new calls. The kernel does not generate such calls, although it may do in future releases.

As with other system calls, the parameters are passed from the calling program in processor registers. However, because any function can be defined by the kernel, the file manager, or the device driver, the kernel or file manager cannot simplify the environment of the device driver by passing the parameters in process registers to the device driver. Instead, the driver must read the calling program's register stack frame (built by the kernel when the system call is made), which is pointed to by the **PD_RGS** field of the path descriptor. Similarly, to return values to the calling program the driver must write to the stack frame. For example, to read the calling program's **d2** register (in this case, into the device driver's **d0** register):

```
movea.l PD_RGS(a1),a5   get stack frame pointer
move.l  R$d2(a5),d0     get caller's d2 register
```

In C an equivalent code fragment would be:

```
x=pathdesc->pd_rgs->d[2];       /* get caller's d2 register */
```

The file 'DEFS/process.a' defines the symbolic definitions - such as **R$d2** - for the structure of the stack frame in assembly language, while 'DEFS/MACHINE/reg.h' declares the same for C. If the file manager itself generates a call that requires parameters, it must save the current parameter register values from the stack frame, put in the parameters it wishes to pass, call the device driver routine, and then restore the saved values. This complication is not needed if the call is one that will not be made from a program - it is only internally generated by the file manager. In this case the file manager could pass the parameters in processor registers, as is done for the read and write routines.

In general, a file manager will recognize at least the **SS_Opt** function of the Set Status call. This requests the file manager to update the options section of the path descriptor. The file manager implements this call, rather than the kernel, because normally the file manager will only permit the calling program to alter certain fields of the options section. For example, SCF will allow the program to modify any of the fields of the options section proper,

while RBF will only allow modification of fields up to and including the **PD_SAS** field.

SCF and RBF generate a Set Status call **SS_Open** to the device driver when a path is opened or created, so a new path descriptor has been created, and a Set Status call **SS_Close** when the last image of a path is closed (the path descriptor is about to be de-allocated). SCF also generates a Set Status call **SS_Relea** when process closes a path and the process does not have any remaining duplications of the path. This is done in case a process requested the sending of a signal when data was received (**SS_SSig** Set Status call to the driver), and then died without being sent the signal, and without having cancelled the request.

If the cancellation was not forced by SCF, when new data arrived the driver might send a signal to a new (and unsuspecting) process that was created with the ID released by the dead process. Note that the device driver writer must bear this kind of complication in mind when adding Set Status or Get Status functions to a device driver. A process that has installed a request for action at some future time may die unexpectedly in the intervening period, and this should not be catastrophic to the system. An SCF driver can use the **SS_Relea** and **SS_Close** calls from the file manager to ensure that all such pending requests for a process are cancelled. This requires that the driver save the process ID and system path number when the request is first made, so that it can match the pending request with the call from the file manager.

While device drivers will vary in the Get Status and Set Status functions that they support, a device driver for general use should support at least the functions that are supported by the standard device drivers supplied by Microware as example source code, and in the OS-9 implementations that Microware has carried out. This ensures a common base level environment that all programs can expect. The list below shows the standard functions for an SCF and an RBF device driver. The assembly language symbolic name for the function code is given, together with the C library function provided to make the call from a C program.

☐ Get Status calls for an SCF Driver

Name	C function	Description
SS_Ready	_gs_rdy	Returns the number of characters available in the input buffer to the caller's **d1** register. If there are no characters in the input buffer, returns "not ready" error - **E$NotRdy**.

Name	C function	Description
SS_EOF	_gs_eof	Returns "end of file" error if at end of file. As SCF does not support a filing structure, there is no static end of file condition, so this function never returns an error in an SCF device driver.

The assembly language code below is typical of the Get Status routine of an SCF driver, and illustrates the use of the calling program's stack frame. Note that although the calling program passes the function code in the **d1.w** register, the file manager moves it to the **d0.w** register before calling the device driver. Of course, the function code could also be obtained from the **d1.w** register of the calling program's stack frame.

```
* GetStat
* SCF device driver Get Status "wild card" routine
* Passed:   d0.w = function code
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
* Returns:  depends on function
*
GetStat:    movea.l PD_RGS(a1),a5   get caller's stack frame pointer
            movea.l V_PORT(a2),a3   get port (interface chip) address
            cmpi.w  #SS_Ready,d0    check data ready?
            bne.s   GetStat10       ..no
            move.l  InBufCnt(a2),d0 get number of characters in input
*                                   buffer
            beq.s   NotRdyErr       ..none; return "not ready" error
            move.l  d0,R$d1(a5)     return number in caller's d1
            bra.s   GetStatEx       ..exit; carry is clear

GetStat10   cmpi.w  #SS_EOF,d0      check for end of file?
            bne.s   UnkSvcErr       ..no; unknown request
* The carry flag is now clear. Fall through to exit - there is never an
* end of file condition on a terminal or printer:

GetStatEx   rts

* Return "not ready" error:
NotRdyErr   move.w  #E$NotRdy,d1    set error code in d1.w register
            ori     #Carry,ccr      set carry flag to show error
            rts

* Return "unknown request" error:
UnkSvcErr   move.w  #E$UnkSvc,d1    set error code in d1.w register
            ori     #Carry,ccr      set carry flag to show error
            rts
```

☐ Get Status calls for an RBF Driver

Prior to OS-9 version 2.4 there were no standard Get Status calls for an RBF device driver. The following calls were added to RBF drivers in OS-9 version 2.4. As described with each call, drivers written prior to OS-9 version 2.4 - and so returning an "unknown request" error to these calls - will still work with OS-9 version 2.4, but without the benefit of some of the added features of OS-9 version 2.4.

Name	C function	Description
SS_VarSect	(none)	RBF makes this call when opening a path, to determine whether the driver and device can support sector sizes other than 256 bytes. If so, the driver should check the value in **PD_SSize**. If it is not zero, the driver should verify that it is a valid sector size, supported by the device and the driver. If so, the driver returns no error, otherwise the driver returns an error - typically "parameter error" - **E$Param** - or "hardware error" - **E$Hardware**. If **PD_SSize** is zero, the driver should put the current device sector size in **PD_SSize**, and return no error. If the driver does not support sector sizes other than 256 bytes (for example, most drivers prior to OS-9 version 2.4), the driver returns an "unknown request" error, in which case RBF assumes a sector size of 256 bytes, and ignores any value in **PD_SSize**. If the driver returns an error other than "unknown request" RBF aborts the opening of the path, otherwise RBF puts the sector size in **PD_SctSiz**. Note that RBF makes this request before allocating its sector buffers, so the driver cannot use the memory pointed to by **PD_BUF**.
SS_DSize	(none)	Request the disk data capacity in sectors. The **format** utility makes this call, to avoid the need to use the path descriptor options section parameters to calculate the disk data capacity if the controller can determine the capacity, thus allowing one form of device descriptor to be used for a range of disk drives. If the disk controller can determine the disk data capacity (that is, the disk space usable by the filing system, excluding sectors reserved by the controller or disk drive), the driver should issue a command to the controller to determine the disk capacity, and return it in the calling program's **d2.l** register (in the register stack frame). Otherwise the driver should return an "unknown request" error, in which case the **format** utility calculates the disk capacity from the number of cylinders, tracks per cylinder, and sectors per track specified in the path descriptor options section.

☐ Set Status calls for an SCF Driver

Name	C function	Description
SS_Opt	_ss_opt	Modify the path descriptor options section. The device driver must check whether the parameters it uses to configure the device (such as **PD_PAR** and **PD_BAU**) have changed. If so, the driver must reconfigure the device. If there is no change, the driver should not reconfigure the device, as altering the configuration registers of an interface chip may corrupt characters currently being transmitted or received.
SS_SSig	_ss_ssig	Request that the driver send a signal to the process when data becomes available. The calling program passes the desired signal code in the **d2.w** register. The device driver must save not only the signal code, but also the caller's process ID, and the system path number (from the path descriptor field **PD_PD**), in order to know which process to send the signal to, and to provide a check for a subsequent **SS_Relea** call. Once the signal has been sent (usually by the interrupt service routine, when a character is received), the driver "forgets" the call. A new **SS_SSig** call must be made by the program if it wishes to receive another signal. If data is already available in the input buffer the driver must send the signal immediately. The driver must mask interrupts up to the interrupt level of the device while checking for input data (and perhaps sending a signal), to avoid a race condition with an incoming character invoking the interrupt service routine. Normally the device driver will permit only one such request to be pending at any one time. That is, if a process has made this request but has not yet be sent a signal (and has not cancelled the request with **SS_Relea**), then a "not ready" error - **E$NotRdy** - is returned to any other **SS_SSig** request. Also, typically the driver will return a "not ready" error to any call to the read routine while an **SS_SSig** request is pending.
SS_DCOn	_ss_dcon	Request that the driver send a signal when the DCD (Data Carrier Detect) handshake input becomes asserted. This request (and the **SS_DCOff**, **SS_EnRTS**, and **SS_DsRTS** requests) is only appropriate for a serial port device with modem handshake lines (typically available on all asynchronous serial ports, such as RS232C). This request is similar to the **SS_SSig** request, except that the read routine does not return a "not ready" error if an **SS_DCOn** request is pending. This function can only be supported if the device can generate an interrupt when DCD becomes asserted (otherwise the driver must return an "unknown request" error - **E$UnkSvc**).
SS_DCOff	_ss_dcoff	Request that the driver send a signal when the DCD (Data

Name	C function	Description
		Carrier Detect) handshake input becomes negated. This is similar to the **SS_DCOn** request. This function can only be supported if the device can generate an interrupt when DCD becomes negated (otherwise the driver must return an "unknown request" error - **E$UnkSvc**).
SS_Relea	_ss_rel	Cancels any outstanding **SS_SSig**, **SS_DCOn**, and **SS_DCOff** requests for this process on this path.
SS_EnRTS	_ss_enrts	Requests that the driver assert the RTS handshake output of the device. The circuit configuration or the interface chip behaviour may make it more appropriate to assert the DTR handshake output signal instead - the device driver documentation should make it clear which signal is manipulated by this function. If no output signal can be manipulated manually by the device driver it should return an "unknown request" error.
SS_DsRTS	_ss_dsrts	This is the complement of the **SS_EnRTS** request. It requests that the device driver negate the RTS (or DTR) handshake output of the device.

☐ Set Status calls for an RBF Driver

Name	C function	Description
SS_Reset	_ss_rest	Restore the drive head to cylinder zero. Floppy disk drives usually have a "head at cylinder zero" sensor. The only way in which the controller can know which cylinder the drive head is on is by knowing how many steps the head has taken from cylinder zero. After many steps backwards and forwards a slight positional error may accrue, due to the mechanical characteristics of the drive. The driver will therefore usually automatically use a special controller command to restore the head to cylinder zero when reading or writing a sector on cylinder zero, or after a read or write operation gives a "seek error" (in order to reconfirm that the head is on the correct cylinder). During a format operation, however, the controller cannot give a seek error, as it is not reading sector address information from the unformatted disk. Therefore the **format** utility makes this request after formatting a number of cylinders, to ensure the head alignment is correct (the driver will move the drive head to the correct cylinder on the next "format a track" request).
SS_WTrk	_ss_wtrk	Request to format a track of a disk. The calling program (usually the **format** utility) specifies a surface of a cylinder to format (see below).

The **SS_WTrk** request is used by the **format** utility, which issues a request to format each surface of each cylinder. The cylinder number (base zero) is specified in the caller's **d2.w** register. The surface number (base zero) is specified in bits 8:15 of the caller's **d3.w** register. Many controllers (such as SCSI controllers) have a command to format the whole disk. A driver for such a controller only reacts to a **SS_WTrk** request for track zero (both cylinder and surface are zero), in response to which the driver issues the controller command to format all of the disk. Such a driver takes no action and returns no error for requests on other cylinders or surfaces.

Many floppy disk control circuits use Western Digital controller chips, such as the 177x series, and the 279x series. These controllers require the computer to provide a complete byte stream to format the track. Because these controllers are in common use, and generating such a "track buffer" requires a great deal of detailed knowledge and programming effort, Microware has included the generation of a Western Digital track buffer in the **format** utility. The buffer is pointed to by the caller's **a0** register, and can be transferred directly to the controller chip in the same way as read or write data.

A driver for a different type of controller may need to build a track buffer or sector address list. This requires the use of the "physical sector interleave factor" to determine the order in which the sectors are to be numbered on the track. (This allows the optimization of reading or writing logically consecutive sectors). Because this is also a common requirement, the **format** utility builds a sector number list, otherwise known as an interleave table. The list is pointed to by the caller's **a1** register, and is the same for every track. It consists of an array of bytes, one for each sector number, in the order in which the sector numbers are to be used on the track. The sector numbers are base zero. Therefore the driver should add the offset in **PD_SOffs** to each sector number when building the track buffer or sector address list.

The third group of controllers comprises those (such as SCSI controllers) that require only a high level command to format a track or the whole disk. Such a controller may allow the driver to specify the physical sector interleave factor. The interleave factor is given in the **PD_ILV** field of the path descriptor options section. However, this field cannot be modified by the **SS_Opt** Set Status call. Therefore, in order to allow the user to specify the interleave factor using the '-i' option, the **format** utility passes the interleave factor in the **d4.b** register (using the value in **PD_ILV** if the '-i' option is not used). Therefore the driver should use the value in the caller's **d4.b** register, rather than the value in **PD_ILV**.

In order that the user can format a single track density disk in a double track density drive, the **format** utility also passes a field of format flags in the **d3.b** register. This field has the same structure as the **DD_FMT** field of LSN zero and the drive table, but applies separately for each track. For example, bit 0 is zero when formatting surface (side) zero, and one when formatting surface one (or other surfaces), rather than indicating whether the disk is single or double sided. To support disks with more than two surfaces, bits 8:15 of the **d3.w** register specify the surface number (base zero).

Some drivers (particularly those for "intelligent" controllers, such as SCSI controllers) also implement the **SS_DCmd** Set Status request, allowing the calling program to pass any command directly to the controller. There is no set parameter format for this request, and the calling program must know both the request parameter format and the controller command and response structure.

12.7 INTERRUPTS

This section discusses the purpose of interrupts, and how they are used under the OS-9 operating system. Although OS-9 makes no requirement that a device driver must use interrupts, they are essential to the proper operation of any multi-tasking or real time operating system. Interrupts are used for two distinct purposes:

a) To signal the occurrence of hardware events for which there may be no process waiting. Examples are clock ticks, and serial port type-ahead and write-behind. I have named this an **unsolicited interrupt**, because the interrupt occurs without being specifically requested.

b) To wake up a sleeping process that is waiting for the completion of a hardware operation, so allowing the processor to execute other processes. Examples are disk and tape operations. I have named this a **solicited interrupt**, because the interrupt cannot occur unless a process has requested the interrupt and is waiting for it.

Of course, unsolicited interrupts can only occur if the device driver (or other software) has enabled interrupt generation in the interface chip. Nonetheless, the distinction between solicited and unsolicited interrupts is an important one, with significant implications for the device driver writer.

Interrupts are a function of external hardware, and are therefore *totally asynchronous* to the normal program flow of control. It is most likely that the process waiting for the interrupt (if there is one!) is not executing at the time of the interrupt. It will be asleep, and another process will be executing. It is this asynchronicity that gives rise to all of the conceptual and programming problems of interrupts. Once this concept has been mastered the programming precautions necessary to use interrupts are obvious and simple.

An interrupt is handled by an interrupt service routine. Interrupt service routines are normally in device drivers, because it is the device drivers that handle the hardware that causes the interrupts. However, this is not a fixed requirement of OS-9. Any software operating in system state can install an interrupt service routine (using the **F$IRQ** system call) - for example, a kernel customization module or a system state trap handler. The interrupt service routine itself can be located anywhere in memory, although it is normally located within the module that installed it. Otherwise there may be a risk that the module containing the interrupt service routine is unlinked before the corresponding interrupts are disabled.

The 68000 family of microprocessors supports 199 separate interrupt vectors - 7 autovectors (25 to 31) and 192 normal vectors (64 to 255). An OS-9 interrupt service routine services one such vector (although multiple interrupt service routines can be installed on the same vector). Because OS-9 is completely customizable, interrupt service routines can be dynamically installed and removed, using the **F$IRQ** system call. At coldstart the kernel sets all of the exception jump table entries for interrupts to point to the kernel's interrupt handler function, and builds a table in the System Globals of 199 pointers, all initially null.

When the **F$IRQ** system call is made to install an interrupt service routine, the kernel finds a free entry in the array of structures known as the interrupt polling table. It adds the entry to the linked list pointed to by the root pointer for the vector on which the interrupt service routine is being installed. Thus the interrupt polling table contains up to 199 separate linked lists, intertwined together. Within each linked list the order of the entries is determined by the "polling priority" value passed to the **F$IRQ** call - a low priority value puts the entry nearer the root of the list. If two entries have the same priority value the chronologically later entry is placed after the earlier entry. A priority value of zero is a special case - the kernel ensures that this is the only entry on the specified vector. If the linked list ("queue") for this vector is not empty when the request with priority zero is made, the caller is returned a "vector busy" error (**E$VctBsy**). The same error is

returned if an **F$IRQ** call is made for a vector on which there is already an entry installed with a priority of zero.

The interrupt polling table entry contains the address of the interrupt service routine, the static storage pointer passed to the **F$IRQ** call (usually the address of the device static storage), and the "port address" passed to the **F$IRQ** call. When the processor initiates interrupt exception processing it jumps to the exception jump table entry for the interrupting vector, which jumps to the kernel's interrupt handler. The kernel uses the vector number to select the appropriate root pointer. It then calls the interrupt service routine from the first entry in the queue (lowest priority value). If that routine returns the processor carry flag set, the kernel calls the routine from the next entry, and so on until a routine returns the carry flag clear (or the queue is exhausted - see below).

This technique allows all 199 vectors to be supported without wasting memory. Multiple devices can use the same vector. However, this is normally only necessary for autovectored devices (the vector is determined by the interrupt level), as there are only 7 autovectors. The **F$IRQ** system call is also used to remove an entry from the interrupt polling table, permitting complete termination of the resources of a device.

The installed interrupt service routine is called by the kernel with the **a2** register containing the same static storage pointer (and the **a3** register containing the same "port address") as was passed to the **F$IRQ** system call. The static storage pointer will normally be the address of the device static storage for the interrupting device, allowing the device driver and interrupt service routine to have common access to shared variables. As the interrupt service routine is normally part of the device driver module, the interrupt service routine will use the same symbolic names for the variables as the main body of the device driver. The "port address" in the **a3** register is not used at all by the kernel - it is passed merely as a convenience to the interrupt service routine (which could otherwise have read it from the device static storage).

The interrupt service routine is only permitted to destroy the **d0**, **d1**, **a0**, **a2**, **a3**, and **a6** registers (unless bit 0 of the first compatibility byte in the **init** module is set). An interrupt service routine will not normally modify the processor interrupt mask in the status register, except perhaps to temporarily set the mask to level 7 to mask interrupts from other devices when executing code fragments that interact with other interrupting devices.

The interrupt mask should never be lowered below the interrupt level of the interrupting device, as this could lead to nested interrupts, eventually crashing the system. If the interrupt service routine cannot handle one interrupt from the device before it generates another interrupt, it will not help to expose the system to the second interrupt before the first has been handled! When the interrupt service routine finishes, it returns to the kernel with the **rts** instruction just like any other subroutine, not the **rte** instruction. In summary, the calling convention for an interrupt service routine is:

```
Passed:  (a2) = static storage (usually device static storage)
         a3.l = port address
         (a6) = System Globals
Returns: carry set if not this driver's interrupt
May destroy:     d0-d1/a0/a2-a3/a6
```

While the kernel's interrupt handler does not make any use of the static storage pointed to by the interrupt polling table entry, the static storage pointer value is used to identify an entry in the linked list for a vector when the **F$IRQ** call is used to remove an entry from the polling table. The kernel determines that the call is being used to remove an entry because the interrupt service routine address (in the **a0** register) is zero. It then scans the linked list for the given vector, looking for a match for the given static storage pointer (in the **a2** register). This implies that two entries on the same vector must not be installed with the same static storage pointer. This is not a problem - different device drivers (even different incarnations of the same device driver module) will have different device static storage addresses, and a driver will not need to install two interrupt service routines on the same vector.

An interrupt service routine cannot make use of most of the system calls. This is because the interrupt may occur while a process is making the same system call (or a related one), and a "nested" call might damage operating system memory structures. The following system calls are available for use by interrupt service routines (the kernel masks interrupts during critical code fragments in these system calls):

F$Event	All event functions except **Ev$Creat**, **Ev$Delet**, **Ev$Link**, and **Ev$Info**.
F$Send	Send a signal.
F$APROC	Put a process into the active queue.
F$NProc	Make the next process in the active queue the

	current process.
F$Move	Copy a block of memory.
F$CCtl	Flush, enable, or disable the processor caches.
F$Time	Get the current date and time.
F$Julian	Convert Gregorian date and time to Julian.
F$Gregor	Convert Julian date and time to Gregorian.

Because of the asynchronous nature of OS-9 signals - able to cause the asynchronous execution of a signal intercept handler function - there is often conceptual confusion between interrupts and signals. The confusion is sometimes increased because most interrupt service routines send signals. Interrupts are a function of external hardware and the interrupt circuitry of the processor. Interrupts are masked using the interrupt mask field of the processor's status register. By contrast, signals are a software function only, and are masked by the **F$SigMask** system call. If an interrupt service routine sends a signal, the receiving process's signal intercept handler is not called until the process next runs in user state, which cannot occur at least until the interrupt service routine has completed. The signal intercept handler is *not* called during the execution of the interrupt service routine, and interrupts are *not* masked when a signal intercept handler is called.

While the job to be done by an interrupt service routine varies widely, some basic principles apply. The interrupt service routine must first ascertain that its device caused the interrupt, usually by reading a status register from the interface chip. If not, it simply returns to the kernel with the processor's carry flag set. If the interrupt service routine was installed in the polling table with a priority of zero then it does not need to check that its device caused the interrupt, as it is the only device using this vector number. This is an essential mechanism for some interfaces that have no status flag showing that they have an interrupt pending.

Once the interrupt service routine has verified that its device generated the interrupt, it must:

a) Clear the interrupt to the processor. Many normal vectoring devices clear the interrupt automatically once they have sent the interrupt vector to the processor (interrupt acknowledge cycle).

b) Carry out any immediately required operations. These must be kept to a minimum - processes cannot run while an

interrupt is being serviced, and other interrupts on the same and lower interrupt levels cannot be serviced. In general, if at all possible operations should be left to be carried out by the device driver main body once it has been woken – which may incur a delay of tens of milliseconds.

c) Wake up any waiting process. This refers to the main body of the device driver having executed a "sleep" request (**F$Sleep**) on behalf of the calling process, waiting for the interrupt to occur. For a solicited interrupt there will always be a waiting process. For an unsolicited interrupt there may be a waiting process, but not always.

The handling of interrupts, as with most of the code in device drivers, is very much to do with understanding and managing the hardware. However, a discussion of the control of hardware interface devices is outside the scope of this book. From an operating system point of view the important element is the interaction with any waiting process. It is with this aspect that the following discussion is concerned.

12.7.1 Solicited Interrupts

A solicited interrupt should be used wherever the device driver estimates that a hardware operation will take longer than the time that would be required for the driver to go to sleep and be woken by an interrupt service routine. That is, more processor time will be used by polling the interface status register until the operation is complete than by waiting for an interrupt.

Solicited interrupts are relatively easy to handle. The device driver decides that a hardware operation is going to take some time, and rather than wait by polling a status register it elects to give up its usage of processor time and wait for an interrupt. As the interrupt cannot occur until the driver has performed the device function that initiates the interrupt mechanism, control is straightforward:

1) Set a flag in the static storage indicating to the interrupt service routine that a process needs waking, together with the ID of the process to wake (the current process). It is usually convenient to combine the two items, because no process has a process ID of zero. Therefore if the static storage field containing the ID of the process to wake is zero, no process is

waiting to be woken.

2) Initiate the device operation, with the interface chip set to generate an interrupt when the operation is complete.

3) Go to sleep. The **F$Sleep** system call will return when the process is woken by a signal, or – for a timed sleep – when the sleep time expires. A timed sleep is only used if the driver wishes to implement a timeout on the hardware operation.

It is very important not to reverse operations 1 and 2. The interrupt may come in at any time *after* the device operation has been initiated, and the interrupt service routine must know that it has a process to wake. It does not matter if the interrupt occurs between stages 2 and 3 (that is, before the driver has executed the "sleep" request). The kernel leaves the signal sent by the interrupt service routine pending in the process descriptor. The **F$Sleep** system call sees that a signal has been received and immediately returns to the driver without putting the process to sleep.

Once the driver has been woken it must verify that the interrupt service routine sent the signal – the signal may have come from another process communicating with the process that called the device driver. If the hardware operation is not complete the driver must go back to sleep (unless it decides that the received signal was "deadly"). Because the driver is executing in system state, all the signals sent to the process are queued in the process descriptor until the process returns to user state (at the end of the system call that called the driver). Therefore no signals are lost. The "wakeup" signal – **S$Wake** – is an exception. It is not queued, and is therefore only suitable for use by an interrupt service routine waking up a device driver.

The driver is woken by each signal received. The kernel sets a flag in the process descriptor to show that the latest signal caused a wakeup, so that when the driver goes back to sleep (because the signal was not from the interrupt service routine), the **F$Sleep** system call permits the sleep – it does not return immediately to the driver, even though a signal is pending for the process.

Because this mechanism is so commonly used, Microware have defined two fields in the kernel part of the device static storage to support it: **V_BUSY** and **V_WAKE**. These fields are not used at all by the kernel. The field **V_BUSY** contains the ID of the calling process, set by the file manager as part of its interlock on the device (see the section on Resource Control in the chapter on File Managers). The field **V_WAKE** is the flag field described

above. The driver copies the process ID to this field, setting it non-zero as an indication to the interrupt service routine that a process needs waking. The interrupt service routine clears the field (after taking the process ID) as a handshake to the main body of the driver, and to prevent further wakeups. For example:

```
            move.w  V_BUSY(a2),V_WAKE(a2)   set flag and process ID
            bsr     IssCmd                  initiate device operation
Loop        moveq   #0,d0                   indicate indefinite sleep
            os9     F$Sleep                 sleep until woken
            tst.w   V_WAKE(a2)              woken by interrupt?
            bne.s   Loop                    ..no; go back to sleep
```

In this example the driver does not consider any signal is "deadly" - that is, a signal important enough to abort the operation. Therefore if on wakeup it finds that it has not been woken by the interrupt service routine, it goes back to sleep without checking the signal that caused the wakeup.

Note the use of the **V_BUSY** field as the source of the process ID. Most file managers put the current process ID in this device static storage field. However, the kernel does not set this field, and so it is not valid during the initialization and termination routines. A driver that needs to use interrupts within the initialization or termination routines must take the process ID from the process descriptor:

```
            move.w  P$ID(a4),V_WAKE(a2)     set flag and process ID
```

If a common "sleeping" subroutine is used that assumes **V_BUSY** contains the process ID, then the initialization and termination routines could copy the process ID to the field. However, the initialization routine must be sure to clear this field before exiting, as the file manager will expect it to be clear in subsequent I/O calls (see the chapter on File Managers). Note that it is in any case inadvisable to sleep within the initialization routine (see the preceding section on the Initialize routine).

The corresponding interrupt service routine would be as shown below, assuming the routine has already determined that this is its interrupt, and taken any necessary action to clear it:

```
            move.w  V_WAKE(a2),d0   get ID of process to wake
            beq.s   IRQExit         ..none; (should not happen)
            clr.w   V_WAKE(a2)      show valid interrupt wakeup
            moveq   #S$Wake,d1      send special wakeup signal
            os9     F$Send          send the signal
IRQExit     moveq   #0,d1           clear carry - interrupt serviced
            rts                     return to kernel
```

Note the use of the signal code **S$Wake**. As already described, the kernel assigns special properties to this signal code, so that its only function is to ensure that a process is in the active queue.

12.7.2 Unsolicited Interrupts

Unsolicited interrupts – such as from serial port received data – are slightly more complex to handle. The device may generate an interrupt at any time, so it is important to prevent timing race conditions between the interrupt service routine and the main body of the device driver. This is done by preventing the recognition of the interrupt by the processor during critical code fragments in the main body of the driver. To do this, interrupts are masked in the status register up to the interrupt level of the device.

The interrupt level of the device is specified in the **M$IRQLvl** field of the device descriptor. The initialization routine of the driver can build a status register image with the interrupt mask set to that level, and save it in the device static storage for later use:

```
move.b  M$IRQLvl(a1),d0 get device interrupt level
lsl.w   #8,d0           shift to bits 8:10
bset    #SupvrBit+8,d0  set supervisor state bit
move.w  d0,IRQMask(a2)  save sr image
```

The following example is typical of a serial port device driver read routine. For simplicity this example ignores the need to send the XON flow control character if XOFF had been sent and the buffer is now at the low water mark:

```
Read        tst.w   SigPrc(a2)      SS_SSig request pending?
            bne     NotRdyErr       ..yes; read request not allowed
            move    sr,-(a7)        save current interrupt mask
            move    IRQMask(a2),sr  mask interrupts to device level
            bsr     InBufOut        get character from input buffer
            bcc.s   Read20          ..got one (in d0.b)
* The input buffer was empty. Sleep, waiting for data:
            move.w  V_BUSY(a2),V_WAKE(a2)   set flag and process ID
            move    (a7)+,sr        unmask interrupts
            bsr     Sleep           sleep
            bcs.s   ReadEx          ..fatal signal received; abort
            bra.s   Read            ..else try again
Read20      move.b  V_ERR(a2),d1    get error flag
            clr.b   V_ERR(a2)       reset it
            move    (a7)+,sr        unmask interrupts
            tst.b   d1              any errors?
            beq.s   ReadEx          ..no; carry is clear
            move.w  #E$Read,d1      return read error
            ori     #Carry,ccr      set carry to show error
```

```
ReadEx      rts

* Read request made while SS_SSig request is pending:
NotRdyErr   move.w  #E$NotRdy,d1    return "not ready" error
            ori     #Carry,ccr
            rts
```

The **InBufOut** subroutine gets a character from the input circular buffer, returning it in the **d0.b** register. If the input buffer is empty, the subroutine returns the processor carry flag set.

The **Sleep** subroutine sleeps indefinitely until woken by a signal, and then checks whether a deadly signal has been received by the process, or the process has been condemned (sent a "kill" signal). If so, the driver exits immediately with the signal code as the error code (or 1 if the process is condemned), not waiting to complete the I/O operation. (This would not be suitable in an RBF driver, where the operation must be completed or the disk filing system may be corrupted.)

Note that prior to OS-9 version 2.4 the **P$Signal** field of the process descriptor contained the *oldest* pending signal (the next signal to be processed). Therefore if a non-deadly signal was received followed by a deadly signal the check in the driver would only see the non-deadly signal, and not abort. From OS-9 version 2.4 onwards the **P$Signal** field contains the most recently received signal (not yet processed by the user program), so by checking this field the driver will see each signal in turn. Also, prior to OS-9 version 2.4 only the abort (quit) and interrupt signals (2 and 3) were considered deadly. From OS-9 version 2.4 onwards all signal codes below 32 are considered deadly.

```
Sleep       moveq   #0,d0           sleep without timeout
            os9     F$Sleep
            move.w  P$Signal(a4),d1 get most recent signal in d1.w
            beq.s   Sleep10         ..none
            cmpi.w  #S$Deadly,d1    deadly signal?
            bcs.s   SleepEr         ..yes; error
Sleep10     moveq   #0,d0           ensure carry is clear
            btst    #Condemn,P$State(a4)    is process dead?
            beq.s   SleepEx         ..no; exit with carry clear
            moveq   #1,d1           "unconditional abort" error
SleepEr     ori     #Carry,ccr      set carry to show error
SleepEx     rts
```

The corresponding code fragment in the interrupt service routine to wake up the waiting driver is the same as for a solicited interrupt. However, while for a solicited interrupt device driver it would be an error for an interrupt to occur without there being a process to wake up, in the case of unsolicited interrupts this is a common occurrence.

The important point to note in the above read routine example is that the processor interrupt mask was set to the interrupt level of the device while the check on the input buffer was made, and interrupts were not unmasked until a character had been taken from the buffer or the wakeup handshake flag **V_WAKE** had been set. Remember also that the processor automatically sets its interrupt mask to the interrupt level of the device during an interrupt service routine. The result is that these two code fragments are mutually exclusive - neither can asynchronously break into the other - permitting an "indivisible" set of operations. This does not preclude a higher level interrupt from another device being serviced while either routine is executing, but as that interrupt service routine is not communicating with this driver the possibility is not relevant. (If the device driver *does* also service a higher level interrupt, critical code fragments should mask interrupts to the higher level).

Note that once the **V_WAKE** flag is set the driver can unmask interrupts (indeed, it must unmask interrupts before making the **F$Sleep** system call). If an interrupt comes in after **V_WAKE** is set but before the driver has gone to sleep, the interrupt service routine will still send the signal. Because the process is the current process (it is not yet in the sleeping queue), the kernel will set the **B_WAKEUP** bit of the **P$SigFlg** field of the process descriptor, and the subsquent **F$Sleep** system call will return to the driver without sleeping.

Similarly, because the interrupt service routine clears the **V_WAKE** field when sending the signal, on wakeup the device driver will find this field clear if it has been sent the signal (although in this example the driver does not use this flag, but instead checks the input buffer again). Provided the device driver writer takes care to provide such an indivisible handshake between the main body of the driver and the interrupt service routine, there is no possibility of a timing race condition, and no interrupts will be missed.

The write routine for a serial port device driver is almost identical to the read routine, except that the write routine must sleep if the output buffer is full when it tries to put a character into the buffer. Also, the write routine has the responsibility for starting the "transmit stream" if transmitter interrupts had been disabled because the output buffer was empty.

When the interface chip generates a "transmitter ready" interrupt, the interrupt service routine checks the output buffer. If the buffer is not empty the interrupt service routine takes the next character from the buffer and writes it to the transmit register of the chip. The chip will generate another interrupt when its transmit register becomes empty again. Thus a continuous

stream of interrupts and character transmissions is maintained so long as the output buffer is not empty, which will be the case so long as the program (and SCF and the driver) provides data faster than the data transmission rate of the interface. If the buffer is empty, the interrupt service routine must command the chip to disable further "transmitter ready" interrupts, and set a "transmitter interrupts disabled" flag in the device static storage. Note that at this time the chip has room for at least one character in its transmitter register.

Before attempting to put the character in the output buffer (but after masking interrupts), the write routine checks whether the buffer is empty and transmitter interrupts are disabled (transmitter interrupts could be disabled because the "data received" interrupt service routine received the XOFF flow control character). If so, it knows the transmit stream has been broken, and must be restarted. It does this by writing the character directly to the transmit register of the chip (rather than to the output buffer in the device static storage), and enabling transmitter interrupts from the chip. It then clears the "transmitter interrupts disabled" flag.

Whether the write routine writes the character to the transmit register and then enables transmitter interrupts in the chip, or vice versa, depends on the behaviour of the transmitter interrupts of the chip. It is more widely applicable to enable the interrupts first, and then write the character. This will work if the chip generates an interrupt so long as the transmit register is empty (the interrupt will be generated, but then cleared when the character is written – meanwhile, the write routine has interrupts masked in the processor). It will also work if the chip generates an interrupt when the transmit *becomes* empty, provided the transmitter interrupts are enabled at that time. Enabling the interrupts before writing the character avoids a potential race condition.

12.7.3 Choosing Interrupt Levels

There has frequently been a much confusion over the philosophy which should be used to decide what interrupt level to assign to each device. However, a little thought will show that the decision can be made very easily. The only benefit of assigning a higher level of interrupt to one device than to another is that interrupts from the first device will pre-empt the service of interrupts for the second, and be accepted by the processor when the device driver for the second device has interrupts masked to the level of its device.

As all interrupts must eventually be handled by the processor, the important concern is that the interrupt from a device must be handled before the device

wishes to generate another interrupt of the same type. For example, if a serial port chip generates a "data received" interrupt, it must be serviced - and the character read from the chip - before the chip receives another character, assuming the chip has only a single level of buffer for received characters in addition to its receive shift register. However, if the chip has an 8 byte FIFO for received characters, it does not matter if the interrupt is not serviced before another character is received, provided it is serviced before 8 characters are received.

One important point is apparent here - solicited interrupts almost never need to be on a high level interrupt. Such interrupts are only generated in response to a command from the driver to the chip. If the driver takes a long time responding to the interrupt, no problem is caused, because the chip cannot need to generate another interrupt until the driver issues another command. Therefore chips that only generate solicited interrupts can be on a low interrupt level - 1 or 2, for example. A high interrupt level is only needed if a remote device needs a rapid response. For example, a communications protocol may specify a maximum response time.

This only leaves the question of how to select the interrupt levels for devices that generate unsolicited interrupts, such as communications ports, the clock tick hardware, and some network interfaces. Again, the answer is simple. The highest level of interrupt should be assigned to the device that can produce the shortest interval from one interrupt to the next. For example, a serial port operating at 19200 baud will generate interrupts roughly every 500μs, whereas a typical clock tick is 10ms. It follows that unless the serial port has a 20 character FIFO (unlikely!), it should have a higher level interrupt than the tick hardware. That is to say, it is less important that the response to a tick interrupt be delayed by a few microseconds, than that the serial port interrupt response be delayed by a similar time.

The only modifying consideration is the seriousness of the loss of an interrupt from a particular device. For example, a lost serial port interrupt will cause a communications error - hopefully recoverable - while a lost tick interrupt will cause an unrecoverable date and time error. However, a system that is so heavily loaded with interrupts is probably on the edge of failing in the application in any case.

12.8 A SKELETON DEVICE DRIVER

Often, part of the problem in writing a device driver for OS-9 is in knowing how to start. To help overcome this difficulty, this section shows the skeleton

of a device driver in assembly language. It provides the bones on which a device driver that actually controls a hardware interface can be built. Note the use of the file '/dd/DEFS/oskdefs.d'. This file contains definitions – such as module types – that cannot conveniently be taken from a library, due to limitations in the linker on the use of external symbols in arithmetic expressions.

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

* Static storage definitions (to form the last part of the Device
* Static storage):
            vsect
IRQMask     ds.w    1               sr image with interrupts masked
            ends                    end of static storage definitions

* Routine offset table:
EntryTable  dc.w    Init            initialize
            dc.w    Read            read
            dc.w    Write           write
            dc.w    GetStat         get status
            dc.w    SetStat         set status
            dc.w    Term            terminate
            dc.w    0               (exception handler)

* Initialize
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a6,ccr
Init        tst.w   d0              clear carry - no error
            rts

* Terminate
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* (kernel ignores any returned error)
* May destroy:  d0-d7/a0-a5,ccr (NOT a6)
Term        tst.w   d0              clear carry - no error
            rts
```

```
* Read
* Passed:   (a1) = Path Descriptor - (NOT SBF)
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
*   RBF only:   d0.l = number of sectors to read
*   RBF only:   d2.l = LSN of first sector to read
*   SBF only:   d0.l = number of bytes to read
*   SBF only:   (a0) = buffer to read to
*   SBF only:   (a3) = drive table
* Returns:  carry set if error, with error code in d1.w
*   SCF only:   d0.b = character read
*   SBF only:   d1.l = number of bytes read
* May destroy:  d0-d7/a0-a6,ccr
Read        tst.w   d0              clear carry - no error
            rts

* Write
* Passed:   (a1) = Path Descriptor - (NOT SBF)
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
*   RBF only:   d0.l = number of sectors to write
*   RBF only:   d2.l = LSN of first sector to write
*   SCF only:   d0.b = character to write
*   SBF only:   d0.l = number of bytes to write
*   SBF only:   (a0) = buffer to write from
*   SBF only:   (a3) = drive table
* Returns:      carry set if error, with error code in d1.w
*   SBF only:   d1.l = number of bytes written
* May destroy:  d0-d7/a0-a6,ccr
Write       tst.w   d0              clear carry - no error
            rts

* Get status
* Passed:   (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
*           d0.w = function code
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a6,ccr
GetStat     move.w  #E$UnkSvc,d1    unknown code
            ori     #Carry,ccr      return error
            rts
```

```
* Set status
* Passed:   (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor of current process
*           (a6) = System Globals
*           d0.w = function code
* Returns:  carry set if error, with error code in d1.w
* May destroy:  d0-d7/a0-a6,ccr
SetStat     move.w  #E$UnkSvc,d1    unknown code
            ori     #Carry,ccr      return error
            rts

            ends                    end of code
```

12.9 CLOCK DRIVERS

Each OS-9 system must have "clock" hardware and a clock driver, to support time-sliced multi-tasking, timed sleeps, alarms, and the maintenance of the date and time. As a minimum, the clock hardware must have a periodic timer, generating interrupts at regular intervals. This is the system "tick" interval, usually 10ms. The interval between ticks must be precisely constant, as the ticks are used to maintain the system date and time. This implies that the timer must be cyclic - there must be no need for software to retrigger the timer. Also, the tick period must be such that there are an integral number of ticks per second.

The clock hardware may also include a battery-backed "time of day" circuit. This is typically a separate chip, maintaining the date and time even when the computer is switched off.

It is the job of the clock driver to control this hardware, and to call the kernel's tick handler when a tick interrupt occurs. The clock driver is different from the OS-9 I/O device drivers. There is no associated device descriptor, path descriptor, device static storage, or file manager, and there can only be one clock driver in each system. The clock driver does not have a routine offset table such as device drivers have. Instead, the execution entry offset in the module header (the **M$Exec** field) points directly to the initialization routine of the clock driver. The kernel takes the name of the clock driver module from the **init** configuration module. The clock driver name string is pointed to by the offset in the **M$Clock** field of the **init** module.

The **F$STime** system call is used to set the date and time. The kernel's handler for this function writes the new date and time to the System Globals (**D_Year**, **D_Month**, **D_Day**, **D_Second**, and **D_Julian** fields), and then

calls the initialization routine of the clock driver. The clock driver must ensure that the tick hardware is configured and running, and write the date and time to the battery-backed clock chip (if the driver supports one). Because the **F$STime** system call may be made more than once, the driver should check whether it has already initialized the tick hardware. If so, it should not re-initialize it. The clock driver's initialization routine is called with the following parameters:

(a4) = Process Descriptor of calling process
(a5) = Caller's register stack frame
(a6) = System Globals

The initialization routine may destroy any registers except **a4**, **a5**, and **a6**. If the routine encounters an error, it should return it in the normal way - the carry flag set, and an error code in the **d1.w** register. When initializing the tick hardware, the driver must perform three functions:

a) Install its tick interrupt handler, using the **F$IRQ** system call. As there is no clock device descriptor, the driver must use hard-coded values for the interrupt vector and software polling priority (and the interrupt level). Although when making the call the **a2** register must not match that for any other device installed on the same vector, this is not normally a problem for the clock driver, as the kernel passes **a2** pointing to the module directory entry for the clock driver. However, if the clock driver uses **a2** before making the **F$IRQ** call, the driver writer must ensure it cannot be equal to the device static storage address of any present or subsequently installed device. Setting **a2** to zero for the **F$IRQ** system call is therefore recommended by Microware as being safe and consistent.

b) Initialize the **D_TckSec** (ticks per second) field of the System Globals. It is the responsibility of the clock driver to determine the number of ticks the tick hardware will generate each second. This keeps the kernel independent of the tick hardware. Any tick rate is permissible, provided that there is an integral number of ticks per second. 10ms is a typical period, giving 100 ticks per second. Too small a tick interval will cause tick interrupts and process scheduling to consume too large a fraction of the processor's time. Too large a tick interval may delay the real-time response of processes in a multi-tasking application, and may give too coarse a resolution for timed sleeps and alarms.

c) Initialize the tick hardware, including enabling the tick interrupts, provided the hardware has not been initialized by a previous call to the clock driver's initialization routine.

Because the driver must set the **D_TckSec** field of the System Globals, and the kernel initializes this field to zero, the driver can use this field to check whether its initialization routine has been called before – **D_TckSec** will be zero if the initialization routine is being called for the first time.

Having initialized the tick hardware if necessary, the clock driver must write the new date and time to the battery-backed clock chip, if one is supported. The time and date are in the caller's register stack frame. **R$d0(a5)** gives the time as 00HHMMSS, and **R$d1(a5)** gives the date as YYYYMMDD (as required by the **F$STime** system call).

However, if the month and day are zero, this is a request to read the date and time from the battery-backed clock chip, if one is supported. Instead of writing to the chip, the driver must read the current date and time from the chip, and set the **D_Year**, **D_Month**, **D_Day**, **D_Second**, and **D_Julian** fields of the System Globals. The **F$Julian** and **F$Gregor** system calls can be used to translate between Gregorian and Julian date and time formats. Note that the **D_Second** field is the number of seconds left until midnight, rather than seconds since midnight.

The kernel makes the **F$STime** system call with a date of zero as part of its coldstart procedure (after the calls to open the default paths, change the directories to the default mass storage device, and install the kernel customization modules, if any), unless bit 5 is set in the first compatibility byte of the **init** module. In this way the kernel starts the clock, and reads the current date and time if a battery-backed clock chip is supported by the clock driver.

The interrupt service routine of the clock driver is usually straightforward. As with any interrupt service routine, it must determine that the interrupt was generated by the tick hardware, and return to the kernel with the carry flag set if not. Otherwise it must clear the tick interrupt in the tick hardware (if it is not automatically cleared by the interrupt acknowledge cycle), and call the kernel's tick handler. The address of the kernel's tick handler is in the **D_Clock** field of the System Globals:

```
movea.l D_Clock(a6),a0  get tick handler address
jmp     (a0)            ..go to it
```

Just like any other interrupt service routine, the clock driver interrupt service routine may only destroy the **d0**, **d1**, **a0**, **a2**, **a3**, and **a6** registers (unless bit 0 of the first compatibility byte in the **init** module is set).

In most systems the clock tick interrupts are not produced by the same chip that provides the battery-backed date and time facility, but by a separate timer chip. To simplify the job of the clock device driver writer, from OS-9 version 2.3 onwards Microware's example clock drivers are separated each into two source files. One file contains the routines for managing the clock tick chip, and the other contains the routines for managing the date and time chip. The files have a common interface, so the clock driver can be made for any combination of the two chips.