Random Block File Manager (RBF)

RBF General Description

The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service requests to random-access devices. RBF can handle any number or type of such devices simultaneously (for example, large hard disk systems, small floppy systems, RAM disk systems, etc.) and is responsible for maintaining the logical file structure.

Because RBF is designed to support a wide range of devices with different performance and storage capacities, it is highly parameter-driven.

Some of the physical parameters RBF uses are stored on the media itself. On disk systems, this information is written on the first few sectors of track number zero. The device drivers also use this information, particularly the media format parameters stored on sector 0. These parameters are written by the format program when it initializes and tests the media. Storage systems that initialize themselves without using format are responsible for establishing the initial file structure of the media themselves (for example, RAM disk systems).

The following I/O service requests are handled by RBF:

I$ChgDir	I$Close	I$Create	I$Delete	I$GetStt
I$MakDir	I$Open	I$Read	I$ReadLn	I$Seek
I$SetStt	I$Write	I$Writln		

The following I/O service requests do not call RBF:

I$Attach	I$Detach	I$Dup

RBF I/O Service Requests

When a process makes one of the following system calls to an RBF device, RBF executes the file manager functions described for that call.

I$ChgDir RBF performs the following functions:

- **Sets the directory bit in the access mode**
- **Calls RBF's Open routine to search the specified pathlist**
- **If accessible, updates the appropriate default P$DIO pointer in the process descriptor**
- **Closes the path opened by the Open routine**

I$Close RBF performs the following functions:

- **Flushes any data that has not yet been written to the disk**
  (any partial block of data left from a previous write call)
- **Checks the use count in the path descriptor**
  If the use count is non-zero, no further action is taken. Otherwise, RBF:
  - **Updates the file descriptor, if necessary**
  - **Trims the file size, if necessary**
  - **Calls the device driver with the SS_Close SetStat**
    (ignores any returned errors)

I$Create RBF performs the following functions:

- **Initializes the path descriptor to the default option values**
- **Searches directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **If the file is found, RBF will return an error**
- **Creates a directory entry for the new file**
  If there is no free space in the directory, it is expanded to make room for the new entry.
- **Creates and initializes a file descriptor for the file**
  If an initial size allocation has been specified, RBF attempts to allocate the specified amount of disk space for the file. If not specified, the first I$Write expands the file.
- **Calls the device driver with an SS_Open SetStat**
  RBF ignores E$UnkSvc errors, but aborts I$Create on any other error.

I$Delete RBF performs the following functions:

- **Initializes the path descriptor to the default option values**
- **Searches any directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **Checks the permission attributes of the file**
  The file's directory bit (dirbit) must be turned off using the SS_Attr SetStat call before I$Delete is called. To delete the file, the user must have permission to write to the file and there cannot be other open paths to the file. An error is returned if these conditions are not met.
- **Decrements link count in file descriptor**
  If the link count becomes zero, all disk space associated with the file is returned. This includes the file's file descriptor block. If the link count is non-zero, no disk space is returned.
- **Removes directory entry for the file**

I$GetStt Refer to the I$GetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the RBF supported I$GetStt functions:

SS_EOF	Check for end-of-file condition.
SS_FD	Get a copy of the file descriptor.
SS_FDInf	Get a copy of a specified file descriptor.
SS_Opt	Read path descriptor options.
SS_Pos	Determine file position.
SS_Ready	Test for data ready.
SS_Size	Determine file's size.

All other GetStat calls are passed to the driver.

I$MakDir RBF performs a Create operation with the directory bit set for the file access mode. If the Create succeeds, RBF creates directory entries for "." and ".." in the new directory file and then closes the path opened by Create.

I$Open RBF performs the following functions:

- **Initializes the path descriptor with the default option values**
- **Searches any directories specified or implied by the pathlist**
  If the user does not have permission to access a directory element, an error is returned.
- **Checks the permission attributes of the file**
  If the user does not have permission to open the file in the access mode requested, an error is returned.
- **Updates the last modified date in the file descriptor, if open for writing**

- **Calls the device driver with the SS_Open SetStat**
  RBF ignores E$UnkSvc errors, but aborts the I$Open on any other error.

I$Read RBF performs the following functions:

- **Attempts to acquire a record lock of the section of the file**
  If the record is in use, RBF waits for the time specified by the SS_Ticks SetStat call. This value defaults to zero, resulting in an indefinite sleep until the record becomes available.
- **Determines if there is data left to read in the file**
  If there is none, an end-of-file error (E$EOF) is returned.
- **Calls the driver to read the data, as needed by RBF**
  Complete blocks of data are transferred directly into the process's buffer. Partial blocks are read into a buffer maintained by RBF after which the portion of data requested from those blocks are copied into the calling process's buffer. If the requested data was found in a buffer from a previous read, RBF copies the data to the calling process's buffer without calling the driver**.**

If the file is open only for reading, the record lock on the requested section is released immediately. If the file is open for update, the record remains locked. A read of 0 bytes, a read of a different section, or an I$Write releases the current section's record lock.

I$ReadLn I$ReadLn is similar to I$Read, except that RBF maintains a buffer to read data into using single sector reads. It searches the data until it locates the first end-of-record character (carriage return), or reads the number of bytes requested, whichever comes first. It copies the read buffer into the process's buffer as necessary.

If the file is open only for reading, the record lock on the requested section is released immediately. If the file is open for update, the record remains locked. A read of 0 bytes, a read of a different section, or an I$Write releases the current section's record lock.

NOTE: The portion of the file that is record locked begins at the file position from where the I$ReadLn call was made and continues through the number of bytes requested, regardless of whether the EOR is found earlier.

I$Seek RBF sets the current position in the path descriptor to the specified position. If RBF's internal buffer contains a sector which contains modified data, and the new position is not in that sector, the driver is called to write that sector before the current position in the path descriptor is updated.

I$SetStt Refer to the I$SetStt description in the ***OS-9 Technical Manual*** for a detailed explanation of the RBF supported I$SetStt functions:

SS_Attr	Set file's permission attributes.
SS_FD	Write some file descriptor information.
SS_Lock	Record lock a portion of the file.
SS_Opt	Write the path descriptor options.
SS_RsBit	Reserve bitmap sector.
SS_Size	Set the file's size.
SS_Ticks	Set the record locking time-out value.

All other SetStat calls are passed to the driver.

NOTE: SS_Opt is passed to the driver after processing by RBF. If an unknown service request error (E$UnkSvc) is returned by the driver, it is ignored.

I$Write

RBF performs the following functions:

- **Attempts to acquire a record lock of the section of the file**
  If the record is in use, RBF waits for the time specified by the SS_Ticks SetStat call. This value defaults to zero which results in an indefinite sleep until the record becomes available.
- **Expands the file, if necessary**
- **Calls the driver to write the data, as needed**
  Complete blocks of data are transferred directly from the process's buffer. Partial blocks are copied into a buffer maintained by RBF. This data is written after a subsequent write fills the buffer, or a seek, read, or write is done to another portion of the file, or when the file is closed.

Any active record lock is released once the section has been written. A write of zero bytes also releases the record lock.

I$Writln

I$Writln is similar to I$Write, except that RBF searches the calling process's data buffer for an end-of-record character (carriage return). If one is found, only the data up to that end-of-record character is written. If no end-of-record character is found, RBF writes the number of bytes specified by the caller.

Any active record lock is released once the section has been written. A write of 0 bytes also releases the record lock.

RBF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for RBF devices. The table immediately follows the standard device descriptor module header fields. The size of the table is defined in the M$Opt field.

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Class
$49	PD_DRV	Drive Number
$4A	PD_STP	Step Rate
$4B	PD_TYP	Device Type
$4C	PD_DNS	Density
$4D		Reserved
$4E	PD_CYL	Number of Cylinders
$50	PD_SID	Number of Heads/Sides
$51	PD_VFY	Disk Write Verification
$52	PD_SCT	Default Sectors/Track
$54	PD_T0S	Default Sectors/Track 0
$56	PD_SAS	Segment Allocation Size
$58	PD_ILV	Sector Interleave Factor
$59	PD_TFM	DMA Transfer Mode
$5A	PD_TOffs	Track Base Offset
$5B	PD_SOffs	Sector Base Offset
$5C	PD_SSize	Sector Size (in bytes)
$5E	PD_Cntl	Control Word
$60	PD_Trys	Number of Tries
$61	PD_LUN	SCSI Unit Number of Drive
$62	PD_WPC	Cylinder to Begin Write Precompensation
$64	PD_RWR	Cylinder to Begin Reduced Write Current
$66	PD_Park	Cylinder to Park Disk Head
$68	PD_LSNOffs	Logical Sector Offset
$6C	PD_TotCyls	Number of Cylinders On Device
$6E	PD_CtrlrID	SCSI Controller ID
$6F	PD_Rate	Data transfer/Disk Rotation Rates
$70	PD_ScsiOpt	SCSI Driver Options Flags
$74	PD_MaxCnt	Maximum Transfer Count

NOTE: In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, you must make the following adjustment: (M$DTyp - PD_OPT)

For example, to access the drive number in a device descriptor, use PD_DRV + (M$DTyp - PD_OPT). To access the drive number in the path descriptor, use PD_DRV. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Name	Description

PD_DTP **Device Type**

This field is set to one for RBF devices.

PD_DRV **Drive number**

This field is used to associate a one-byte logical integer with each drive that a driver/controller will handle. Each controller's drives should be numbered 0 to n-1 (n is the maximum number of drives the controller can handle and is set into V_NDRV by the driver's INIT routine). This number defines which drive table the driver and RBF access for this device. RBF uses this number to set up the drive table pointer (PD_DTB). Prior to initializing PD_DTB, RBF verifies that PD_DRV is valid for the driver by checking for a value less than V_NDRV in the driver's static storage. If not, RBF aborts the path open and returns an error. On simple hardware, this logical drive number is often the same as the physical drive number.

PD_STP **Step rate**

This field contains a code that sets the drive's head-stepping rate. To reduce access time, the step rate should be set to the fastest value of which the drive is capable. For floppy disks, the following codes are commonly used:

Step Code	5" Disks	8" Disks
0	30ms	15ms
1	20ms	10ms
2	12ms	6ms
3	6ms	3ms

For hard disks, the value in this field is usually driver dependent.

PD_TYP **Disk Type**

Defines the physical type of the disk, and indicates the revision level of the descriptor.

If bit 7 = 0, floppy disk parameters are described in bits 0-6:

bit 0: 0 = 5 1/4" floppy disk (pre-Version 2.4 of OS-9)
1 = 8" floppy disk (pre-Version 2.4 of OS-9)

bits 1-3: 0 = (pre-Version 2.4 descriptor) Bit 0 describes type/rates.
1 = 8" physical size
2 = 5 1/4" physical size
3 = 3 1/2" physical size
4-7: Reserved

bit 4: Reserved

bit 5: 0 = Track 0, side 0, single density
1 = Track 0, side 0, double density

bit 6: Reserved

If bit 7 = 1, hard disk parameters are described in bits 0-6:

bits 0-5: Reserved

bit 6: 0 = Fixed hard disk
1 = Removable hard disk

PD_DNS **Disk Density ***

Indicates the hardware density capabilities of a floppy disk drive:

bit 0: 0 = Single bit density (FM)
1 = Double bit density (MFM)

bit 1: 1 = Double track density 96 TPI/135 TPI)

bit 2: 1 = Quad track density (192 TPI)

bit 3: 1 = Octal track density (384 TPI)

PD_CYL **Number of cylinders (tracks) ***

Indicates the logical number of cylinders per disk. Format uses this value, PD_SID, and PD_SCT to determine the size of the drive. PD_CYL is often the same as the physical cylinder count (PD_TotCyls), but can be smaller if using partitioned drives (PD_LSNOffs) or track offsetting (PD_TOffs).

If the drive is an autosize drive (PD_Cntl), format ignores this field.

*** These parameters are format specific.**

Name	Description

PD_SID **Heads or Sides ***

This field indicates the number of heads for a hard disk (Heads) or the number of surfaces for a floppy disk (Sides). If the drive is an autosize drive (PD_Cntl), format ignores this field.

PD_VFY **Verify Flag**

This field indicates whether or not to verify write operations.

0 = verify disk write
1 = no verification

NOTE: Write verify operations are generally performed on floppy disks. They are not generally performed on hard disks because of the lower soft error rate of hard disks.

PD_SCT **Default sectors/track***

This field indicates the number of sectors per track. If the drive is an autosize drive (PD_Cntl), format ignores this field.

PD_T0S **Default Sectors/Track (Track 0) ***

This field indicates the number of sectors per track for track 0. This may be different than PD_SCT (depending on specific disk format). If the drive is an autosize drive (PD_Cntl), format ignores this field.

PD_SAS **Segment allocation size**

Indicates the default minimum number of sectors to be allocated when a file is expanded. Typically, this is set to the number of sectors on the media track (for example, 8 for floppy disks, 32 for hard disks), but can be adjusted to suit the requirements of the system.

PD_ILV **Sector interleave factor ***

Indicates the sequential arrangement of sectors on a disk (for example, 1, 2, 3... or 1, 3, 5...). For example, if the interleave factor is 2, the sectors are arranged by 2's (1, 3, 5...) starting at the base sector (see PD_SOffs).

NOTE: Optimized interleaving can drastically improve I/O throughput.

NOTE: PD_ILV is typically only used when the media is formatted, as format uses this field to determine the default interleave. However, when the media format occurs (I$SetStat, SS_WTrk call), the desired interleave is passed in the parameters of the call.

.

*** These parameters are format specific.**

Name	Description

PD_TFM **DMA (Direct Memory Access) transfer mode**

Indicates the mode of transfer for DMA access, if the driver is capable of handling different DMA modes. Use of this field is driver dependent.

PD_TOffs **Track base offset ***

This field is the offset to the first accessible physical track number. Track 0 is not always used as the base track because it is often a different density.

PD_SOffs **Sector base offset ***

This field is the offset to the first accessible physical sector number on a track. Sector 0 is not always the base sector.

PD_SSize **Sector Size**

Indicates the physical sector size in bytes. The default sector size is 256. Depending upon whether the driver supports non-256 byte logical sector sizes (that is, a variable sector size driver), the field is used as follows:

- **Variable Sector Size Driver**

  If the driver supports variable logical sector sizes, RBF inspects this value during a path open (specifically, after the driver returns "no error" on the SS_VarSect GetStat call) and uses this value as the *logical* sector size of the media. This value is then copied into PD_SctSiz of the path descriptor options section, so that applications programs can know the logical sector size of the media, if required. RBF supports logical sector sizes from 256 bytes to 32,768 bytes, in integral binary multiples (256, 512, 1024, etc.).

  During the SS_VarSect call, the driver can validate or update this field (or the media itself) according to the driver's conventions. These typically are:

  ¿ If the driver can dynamically determine the media's sector size, and PD_SSize is passed in as 0, the driver updates this field according to the current media setting.

  ¡ If the driver can dynamically set the media's sector size, and PD_SSize is passed in as a non-zero value, the driver sets the media to the value in PD_SSize (this is typical when re-formatting the media).

¬ If the driver cannot dynamically determine or set the media sector size, it usually validates PD_SSize against the supported sector sizes, and returns an error (E$SectSiz) if PD_SSize contains an invalid value.

*** These parameters are format specific.**

- **Non-Variable Sector Size Driver**
  If the driver does not support variable logical sector sizes (that is, logical sector size is fixed at 256 bytes), RBF ignores PD_SSize. In this case, PD_SSize can be used to support deblocking drivers that support various physical sector sizes.

  NOTE: A non-variable sector sized driver is defined as a driver which returns the E$UnkSvc error for GetStat (SS_VarSect).

PD_Cntl **Device Control Word**

Indicates options that reflect the capabilities of the device. These options may be set by the user, as follows:

bit 0: 0 = Format enable
1 = Format inhibit

bit 1: 0 = Single-Sector I/O
1 = Multi-Sector I/O capable

bit 2: 0 = Device has non-stable ID
1 = Device has stable ID

bit 3: 0 = Device size determined from descriptor values
1 = Device size obtained by SS_DSize GetStat call

bit 4: 0 = Device cannot format a single track
1 = Device can format a single track

bit 5-15: Reserved

Name	Description

PD_Trys

Number of Tries

Indicates whether a driver should try to access the disk again before returning an error. Depending upon the driver in use, this field may be implemented as a flag or a retry counter:

Value	Flag	Counter
0	retry ON	default number of retries
1	retry OFF	no retries
other	retry ON	specified number of retries

Drivers that work with controllers that have error correcting functions (for example, E.C.C. on hard disks) should treat this field as a flag so they can set the controller's error correction/retry functions accordingly.

When formatting media, especially hard disks, the format-enabled descriptor should set this field to one (retry OFF) to ensure that marginal media sections are marked out of the media free space.

PD_LUN

Logical Unit Number of SCSI Drive

Used in the SCSI command block to identify the logical unit on the SCSI controller. To eliminate allocation of unused drive tables in the driver static storage, this number may be different from PD_DRV. PD_DRV indicates the logical number of the drive to the driver, that is, the drive table to use. PD_LUN is the physical drive number on the controller.

PD_WPC

First Cylinder to Use Write Precompensation

Indicates the cylinder to begin write precompensation.

PD_RWR

First Cylinder to Use Reduced Write Current

Indicates the cylinder to begin reduced write current.

Name	Description
PD_Park	**Cylinder Used to Park Head** Indicates the cylinder at which to park the hard disk's head when the drive is shut down. Parking is usually done on hard disks when they are shipped or moved and is implemented by the SS_SQD SetStat to the driver.
PD_LSNOffs	**Logical Sector Offset** The offset to use when accessing a partitioned drive. The driver adds this value to the logical block address passed by RBF prior to determining the physical block address on the media. Typically, using PD_LSNOffs is mutually exclusive to using PD_TOffs.
PD_TotCyls	**Total Cylinders on Device** Indicates the actual number of physical cylinders on a drive. It is used by the driver to correctly initialize the controller/drive. PD_TotCyls is typically used for physical initialization of a drive that is partitioned or has PD_TOffs set to a non-zero value. In this case, PD_CYL denotes the *logical* number of cylinders of the drive. If PD_TotCyls is zero, the driver should determine the physical cylinder count by using the sum of PD_CYL and PD_TOffs.
PD_CtrlrID	**SCSI Controller ID** The ID number of the SCSI controller attached to the drive. The driver uses this number to communicate with the controller.
PD_ScsiOpt	**SCSI Driver Options Flags** Indicate the SCSI device options and operation modes. It is the driver's responsibility to use or reject these values, as applicable. bit 0: 0 = ATN not asserted (no disconnect allowed) 1 = ATN asserted (disconnect allowed) bit 1: 0 = Device cannot operate as a target 1 = Device can operate as a target bit 2: 0 = Asynchronous data transfer 1 = Synchronous data transfer bit 3: 0 = Parity off 1 = Parity on All other bits are reserved.

Name	Description
PD_Rate	**Data Transfer/Rotational Rate** Contains the data transfer rate and rotational speed of the floppy media. Note that this field is normally used only when the physical size field (PD_TYP, bits 1-3) is non-zero. bits 0-3: Rotational speed 0 = 300 RPM 1 = 360 RPM 2 = 600 RPM All other values are reserved. bits 4-7: Data transfer rate 0 = 125K bits/sec 1 = 250K bits/sec 2 = 300K bits/sec 3 = 500K bits/sec 4 = 1M bits/sec 5 = 2M bits/sec 6 = 5M bits/sec All other values are reserved.
PD_MaxCnt	**Maximum Transfer Count** Contains the maximum byte count that the driver can transfer in one call. If this field is 0, RBF defaults to the value of $ffff (65,535).

RBF Path Descriptor Definitions

The first 26 fields of the path options section (PD_OPT) of the RBF path descriptor are copied directly from the device descriptor standard initialization table. All of the values in this table may be examined using I$GetStt by applications using the SS_Opt code. Some of the values may be changed using I$SetStt; some are protected by the file manager to prevent inappropriate changes.

Refer to the previous section on RBF device descriptors for descriptions of the first 26 fields. The last five fields contain information provided by RBF:

Name	Description
PD_ATT	**File Attributes** The file's attributes are defined as follows: bit 0: Set if owner read. bit 1: Set if owner write. bit 2: Set if owner execute. bit 3: Set if public read. bit 4: Set if public write. bit 5: Set if public execute. bit 6: Set if only one user at a time can open the file. bit 7: Set if directory file.
PD_FD	**File Descriptor** The LSN (Logical Sector Number) of the file's file descriptor is written here.
PD_DFD	**Directory File Descriptor** The LSN of the file's directory's file descriptor is written here.
PD_DCP	**File's Directory Entry Pointer** The current position of the file's entry in its directory.
PD_DVT	**Device Table Pointer (copy)** The address of the device table entry associated with the path.
PD_SctSiz	**Logical Sector Size** The logical sector size of the device associated with the path. If this is 0, assume a size of 256 bytes.
PD_NAME	**File Name**

NOTE: In the following chart, the term ***offset*** refers to the location of a path descriptor field relative to the starting address of the path descriptor. Path descriptor offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Offset	Name	Description
$80	PD_DTP	Device Class
$81	PD_DRV	Drive Number
$82	PD_STP	Step Rate
$83	PD_TYP	Device Type
$84	PD_DNS	Density
$85		Reserved
$86	PD_CYL	Number of Cylinders
$88	PD_SID	Number of Heads/Sides
$89	PD_VFY	Disk Write Verification
$8A	PD_SCT	Default Sectors/Track
$8C	PD_TOS	Default Sectors/Track 0
$8E	PD_SAS	Segment Allocation Size
$90	PD_ILV	Sector Interleave Factor
$91	PD_TFM	DMA Transfer Mode
$92	PD_TOffs	Track Base Offset
$93	PD_SOffs	Sector Base Offset
$94	PD_SSize	Sector Size (in bytes)
$96	PD_Cntl	Control Word
$98	PD_Trys	Number of Tries
$99	PD_LUN	SCSI Unit Number of Drive
$9A	PD_WPC	Cylinder to Begin Write Precompensation
$9C	PD_RWR	Cylinder to Begin Reduced Write Current
$9E	PD_Park	Cylinder to Park Disk Head
$A0	PD_LSNOffs	Logical Sector Offset
$A4	PD_TotCyls	Number of Cylinders On Device
$A6	PD_CtrlrID	SCSI Controller ID
$A7	PD_Rate	Data Transfer/Rotational Rates
$A8	PD_ScsiOpt	SCSI Driver Option Flag
$AC	PD_MaxCnt	Maximum Transfer Count
$B0		Reserved
$B5	PD_ATT	File Attributes
Offset	**Name**	**Description**

$B6	PD_FD	File Descriptor
$BA	PD_DFD	Directory File Descriptor
$BE	PD_DCP	File's Directory Entry Pointer
$C2	PD_DVT	Device Table Pointer (copy)
$C6		Reserved
$C8	PD_SctSiz	Logical Sector Size
$CC		Reserved
$E0	PD_NAME	File Name

RBF Device Drivers

RBF reads and writes in logical blocks, called sectors. A logical sector can be any integral binary power from 256 to 32768. The file manager takes care of all file system processing and passes the driver a starting logical sector number (LSN), a sector count, and the address of the data buffer for each read or write operation.

The logical sector size of the media is determined by RBF when a path is opened to the device. RBF queries the driver to determine whether the driver can support variable sector sizes or not, using the SS_VarSect GetStat call.

If the driver supports variable sector size, RBF assumes that the logical and physical sector sizes are the same, with the size that is specified in PD_SSize.

If the driver does not support variable sector sizes, RBF assumes a logical sector size of 256 bytes, and ignores the value in PD_SSize. If the media physical sector size is not 256 bytes, it is the driver's responsibility to translate and deblock RBF LSNs into the media's LSNs. For example, if PD_SSize is set to 512, and a read request of eight sectors at LSN four is made, the driver should translate the operation into a read of four sectors at LSNtwo.

Read and write calls to the driver initiate the sector read/write operations and, if required, a prior seek operation.

If the controller cannot be interrupt-driven, it must wait until the media is ready, and then transfer the data by polling. If possible, avoid disk controllers that cannot be interrupt-driven. They cause the driver to dominate the system CPU while disk I/O is in progress.

For interrupt-driven systems, the driver initiates the I/O operation and suspends itself (F$Sleep or F$Event) until the interrupt arrives. The interrupt service routine then services the interrupt and "wakes up" the driver.

NOTE: If the driver is awakened by a signal (for example, a keyboard abort) while waiting for the I/O interrupt to occur, it should suspend itself again until the I/O interrupt has occurred. This is because many read/write calls to a driver are made by RBF on behalf of itself, such as in directory searching or bitmap updating. If a signal causes a process to terminate, RBF determines the appropriate time to return to the kernel. Failure to enforce the I/O interrupt completion may result in "locked" disks or corrupted media.

If DMA (Direct Memory Access) hardware support is available, I/O performance increases dramatically because the driver will not have to move the data between memory and the controller.

When the driver reads sector zero, it should copy the first 21 bytes of the sector into the drive table (PD_DTB) associated with the logical unit. Sector zero of the disk media has format information recorded by the format utility. This information allows the driver to determine the actual format of the media and to compare the device physical capabilities specified in the path descriptor options with the media format. This allows the driver to adapt its operation for reading and writing multiple formats in one physical drive. For example, a floppy drive that can read/write double-sided, double-density disks can be made to operate with single-sided or single-density media.

RBF always reads sector zero of the media when a file is opened. Many RBF drivers provide ***caching*** of sector zero to improve the performance of I$Open calls by RBF. This function is generally associated with media that is non-removable (for example, fixed hard disks). When a hard disk driver reads sector zero, it updates the drive table and copies the full sector zero into a local buffer. The state of the buffered sector for the unit is recorded in the logical unit drive table variables V_ZeroRd and V_ScZero. This enables the driver to return sector zero data on subsequent calls by RBF without accessing the disk. Removable media should not have sector zero buffered unless the driver is capable of automatically detecting the media removal (by an interrupt) and marking sector 0 unbuffered.

GetStat calls to RBF devices are generally processed by RBF itself, and thus are not normally seen by the driver. The main exception is the SS_VarSect call, which RBF uses to inquire about the driver's ability to support non-256-byte logical sectors.

The INIT and TERM routines of RBF drivers are called directly by the kernel when the device is attached and detached. Typically, the INIT routine only performs controller-specific initialization such as adding the controller to the IRQ polling table, setting default values in the drive tables, and initializing the controller hardware interface.

NOTE: The INIT routine generally does not perform initialization of the logical units attached to the controller, for example, disk parameter definitions for SCSI drives. This type of initialization should normally be done when the first Read/Write/GetStat/SetStat call is made to the unit.

The TERM routine typically disables the device's interrupts, if required, and removes the controller from the IRQ polling table.

Main Driver Types

The complexity of RBF drivers depends on the capabilities of the hardware involved. Simple hardware controllers require more effort by the driver than do intelligent controllers. Generally RBF drivers fall into three levels of complexity:

- **Simple Floppy Interfaces**
  These types of drivers perform all physical drive movement operations explicitly: seek head, wait for head settle delay, etc. They translate the RBF LSN into a track/head/sector, select the drive, move the disk head to the required position, and then issue the I/O command. If multiple drives are connected to the controller, the driver often has to maintain a record of the current head position of each drive.

- **Combined Hard/Floppy Interfaces**
  These types of drivers deal with "medium" intelligence controllers. Typically, the physical drive selection and automatic seeking are handled by the controller itself. The driver becomes somewhat simpler because it must only translate the RBF LSN into a track/head/sector value. The addition of hard disk operation to the driver adds some minor complexity to the driver due to the differences in floppy vs. hard disk operation.

- **Intelligent Controllers**
  These types of drivers are typically used with SCSI or similar style controllers. These controllers usually accept only a command "packet" indicating the operation required and the address of the operation. These drivers are similar to "medium" intelligence controllers, with the exception that the RBF LSN is usually accepted directly by the controller as the physical sector number.

RBF Device Driver Storage Definitions

RBF device driver modules contain a package of subroutines that perform sector-oriented I/O to or from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical I/O controllers.

The kernel allocates a static storage area for each device (which may control several drives). The size of the storage area is specified in the device driver module header (M$Mem). Some of this storage area is required by the kernel and RBF; the device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the rbfstat.a and drvstat.a DEFS files. Static storage is used as follows:

Offset	Name	Maintained By	Description
$00	V_PORT	Kernel	Device base port address
$04	V_LPRC	File Manager	Last active process ID
$06	V_BUSY	File Manager	Current active process
$08	V_WAKE	Driver	Process ID to awaken
$0A	V_PATHS	Kernel	Linked List of Open Paths
$2E	V_NDRV	Driver	Number of Drives
$2F			Reserved
$36	DRVBEG	Driver/File Man.	Drive Tables

NOTE: ***Offset*** refers to the location of a field, relative to the starting address of the static storage. Offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l.

Name	Description
V_PORT	**Device base port address** The device's physical port address. It is copied from M$Port in the device descriptor when the device is attached by the kernel.
V_LPRC	**Last active process ID** The process ID of the most recent process to use the device. This field is required by the kernel for all device driver static storage, but is not used by RBF.
V_BUSY	**Current active process** The process ID of the process currently using the device. It is used to implement I/O Blocking by RBF. This field is also used by the interrupt drivers when they wish to suspend themselves, by copying V_BUSY to V_WAKE (prior to suspending themselves). A value of zero indicates the device is not busy.
V_WAKE	**Process ID to awaken** The process ID of any process that is waiting for the device to complete I/O. A value of zero indicates that no process is waiting. V_WAKE is set by the driver from V_BUSY and provides the interlock between the driver and the driver's interrupt service routine.
V_PATHS	**Linked List of Open Paths** This is a singly-linked list of all paths currently open on this device.
V_NDRV	**Number of drives** This field is set by the driver's INIT routine to indicate the maximum number of logical drives the driver can use. RBF validates the logical drive number of the drive (PD_DRV) against this number prior to setting the drive table pointer (PD_DTB). PD_DRV must be less than V_NDRV.
V_DRVBEG	**Drive Tables** This section contains one table for each drive that the controller will handle. The drive table associated with the drive is indicated by the drive table pointer (PD_DTB) in the path descriptor.

Device Driver Tables

After the driver's INIT routine has been called, RBF requests the driver to read the identification sector (LSN 0) from the drive. After reading sector zero, the driver must initialize the corresponding drive table. It does this by copying the number of bytes specified by DD_SIZ (21) from the beginning of sector 0 into the appropriate table (PD_DTB). The following is the format of each drive table:

Offset	Name	Maintained By	Description
$00	DD_TOT	Sector 0	Total Number of Sectors
$03	DD_TKS	Sector 0	Track Size (in sectors)
$04	DD_MAP	Sector 0	Number of Bytes in Allocation Map
$06	DD_BIT	Sector 0	Number of Sectors/Bit (cluster size)
$08	DD_DIR	Sector 0	LSN of Root Directory FD
$0B	DD_OWN	Sector 0	Owner ID
$0D	DD_ATT	Sector 0	Attributes
$0E	DD_DSK	Sector 0	Disk ID
$10	DD_FMT	Sector 0	Disk Format: Density/Sides
$11	DD_SPT	Sector 0	Sectors/Track
$13	DD_RES		Reserved
$16	V_TRAK	Driver	Current Track Number
$18	V_FileHd	File Manager	Open File List for Disk
$1C	V_DiskID	File Manager	Disk ID
$1E	V_BMapSz	File Manager	Bitmap Size
$20	V_MapSct	File Manager	Lowest Bitmap Sector to Search
$22	V_BMB	File Manager	Bitmap In Use Flag
$24	V_ScZero	Driver	Pointer to Sector 0
$28	V_ZeroRd	Driver	Sector 0 Read Flag
$29	V_Init	Driver	Drive Initialized Flag
$2A	V_Resbit	File Manager	Reserved Bitmap Sector Number
$2C	V_SoftEr	Driver	Number of Recoverable Errors
$30	V_HardEr	Driver	Number of Non-Recoverable Errors
$34	V_Cache	Cache Utility	Drive Cache Queue Head
$38	V_DText	Driver	Drive Table Extension pointer
$3A	V_MapMax	File Manager	Maximum Bitmap Sector Number
$3C			Reserved (22 bytes)

NOTE: There must be as many tables as are specified in V_NDRV. All references to Sector 0 in the **Maintained By** column mean that this field is initialized by the driver with information obtained from Sector 0 when it is first read.

Name	Description
DD_TOT	**Total Number of Sectors** Contains the size of the media in sectors. RBF uses this field to set the size of the "raw" device file ("@" file opens). The driver can also use this value to verify that the LSN passed by RBF is in range for the media. Driver INIT routines typically initialize this field in the drive table to a non-zero value, so that sector 0 may be read initially.
DD_TKS	**Track Size (in sectors)** Contains the number of sectors per track, as a byte value.
DD_MAP	**Number of Bytes in Allocation Map** Contains the size of the media bitmap.
DD_BIT	**Number of Sectors/Bit (cluster size)** Contains the size of a cluster of sectors on the disk. This value is always an integral power of two.
DD_DIR	**LSN of Root Directory FD** Contains a pointer to the file descriptor of the media's root directory.
DD_OWN	**Owner ID** The user ID of the disk owner.
DD_ATT	**Attributes** Defines the access attributes of the media.
DD_DSK	**Disk ID** Contains a pseudo-random number which identifies the media volume. This number is put here by the format utility.
DD_FMT	**Disk Format: Density/Sides** Defines the format of the media volume, to enable drivers to adapt to different formats: bit 0: 0 = Single-sided 1 = Double-sided bit 1: 0 = Single-density (FM) 1 = Double-density (MFM) bit 2: 1 = Double-track density (96 TPI/135 TPI) bit 3: 1 = Quad track density (192 TPI) bit 4: 1 = Octal track density (384 TPI)

Name	Description
DD_SPT	**Sectors/Track** A two byte value of DD_TKS.
V_TRAK	**Current Track Number** This value is used to record the current track number of a logical unit for those drivers that need to perform seek functions explicitly. Typically, driver INIT routines initialize this field to an unknown track number (for example, $FF), so that the first access to the drive results in a restore operation.
V_FileHd	**Open File List for Disk** A pointer to the list of all files open on the logical unit.
V_DiskID	**Disk ID** A copy of DD_DSK.
V_BMapSz	**Bitmap Size** The size of the media's bitmap.
V_MapSct	**Lowest Bitmap Sector to Search** The starting sector number to begin bitmap allocation functions.
V_BMB	**Bitmap In Use Flag** Indicates whether or not the bitmap is in use.
V_ScZero	**Pointer to Sector 0** A pointer to a buffered sector zero for the unit. This is only used by drivers that perform this function.
V_ZeroRd	**Sector 0 Read Flag** Used by the driver to indicate whether or not the buffered sector zero is valid. If the data is valid, this flag should be non-zero.
V_Init	**Drive Initialized Flag** Used by the driver to indicate whether or not the logical unit has been initialized. If the unit has been initialized, this field should be non-zero.
V_Resbit	**Reserved Bitmap Sector Number** Indicates the bitmap sector number to ignore during RBF bitmap allocation functions. It is set by the SS_RsBit SetStat call.

Name	Description
V_SoftEr	**Number of Recoverable Errors** Allows the driver to keep a count of "soft" errors during I/O operations. The value is typically returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
V_HardEr	**Number of Non-Recoverable Errors** Allows the driver to keep a count of "hard" errors during I/O operations. The value would typically be returned by a SS_ELog GetStat call. After reading this value, it is typically reset to zero.
V_Cache	**Drive Cache Queue Head** A pointer to the cache queue for the drive.
V_DTExt	**Drive Table Extension Pointer** A pointer to an extension of the drive table. Drivers that require storage of additional drive table variables can use this field as a pointer to the extra information.
V_MapMax	**Maximum Bitmap Sector Number** The sector number of the last sector of the bitmap.

Linking RBF Drivers

After a RBF driver has been assembled into its relocatable object file (ROF), the driver needs to be linked to produce the final driver module. Linking resolves all code references in drivers that are comprised of several ROF files. It also resolves the external data and static storage references by the driver.

The most important part of linking is to correctly resolve the static storage references. Generally, the static storage area is composed of three sections in this order (see Figure 2-1):

- I/O globals
- Drive tables (one per logical drive)
- Driver-declared variables

The driver-declared variables are declared in vsect areas of the driver, but they *must* be allocated after the drive table storage areas. The method that you must use to allocate all of the storage, *in the correct order*, is to link one of the drvsX.l library files *before* the user written ROF files. The drvsX.l files are usually found in the system's LIB directory. Each drvsX.l file contains vsect declarations that allocate the I/O system variables and the appropriate number of drive tables. For example, drvs1.l allocates the I/O system-defined section and one drive table, while drvs4.l allocates the I/O system-defined section and four drive tables. The following is a typical linker command line for an RBF driver:

```
l68 /dd/LIB/drvs4.l REL/rb320.r -O=OBJS/rb320
```

NOTE: Specifying the drvsX.l file first causes the vsect variables declared by the file to be allocated *before* the vsect variables in the ROF file. Failure to correctly allocate the I/O system and drive table variables first, or failure to link the correct number of drive tables at all, results in erratic driver operation.

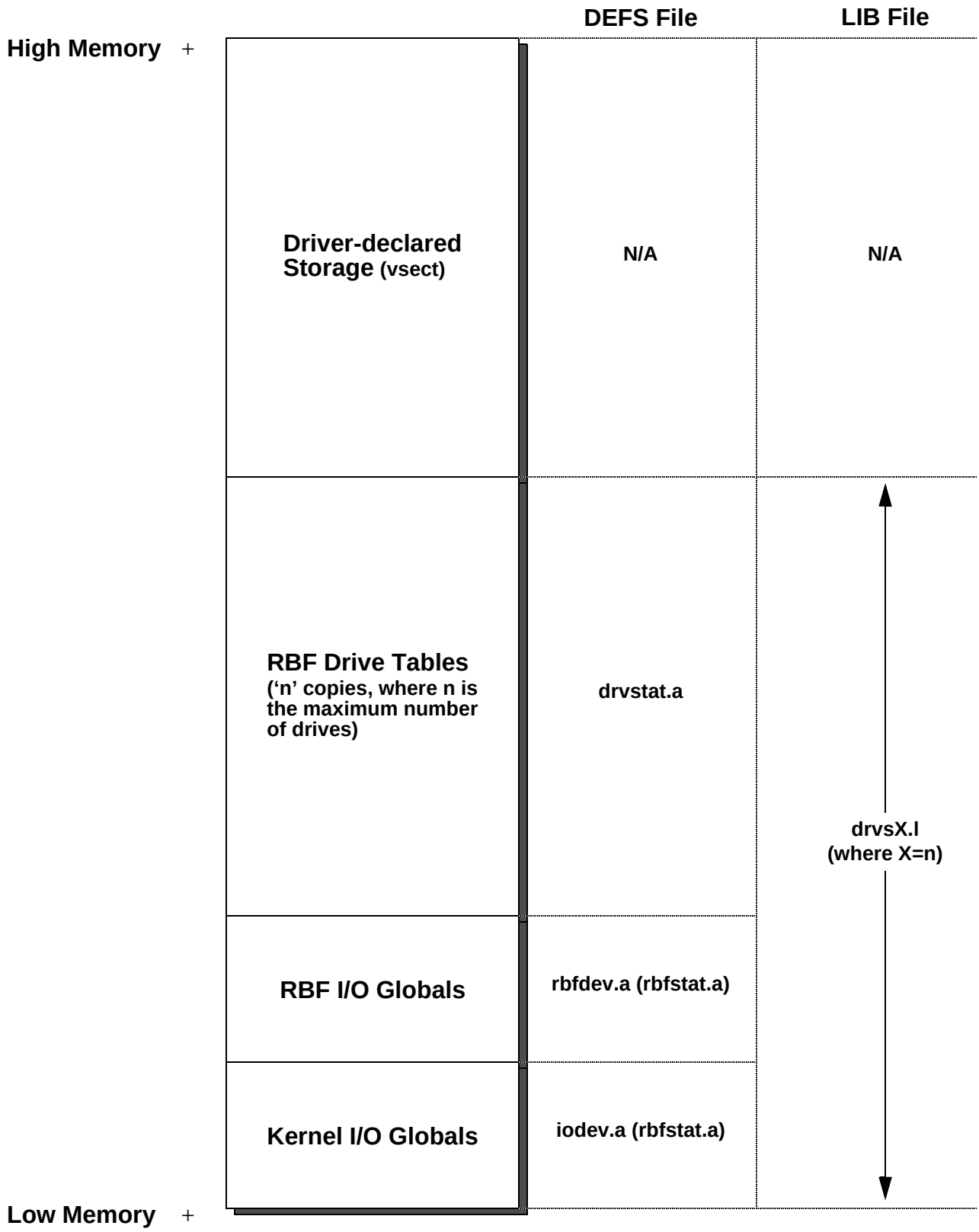

Figure 2-1: RBF Static Storage Layout

RBF Device Driver Subroutines

As with all device drivers, RBF device drivers use a standard executable memory module format with a module type of Drivr (code $E0). RBF drivers are called in system state.

NOTE: I/O system modules must have the following module attributes:

- They must be owned by a super-user (0.n).
- They must have the system-state bit set in the attribute byte of the module header. (OS-9 does not currently make use of this, but future revisions will require that I/O system modules be system-state modules.)

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of a corresponding subroutine. The branch table appears as follows:

```
ENTRY dc.w    INIT      initialize device
      dc.w    READ      read character
      dc.w    WRITE     write character
      dc.w    GETSTAT   get device status
      dc.w    SETSTAT   set device status
      dc.w    TERM      terminate device
      dc.w    TRAP      handle illegal exception (0 = none)
```

Each subroutine should exit with the carry bit of the condition code register cleared, if no error occurred. Otherwise, the carry bit should be set and an appropriate error code returned in the least significant word of register d1.w.

The TRAP entry point is currently not used by the kernel, but in the future will be defined as the offset to error exception handling code. Because no handler mechanism is currently defined, this entry point should be set to zero to ensure future compatibility.

The following pages describe each subroutine.

INIT — Initialize Device and its Static Storage Area

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The INIT routine must:

¨ Initialize the device's permanent storage. Minimally, this consists of:

- Initializing V_NDRV to the number of drives with which the controller will work.
- Initializing DD_TOT in each drive table to a non-zero value so that sector zero may be read or written to.
- If the driver must perform explicit seeks, initializing V_TRAK to $FF so that the first seek will find track zero.

¦ Place the IRQ service routine on the IRQ polling list by using the F$IRQ system call.

Æ Initialize device control registers (enable interrupts if necessary).

Prior to being called, the device static storage is cleared (set to zero), except for V_PORT which contains the device address. The driver should initialize each drive table entry appropriately for the type of disk the driver expects to be used on the corresponding drive.

If INIT returns an error, it does not have to clean up its operation, for example, remove device from polling table or disable hardware. The kernel calls TERM to allow the driver to clean up INIT's operation before returning to the calling process.

Usually, the INIT routine should only perform controller-specific initialization, as opposed to drive-specific initialization. This is because the controller may have more than one type of drive connected to it.

NOTE: If the INIT routine causes an interrupt to occur, you can handle the interrupt in one of the following ways:

- Process the interrupt directly by masking interrupts to the level of the device, polling/servicing the device hardware, and then restoring the previous interrupt level. This is the preferred technique unless the interrupt is time-consuming.

- Allow the interrupt service routine to service the hardware. In this case, the process descriptor contains the process ID (P$ID) to which V_WAKE should be set. V_BUSY cannot be used because it is zero when INIT is called.

READ — Read Sector(s)

INPUT: **d0.l = number of contiguous sectors to read**
d2.l = disk logical sector number to read
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT: **Sector(s) returned in the sector buffer**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The READ routine must perform the following operations:

¿ Locate the associated drive table (PD_DTB) and determine if it is initialized. If not, perform any drive initialization required and mark the drive initialized in the drive table. If the driver will perform sector zero buffering for the unit, allocate a sector zero buffer.

¡ Verify the starting LSN and ending LSN (if a multi-sector read) against the size of the media (DD_TOT).

¬ Compute the physical disk address (track/head/sector) from the LSN, if required.

Ð If the driver supports sector 0 buffering, and the read request is for sector 0, return the sector 0 data to the buffer specified. If no further sectors are requested, return to RBF. Otherwise, proceed to read the remaining sectors into the remainder of the buffer.

× For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require that you save the current track position in the last selected drive's drive table (V_TRAK).

± Prepare the hardware for the read request and start the I/O operation. The data should be read into the buffer specified by PD_BUF.

ð Wait for the I/O operation to complete (with interrupts, if possible).

Š If the starting LSN of the read was not LSN 0, return to RBF. Otherwise:

a) Update the unit's drive table by copying the number of bytes specified by DD_SIZ (21) from the beginning of sector 0 into the appropriate table.

b) If the driver supports buffering sector zero for the unit, copy sector zero into the driver's local buffer (V_ScZero) and mark the buffer valid (V_ZeroRd).

¥ If the logical unit and driver support multiple disk formats, the driver should validate that the media is readable by the drive. If not, the driver should return a Bad Type error (E$BTyp). If it can, the driver should ready itself for the new format by either:

a) Marking the logical unit as uninitialized (V_Init cleared), so the next access will cause the unit to be re-initialized by the driver.

b) Re-initializing the unit hardware for the new format.

µ Return the status of the read to RBF.

Sector/Transfer Count

The number of sectors to transfer is passed by RBF. If bit number one in PD_Cntl is clear, RBF always requests only one sector. If the bit is set, RBF requests a maximum count, based on the value in PD_MaxCnt. The value in PD_MaxCnt is truncated to an exact sector count, so that the device always sees requests in terms of an integral number of sectors.

Sector Zero Reads

Whenever logical sector zero is read from the media, the first part of it must be copied into the drive table for the logical unit. PD_DTB contains the pointer to the drive table. The number of bytes to copy is DD_SIZ.

Drivers that buffer sector zero also update their local copy when sector zero is read from the media. The drive table variables V_ScZero (pointer to sector zero) and V_ZeroRd (sector zero valid flag) allow the driver to maintain this buffer. When the driver receives a read request for LSN zero, it can check these flags. If the buffer is valid, it can simply return the buffered data to RBF without performing any disk I/O.

Sector zero buffering should normally be performed only on fixed media (fixed hard disks). This ensures that media volume changes are noticed by RBF. Failure to detect media changes correctly can result in corruption of the new volume.

If the driver can detect media removal (for example, via an interrupt when the door is opened), it is permissable for the driver to buffer sector 0 while the media is installed.

Sector Size Support

If the driver supports variable sector sizes, RBF assumes that the size of a sector is specified by PD_SSize, and that the logical and physical sector sizes are the same. Drivers operating under this mode simply process the RBF transfer count and LSN address according to the disk's requirements.

If the driver does not support variable sector sizes (logical sector size is 256 bytes) and the physical sector size of the media (PD_SSize) is not 256 bytes, the driver must deblock the media sectors. Typically, this involves the following steps:

- Determine if RBF's starting LSN falls at the start of a media physical sector. If not, check if the physical sector is currently buffered by the driver. If the physical sector is currently buffered by the driver, copy the appropriate part of the buffer to RBF's buffer. If not, read the physical sector into the driver's buffer and return the appropriate part to RBF's buffer.
- If any sectors remain to be read, convert the remaining start address and count into the physical start address and count. Then, read (and count) those sectors into the RBF buffer.
- If any partial sector remains to be read, read that physical sector into the driver's physical buffer. Then, return the appropriate part of the buffer to the end of the RBF buffer.

Interrupt-driven Operation

If the hardware uses interrupts to perform I/O, the driver should perform the following steps:

Synchronization using Signals

¨ Issue the I/O command to the hardware.

¦ Copy V_BUSY to V_WAKE in the static storage.

Æ The driver should then suspend itself (F$Sleep).

Ø The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and sends a wake-up signal (S$Wake) to the driver. The driver's process ID is in V_WAKE. After sending the signal, the IRQ service routine should clear V_WAKE to signify that the interrupt occurred.

× When the driver awakens, it should check V_WAKE. If zero, the interrupt has occurred and the driver can continue to check status, etc. If non-zero, the driver should suspend itself again.

Synchronization using Events

¨ Issue the I/O command to the hardware.

¦ The driver should suspend itself using the event system's "wait" function.

Æ The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and then uses the event system's "signal" function to awaken the driver.

Ø When the driver awakens, it should determine if the event value is within range. If so, the interrupt was serviced and the driver can check the status, etc. If not, the driver should suspend itself again.

WRITE — Write Sector(s)

INPUT: **d0.l = number of contiguous sectors to write**
d2.l = disk logical sector number
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT: **The sector buffer is written to disk.**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: The WRITE routine must perform the following operations:

- ¨ Determine the starting LSN. If zero, the driver should check the format control flag for format protection (PD_Cntl, bit 0). If bit 0 is clear, the media can be formatted and sector 0 may be written. If bit 0 is set, the media is format protected and the driver should return an E$Format error.
- ¡ Locate the associated drive table (PD_DTB) and check if the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.
- ¬ If the driver supports buffering of sector 0 for the unit, and sector 0 is being written, the driver should clear V_ZeroRd to mark that sector 0 is unbuffered.
- Ø Verify the starting LSN (and ending LSN, if a multi-sector write) against the size of the media (DD_TOT).
- × Compute the physical disk address (track/head/sector) from the LSN, if required.
- ± For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require you to save the current track position in the last selected drive's drive table (V_TRAK).
- ð Prepare the hardware for the write request and start the I/O operation. The data should be written from the buffer specified by PD_BUF.
- Š Wait for the I/O operation to complete (with interrupts, if possible).
- ¥ Return the status of the write to RBF.

Sector/Transfer Count

The number of sectors to transfer is passed by RBF. If bit number one in PD_Cntl is clear, RBF always requests only one sector. If the bit is set, RBF requests a maximum count, based on the value in PD_MaxCnt. The value in PD_MaxCnt is truncated to an exact sector count, so that the device always sees requests in terms of an integral number of sectors.

Sector Zero Writes

Whenever the starting LSN is zero, the driver should check whether the media may be formatted (PD_Cntl, bit 0). If bit 0 is set, the media is format protected and sector zero may not be written. The driver should return a E$Format (format protected) error in this case.

If the driver buffers sector zero of the media, it should clear V_ZeroRd to mark the buffer invalid. This ensures that the next read of sector zero will access the media.

Sector Size Support

If the driver supports variable sector sizes, RBF assumes that the size of a sector is specified by PD_SSize, and that the logical and physical sector sizes are the same. Drivers operating under this mode simply process the RBF transfer count and LSN address according to the disk's requirements.

If the driver does not support variable sector sizes (logical sector size is 256 bytes) and the physical sector size of the media (PD_SSize) is not 256 bytes, the driver must deblock the media sectors. Typically, this involves the following steps:

- Determine if RBF's starting LSN falls at the start of a media physical sector. If not, and the physical sector is not currently cached, read the physical sector into the driver's local buffer. Update the appropriate part of the buffer with RBF's data and write the local buffer to the media.
- If any sectors remain to be written, convert the remaining start address and count into the physical start address and count. Then, write (and count) those sectors from the RBF buffer.
- If any partial sector remains to be written, read that physical sector into the driver's local buffer. Next, update the appropriate part of the buffer with RBF's data and write the local buffer to the media.

Interrupt Operation

If the hardware uses interrupts to perform I/O, the driver should perform the following steps:

Synchronization using Signals

¨ Issue the I/O command to the hardware.

¦ Copy V_BUSY to V_WAKE in the static storage.

Æ The driver should suspend itself (F$Sleep).

Ø The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and sends a wake-up signal (S$Wake) to the driver. The driver's process ID is in V_WAKE. After sending the signal, the IRQ service routine should clear V_WAKE to signify that the interrupt occurred.

× When the driver awakens, it should check V_WAKE. If zero, the interrupt has occurred and the driver can continue to check status, etc. If non-zero, the driver should suspend itself again.

Synchronization using Events

¨ Issue the I/O command to the hardware.

¦ The driver should suspend itself using the event system's "wait" function.

Æ The IRQ service routine is called when the interrupt occurs. The IRQ service routine checks that the interrupt occurred for its hardware, services the interrupt, and then uses the event system's "signal" function to awaken the driver.

Ø When the driver awakens, it should check that the event value is within range. If so, the interrupt was serviced and the driver can check the status, etc. If not, the driver should suspend itself again.

GETSTAT/SETSTAT Get/Set Device Status

INPUT: **d0.w = status code**
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer

OUTPUT: **Depends on the function code**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: These routines are wild-card calls used to get/set the device's operating parameters as specified for the I$GetStt and I$SetStt service requests.

Calls which involve parameter passing require the driver to examine or change the register stack variables. These variables contain the contents of the MPU registers at the time of the I$Getstt/I$SetStt request was made. Parameters passed to the driver are set up by the caller prior to using the service call. Parameters passed back to the caller are available when the service call completes.

Typical RBF drivers handle the following I$GetStt/I$SetStt calls:

I$GetStt: SS_DSize, SS_VarSect

I$Setstt: SS_Reset, SS_SQD, SS_WTrk

Any unsupported I$GetStt/I$SetStt calls to the driver should return an unknown service error (E$UnkSvc).

NOTE: A minimal RBF driver should support SS_Reset and SS_WTrk, so that media may be formatted.

The following pages describe the driver implementation of the above I$GetStt/I$SetStt calls.

GetStat Call:

SS_DSize This routine is used to return the media size for autosize devices (PD_Cntl, bit three set). The routine must perform the following steps:

¿ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.

¡ Prepare the hardware for the request and start the I/O operation.

¬ Wait for the I/O operation to complete (with interrupts, if possible).

Ð Return the media size (in terms of its logical sector size) to the caller's d2 register (R$d2 offset from passed a5). Note that if the driver supports deblocking (logical and physical sizes are not the same), the returned sector count should be a "logical" sector count.

f Return status to RBF.

SS_VarSect This routine is called by RBF whenever a path is opened to the device, so that RBF can determine the logical sector size of the media. The driver should indicate its support for variable logical sector sizes as follows:

- If variable logical sector sizes are supported , the driver should return a "no error" status. Upon return to RBF, RBF uses the value in PD_SSize as the media's logical sector size. It is permissable for the driver to query the drive for its current sector size setting and update PD_SSize during this call.

WARNING: Querying the drive *does not* mean issuing a physical read of the disk's sector 0 (to read DD_LSNSize) as RBF has not yet set up the buffer pointers for the path (PD_BUF = 0). Unless you take special care, attempting to perform physical data I/O at this point will probably crash the system. The only type of I/O operations valid at this point are generally internal driver operations (for example, Mode Sense command to a SCSI drive). Drivers that deal with media that cannot return "current sector size" generally require that PD_SSize be set correctly in the device descriptor. The driver returns "no error" to indicate that RBF can use PD_SSize as the logical media size.

- If the driver does not support variable logical sector sizes, it should return an "unknown service request" (E$UnkSvc) error, to indicate to RBF that the logical sector size of the media is 256 bytes and that PD_SSize should be ignored.
- If the driver returns any error other than "unknown service request", RBF aborts the path open operation and returns the error to the caller.

SetStat Calls:

SS_Reset Recallibrate (restore) the media head to the outer track. This is mainly used by format to ensure the media is at a known position.

The restore routine must perform the following functions:

¿ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform any drive initialization required and mark the drive initialized in the drive table.

¡ Prepare the hardware for the request and start the I/O operation.

¬ Wait for the I/O operation to complete (with interrupts, if possible).

Ð Return the status of the restore to RBF.

SS_SQD This is mainly used to move (park) the heads of hard disk drives to a safe area. The park routine must perform the following steps:

¨ Check whether the media may be parked. This typically involves the following:

- Check if the device is a floppy disk. If so, return an E$UnkSvc error.
- Check the PD_Park value. If it is zero or within the range of the RBF media area, return an E$UnkSvc error.

¡ Locate the associated drive table (PD_DTB) and initialize the drive according to the parking function. This typically involves setting the drive's cylinder count to the PD_Park value. After initialization, *do not* mark the drive initialized (V_Init should be clear). This ensures that any subsequent accesses to the drive will cause the drive to be re-initialized correctly (PD_CYL or PD_TotCyls count instead of PD_Park).

¬ Prepare the hardware for the park request and start the I/O operation.

Ð Wait for the I/O operation to complete (with interrupts, if possible).

f Return the status of the park to RBF.

The park operation typically consists of issuing a seek or read command and specifying a sector address on the desired cylinder. On some drives/controllers, this may fail because the parking cylinder is not formatted and the controller attempts to verify the seek/read. In these situations, it is typical for the driver to perform a write track operation on the desired track.

SS_WTrk This is used by format to perform physical initialization of the media. The write track routine must perform the following steps:

¨ Check whether the media may be formatted (PD_Cntl, bit 0 clear). If not, the media is format protected and the driver should return an E$Format error.

¡ Locate the associated drive table (PD_DTB) and check whether the unit is initialized (V_Init). If not, perform the required drive initialization and mark the drive initialized in the drive table. If the driver supports buffering sector 0 for the unit, and the track being formatted is the first track of the media (PD_TOffs), the driver should clear V_ZeroRd to mark that sector 0 is unbuffered.

¬ If the driver supports any buffering of physical sectors (non "VarSect" driver with physical sectors not equal to 256 bytes), it should mark any active buffers as invalid.

Ð For drivers that perform explicit seeking, seek to the desired track. If the seek involves the selection of a drive different from the last one selected, this may also require the current track position to be saved in the last selected drive's drive table (V_TRAK).

f Prepare the hardware for the write track request and start the I/O operation.

Ý Wait for the I/O operation to complete (with interrupts, if possible).

ý Return the status of the write track to RBF.

The method of formatting disk drives varies with the hardware in use. However, note the following points:

¨ The parameters passed are physical parameters, with one exception: the sector interleave table. If the driver must pass the interleave table to the hardware (or prepare its own table), it must add the PD_SOffs value to each interleave table entry so that a physical interleave table is passed to the hardware.

¦ The driver typically only initializes the drive when the track number passed is equal to the PD_TOffs value (that is, at the beginning of the format operation).

Æ SS_WTrk calls to the driver issued by format are dependent on the autosize flag in PD_Cntl (bit three) in the following manner:

- If the media is *autosize capable* (bit three set), format makes only one SS_WTrk call to the driver with the passed track number being equal to PD_TOffs. The driver is expected to format the entire media from this call.
- If the media is *non-autosize capable* (bit three clear), format issues a SS_WTrk call for each track on the media (PD_CYLS x PD_SID). The driver is expected to format the media one track at a time. If the hardware cannot handle individual tracks, the driver must perform a *format all media* operation on the first SS_WTrk call (PD_TOffs equal to the passed track number and side number zero) and simply ignore all other SS_WTrk calls without returning an error.

TERM — Terminate Device

INPUT: **(a1) = address of the device descriptor module**
(a2) = address of device static storage area
(a6) = system global static storage pointer

OUTPUT: **None**

ERROR OUTPUT: **cc = carry bit set**
d1.w = error code

FUNCTION: This routine is called when a device is no longer in use in the system (see I$Detach).

The TERM routine must:

- Wait until any pending I/O has completed.
- Disable the device interrupts.
- Remove the device from the IRQ polling list.
- Return any buffers the driver has requested on behalf of itself, for example, sector zero buffers or physical sector deblocking buffers.

  NOTE: The driver should not attempt to return buffers within its defined static storage area. The kernel releases this memory when the TERM routine completes.

NOTE: If an error occurs during the device's INIT routine, the kernel calls the TERM routine to allow the driver to clean up. If the TERM routine uses static storage variables (for example, interrupt mask values, dynamic buffer pointers), it should validate these variables prior to using them. The INIT routine may not have set up all the variables prior to exiting with the error.

IRQ Service Routine — Service Device Interrupts

INPUT: **(a2) = static storage address**
(a3) = port address
(a6) = system global static storage

OUTPUT: **None**

ERROR OUTPUT: **cc = carry set (interrupt not serviced)**

FUNCTION: This routine is called directly by the kernel's IRQ polling table routines. Its function is to:

- ¨ Check the device for a valid interrupt. If the device does not have an interrupt pending, the carry bit must be set and the routine exited with an RTS instruction as quickly as possible. Setting the carry bit signals the kernel that the next device on the vector should have its IRQ service routine called.

- ¦ Service device interrupts.

- Æ Wake up the driver mainline, using the synchronization method of the driver:

  Signals: Send a wake-up signal to the process whose process ID is in V_WAKE, when the I/O is complete. Also, clear V_WAKE as a flag to the mainline program that the IRQ has occurred.

  Events: Signal the event that the IRQ has occurred, using the event system's signal function.

- Ø Clear the carry bit and exit with an RTS instruction after servicing an interrupt.

Avoid exception conditions (for example, a Bus Error) when IRQ service routines are executing. Under the current version of the kernel, an exception in an IRQ service routine will crash the system.

NOTE: IRQ service routines may destroy the contents of the following registers only: d0, d1, a0, a2, a3, and a6. You must preserve the contents of all other registers or unpredictable system errors (system crashes) will occur.

End of Chapter 2

NOTES