RadiSys.

OS-9 Technical I/O Manual

Version 4.2

www.radisys.com

Revision A • April 2003

Contents

1 The OS-9 Input/Output System

This chapter explains the software components of the OS-9 I/O system and the relationships between those components. It includes the following topics:

- The OS-9 Unified Input/Output System
- IOMAN
- Device Descriptor Modules
- Path Descriptors
- Access Modes and Permissions
- File Managers
- Device Driver Modules

The OS-9 Unified Input/Output System

OS-9 features a versatile, unified, hardware-independent I/O system. The I/O system is modular and can easily be expanded or customized.

The I/O subsystem consists of three modules processing I/O service requests at different levels:

- The I/O Manager
- The File Manager
- The Device Driver

A fourth module, the device descriptor, contains the information used to assemble the different components of an I/O subsystem. The file manager, device driver, and device descriptor modules are standard memory modules you can install and remove dynamically while the system is running.

The I/O Manager

IOMAN manages the following four tasks:

- Supervise the OS-9 I/O system.
- Establish the connections between itself, the file manager, and the device driver.
- Manage various data structures.
- Ensure the appropriate file manager and device driver modules process a particular I/O request.

The File Manager

A file manager performs the processing for a particular class of devices such as disks or terminals. For example, the Random Block File Manager (RBF) maintains directory structures on disks and the Sequential Character File manager (SCF) edits the data stream it receives from terminals.

The Device Driver

A device driver has the following three primary tasks:

- Enable OS-9 to be device-independent.
- Operate on the actual hardware device, sending data to and from the device on behalf of the file manager.
- Isolate the file manager from actual hardware dependencies, such as control register organization and data transfer modes.

IOMAN

When the kernel receives an I/O request, it immediately passes the request to IOMAN. IOMAN provides the first level of service for I/O system calls by routing data between processes and the appropriate file managers and device drivers. IOMAN also allocates and initializes global static storage on behalf of file managers and device drivers.

Many controllers, such as SCSI interfaces and DUARTs (Dual Asynchronous Receiver-Transmitters), operate multiple devices. IOMAN allocates and initializes an additional static storage for each device called “logical unit static storage”; IOMAN uses the storage to assist file managers and drivers with managing these interfaces.

IOMAN maintains two important internal data structures:

- device list
- path table

These tables reflect two other structures respectively:

- device descriptor
- path descriptor

When an `_os_attach()` system call is first performed on a new device descriptor, IOMAN creates a new entry in the device list. Each entry in the device list contains information about each element required to perform I/O on a device.

A device list entry also contains pointers to the various static storages and other data elements in use on the device. The structure definition of a device list entry is defined in the header file `io.h`.

When a path is opened, IOMAN links to the device descriptor associated with the device name specified (or implied) in the pathlist. The device descriptor contains the names of the device driver and file manager for the device. IOMAN saves the information in the device entry list of the device descriptor, so subsequent system calls can be routed to these modules.

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

Device Descriptor Modules

A device descriptor module is a small, non-executable module providing information that associates a specific I/O device with the following:

- logical name
- hardware controller address(es)
- device driver name
- file manager name
- initialization parameters

Device drivers and file managers operate on general classes of devices, not specific I/O ports. A device descriptor tailors its functions to a specific I/O port.

The name of the device descriptor is used as the logical device name by the system and user (it is the device name given in pathlists). Its format consists of a standard module header with a type code of device descriptor (`MT_DEVDESC`).

One device descriptor must exist for each I/O device in the system. However, one device can also have several device descriptors with different initialization constants.

The device descriptor contains a constant table and logical unit static storage initialization information. IOMAN initializes logical unit static storage with the `_os_initdata()` system call, similar to how other processing elements in the system initialize their static storage areas. IOMAN does not restrict the definition or use of logical unit static storage.

A constant table containing information provided by a device descriptor is located at the entry point offset of the device descriptor. IOMAN requires the first part to be common to all device descriptors. File managers and device drivers may add information they require after the common part. The format of the common part is shown here and defined in the header file `io.h`.

Data defined by specific file managers is provided in the OS-9 *Device Descriptor and Configuration Module Reference*.

dd_com

Declaration

```
/* Device descriptor data definitions */
typedef struct {
    void                        *dd_port;   /* device port address */
    u_int16                     dd_lu_num,  /* logical unit number */
                                dd_pd_size, /* path descriptor size */
                                dd_type, /* device type */
                                dd_mode; /* device mode capabilities */
    u_int32                     dd_fmgr, /* file manager name offset */
                                dd_drvr; /* device driver name offset */
    u_int16                     dd_class, /* sequential or random */
                                dd_dscres;/* (reserved) */
} *Dd_com, dd_com;
```

Fields

`dd_port`
: `dd_port` represents the absolute physical address of the hardware controller.

`dd_lu_num`
: Distinguish the different devices driven from a unique controller. Each unique number represents a different logical unit static storage area.

`dd_pd_size`
: `dd_pd_size` is the size of the path descriptor. Path descriptors vary in size. IOMAN uses this value when it allocates a path descriptor.

`dd_type`
: Identify the I/O type of the device. The following values are defined in the header file `io.h`:

Table 1-1. I/O Type Values

Defined Name	Value	Description
`DT_SCF`	0	Sequential Character File Type
`DT_RBF`	1	Random Block File Type
`DT_PIPE`	2	Pipe File Type
`DT_SBF`	3	Sequential Block File Type
`DT_NFM`	4	Network File Type
`DT_CDFM`	5	Compact Disc File Type
`DT_UCM`	6	User Communication Manager

Table 1-1. I/O Type Values (Continued)

Defined Name	Value	Description
DT_SOCK	7	Socket Communication Manager
DT_PTTY	8	Pseudo-Keyboard Manager
DT_GFM	9	Graphics File Manager
DT_PCF	10	PC-DOS File Manager
DT_NRF	11	Non-volatile RAM File Manager
DT_ISDN	12	ISDN File Manager
DT_MPFM	13	MPFM File Manager
DT_RTNFM	14	Real-Time Network File Manager
DT_SPF	15	Stacked Protocol File Manager
DT_INET	16	Inet File Manager
DT_MFM	17	Multi-media File Manager
DT_DVM	18	Generic Device File Manager
DT_NULL	19	Null File Manager
DT_DVDFM	20	DVD File Manager
DT_MODFM	21	Module Directory File Manager

DT-codes up to 127 are reserved for Microware use only.

`dd_mode`

Device mode capabilities.

IOMAN and the various file managers use the bits in this field to determine the modes of which a device is capable. For example, if no read related bits are set for an SCF device, reading will not be permitted from the device. IOMAN has only one check of `dd_mode`. If a device has the S_ISHARE bit set and the S_IFDIR bit clear, only one path may be open to device at one time. This is how a non-sharable device is configured. A printer is an example of a non-sharable device. The following values are defined in the header `modes.h`:

Table 1-2. Device Mode Values

Defined Name	Value	Device Capability
S_IREAD	0x0001	Read
S_IWRITE	0x0002	Write
S_IEXEC	0x0004	Execute
S_ISEARCH	0x0004	Search (directories)
S_IAPPEND	0x1000	Append
S_ISIZE	0x2000	Initial File Size Setting
S_ISHARE	0x4000	Non-sharable
S_IFDIR	0x8000	Directory

`dd_fmgr`
: `dd_fmgr` is the offset to the name string of the file manager module to use.

`dd_drvr`
: `dd_drvr` is the offset to the name string of the device driver module to use.

`dd_class`
: `dd_class` is used to identify the class of the device, as random or sequential access. The following values are defined in the header file `io.h`:

Table 1-3. Class of Device Values

Defined Name	Value	Description
`DC_SEQ`	`0x0001`	Sequential access device
`DC_RND`	`0x0002`	Random access device

Software checking this field should test these bits only; the rest may be defined in the future.

`dd_dscres`
: This field is reserved for future use.

The above offsets are offsets from the beginning address of the device descriptor module.

Path Descriptors

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

Path descriptors are variable in size. The full RBF, SBF, SCF, and PCF path descriptor structures are provided in `rbf.h`, `sbf.h`, `scf.h`, and `pcf.h` respectively. Generally, they consist of three main sections:

- a structure common to all path descriptors: `pd_com`
- a section of elements used by file managers and device drivers
- the path descriptor option section

IOMAN requires the first part to be common to all path descriptors. It uses this common section to manage accesses to the path and to dispatch to the associated file manager. File managers can add the information they need after the common part. The options section is used to contain the dynamically alterable operating parameters for the file or device. The appropriate file manager copies the path descriptor options from the device descriptor module when a path is opened or created.

You can use the `_os_gs_popt()` and `_os_ss_popt()` I/O system calls to update the option section of each path descriptor. You can not update any other fields of the path descriptor. The format of the common part is defined in the header file `io.h` and shown here.

Data defined by specific file managers is provided in the *OS-9 Device Descriptor and Configuration Module Reference*.

In user-state, the default setting for the maximum number of paths each process can have open at any time is 32. You can change this setting by using the `_os_ioconfig()` system call. In system-state, the maximum number of open paths depends on available system resources.

pd_com

Declaration

```
typedef struct pathcom {
   path_id       pd_id;       /* path number */
   Dev_list      d_dev;       /* device list element pointer */
   owner_id      pd_own;      /* path creator */
   struct pathcom             *pd_paths,
                              /* list of open paths on device*/
                              *pd_dpd;
                              /* ptr to default directory path desc*/
   u_int16     pd_mode,     /* mode (READ_, WRITE_, or EXEC_) */

               pd_count,
                 /* actual number of open images */
               pd_type,       /* device type */
               pd_class;      /* device class */
   process_id        pd_cproc;
                 /* current active process ID */
   u_char        *pd_plbuf,
                 /* pointer to partial pathlist */
                 *pd_plist;
                 /* pointer to complete pathlist */
   u_int32       pd_plbsz;
                 /* size of pathlist buffer */
   lk_desc       pd_lock;
                 /* reserved for internal use */
   void          *pd_async;
                 /* asynchronous I/O resource pointer */
   u_int32       pd_state;     /* process status bits */
   error_code   (*pd_callback)(pd_com *, iocb_save_area *, u_int32, ...);
                       /* ioman callback function pointer*/
   void          *pd_callbackdata;
                 /* ioman callback data (reserved)*/
   u_int32    pd_rsrv[5]; /* reserved */
} pd_com, *Pd_com;
```

Fields

`pd_id`
: The system path number of the path descriptor.

`pd_dev`
: Pointer to the device list table entry of the device on which this path is opened.

`pd_own`
: Group/user number of the process that created the path descriptor.

`pd_paths`
: Pointer to the next path descriptor in the list of paths opened on the same device.

`pd_dpd`
: Pointer to the default directory path descriptor. When IOMAN creates a path descriptor, and a device name was not specified in the pathlist, it stores a pointer to the path descriptor for the default data or execution (as specified by the mode) directory in this field.

`pd_mode`
: Requested access mode specified when the path descriptor was created.

`pd_count`
: Number of users using the path. When the path descriptor is created this field is set to 1. `pd_count` is incremented when the path is duplicated using the `_os_dup()` system call. The `_os_close()` request decrements this field.

`pd_type`
: Indicate the device type. The values are shown in Table 1-1 and are defined in the header file `io.h`.

`pd_class`
: Indicate the device class. It is used when loading modules. The following values are defined in Table 1-3 and are defined in the header file `io.h`.

`pd_cproc`
: Process ID of the process currently using the path.

`pd_plbuf`
: Pointer to the partial pathlist buffer. This points to the portion of the pathlist relevant to the file manager.

`pd_plist`
: Pointer to the complete pathlist.

`pd_plbsz`
: Size of the pathlist buffer.

`pd_lock`
: Reserved for internal use.

`pd_async`
: Pointer to resources used for performing asynchronous I/O operations.

`pd_state`
: Process status bits used by file managers and drivers to determine the state of a process.

Table 1-4. Process State Values

Defined Name	Value	Description
`PD_SYSTATE`	`0x00000001`	I/O request made from system state.
`PD_CALLBACK`	`0x00000002`	IOMAN handles callbacks.

`pd_callback`
: Pointer to the callback entry-point in IOMAN. This multi-purpose entry-point can aid in the implementation of file managers that are re-entrant on a given path.

Normally, IOMAN allows only a single process to enter a file manager for each path. This entry point makes it possible for a file manager to call back to IOMAN; from there, IOMAN can perform tasks that allow multiple processes into the file manager that is on the same path.

`pd_callbackdata`
: Reserved for use by IOMAN during a callback.

`pd_rsrv`
: Reserved.

Access Modes and Permissions

The following sections define each of the possible access modes and permissions for applicable OS-9 I/O calls.

Access Modes

Certain characteristics must be defined for paths. `<modes.h>` contains `#define` values, which correspond to bit settings of access permissions. With a path, you typically specify whether you are going to `READ` it, `WRITE` to it, or both. Other special purpose mode settings are also available:

`FAM_READ`	Read mode. Path is open for reading.
`FAM_WRITE`	Write mode. Path is open for writing.
`FAM_EXEC`	Execute mode. Search current execution directory instead of current data directory.
`FAM_APPEND`	Append mode. All writes go at end-of-file.
`FAM_SIZE`	Initial file size specified mode.
`FAM_NONSHARE`	Non-sharable mode.
`FAM_DIR`	Directory mode.
`FAM_NOCREATE`	Do not recreate existing file mode.
`FAM_BLKMODE`	Perform I/O in block mode.

Permissions

Files and named pipes also contain access permissions, which specify how the rest of the world can access the resource:

`FAP_READ`	File created with owner read permission.
`FAP_WRITE`	File created with owner write permission.
`FAP_EXEC`	File created with owner execute permission.
`FAP_GREAD`	File created with group read permission.
`FAP_GWRITE`	File created with group write permission.
`FAP_GEXEC`	File created with group execute permission.
`FAP_PREAD`	File created with world read permission.
`FAP_PWRITE`	File created with world write permission.
`FAP_PEXEC`	File created with world execute permission.

File Managers

File managers perform the following functions:

- Process the raw data stream to or from device drivers for a class of similar devices.
- Service all of the I/O system service requests for a class of devices; those not handled by the file manager are passed to the device driver by the file manager.
- Perform mass storage allocation and directory processing--if applicable to the class of devices they service.
- Buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory.
- Monitor and process the data stream.

File managers are re-entrant. One file manager may be used for an entire class of devices having similar operational characteristics. OS-9 systems can have any number of file manager modules.

The following file managers are included in typical systems:

Table 1-5. File Managers

File Manager	Description
RBF (Random Block File Manager)	Operates random-access, block-structured devices such as disk systems.
SCF (Sequential Character File Manager)	Used with single-character-oriented devices such as CRT or hardcopy terminals, printers, and modems.
PIPEMAN (Pipe File Manager)	Supports interprocess communication through memory buffers called pipes.
SBF (Sequential Block File Manager)	Used with sequential block-structured devices such as tape systems.

Table 1-5. File Managers (Continued)

File Manager	Description
PCF (PC File Manager)	Transfers files between OS-9 and DOS systems.
SPF (Stacked Protocol File Manager)	Manages communications. Refer to the SoftStax manual set for more information about SPF.

File Manager Organization

A file manager is a collection of major subroutines accessed through a dispatch table in the static storage of the file manager. IOMAN locates this table by adding an offset specified by the `m_share` field of the file manager module header. The table contains the starting address of each file manager subroutine. The first entry of the table contains the number of subroutines pointed to by the table.

Dispatch Table Sample Listing

Declaration

```
#include <types.h>

#define FUNC_COUNT 16


struct {

     u_int32        func_count;             /* number of functions */

     error_code     (*funcs[FUNC_COUNT])(); /* function table */

} dispatch_table = { FUNC_COUNT,

 {Attach, Chgdir, Close, Create, Delete, Detach, Dup, Getstat, Makdir,
Open, Read, Readln, Seek, Setstat, Write, Writeln }

};
```

Description

When IOMAN calls a file manager subroutine, it always passes two parameters. For the `Attach` and `Detach` functions, the first parameter is a pointer to the parameter block of the caller and the second is a pointer to the device list entry. For all other functions, the first parameter is the pointer to the caller's parameter block and the second is a pointer to the path descriptor for the specified path.

Functions

The following list describes functions that may be used to create a dispatch table.

`Attach`

When an `_os_attach()` call is made to a device, a file manager determines whether the device has been previously attached. If it has, the file manager increments the use count for the device and returns. If the device has not been previously attached, the file manager may perform some additional logical unit initialization and calls the init routine of the device driver to initialize the hardware.

If the device driver's init routine returns an error, the file manager returns the error.

`Chgdir`

On multi-file devices, `_os_chdir()` searches for a directory file. IOMAN allocates a path descriptor. This allows `_os_chdir()` to save information about the directory file for later searches. IOMAN saves the path identifier in the I/O process descriptor.

`_os_open()()` and `_os_create()` begin searching in this directory when the caller's pathlist does not begin with a slash (`/`) character. File managers that do not support directories return an appropriate error code.

`Close`

`_os_close()` ensures any output to a device is completed (writing out the last buffer if necessary), and releases any buffer space allocated when the path was opened.

`_os_close()` may perform specific end-of-file processing if necessary, such as writing end-of-file records on tapes.

`Create`

`_os_create()` performs the same function as `_os_open()`. If the file manager controls multi-file devices (RBF and PIPEMAN), a new file is created.

`Delete`

Multi-file device managers usually do a directory search similar to `_os_open()`. Once the specified file is found, these managers remove the file name from the directory. If it is the last link to a file, any media in use by the file is returned to unused status.

`Detach`

When an `_os_detach()` call is made to a device, a file manager decrements the use count for the device. If the count is still non-zero, the file manager returns. If the use count becomes zero, the file manager calls the driver's terminate routine. If the terminate routine returns an error, the file manager returns the error.

`Dup`

IOMAN implements all of the functions of the `_os_dup()` system call on a device. Normally, file managers are called but do nothing.

`Getstat`

The `_os_getstat()` (get status) system calls are wildcard calls that retrieve the status of various features of a device (or file manager) that are not generally device independent.

The file manager can perform a specific function such as obtaining the size of a file. Status calls that are unknown by the file manager are passed to the driver to provide a further means of device independence.

`Makdir`

`_os_makdir()` creates a directory file on multi-file devices. `I_MAKDIR` is neither preceded by a `Create` nor followed by a `Close`. File managers that cannot support directories or do not support multi-file devices should return the `EOS_UNKSVC` (unknown service request) error.

`Open`

`_os_open()` opens a file on a particular device. This typically involves allocating any required buffers, initializing path descriptor variables, and parsing the path name. If the file manager controls multi-file devices (RBF and PIPEMAN), directory searching is performed to find the specified file.

`Read`

`_os_read()` returns the requested number of bytes to the user's data buffer. If no data is available, an EOF error is returned. `_os_read()` must be capable of copying pure binary data, and generally does not perform editing on the data.

`Readln`

`_os_readln()` differs from `_os_readln()` in two respects. First, `_os_readln` is expected to terminate when the first end-of-line character (carriage return) is encountered. Second, `_os_readln` performs any input editing appropriate for the device.

Specifically, the SCF file manager performs editing that involves functions such as handling backspace, line deletion, and echo.

`Seek`

File managers supporting random access devices use `_os_seek()` to position file pointers of the already open path to the byte specified. Typically, this is a logical movement and does not affect the physical device. No error is produced at the time of the seek if the position is beyond the current end-of-file.

File managers that do not support random access usually do nothing, but do not return an `EOS_UNKSVC` error.

`Setstat`

The `_os_setstat()` (set status) system call is the same as the `_os_getstat()` function except it is generally used to set the status of various features of a device or file manager.

The `_os_setstat()` and `_os_getstat()` system calls are wildcard calls designed to access features of a device (or file manager) that are not generally device independent. Status calls that are unknown to the file manager are passed to the device driver.

`Write`

`_os_write()`, like `_os_read()`, must be capable of recording pure binary data without alteration. Usually, the routines for read and write are nearly identical. The most notable difference is `_os_write()` uses the device driver's output routine instead of the input routine. Writing past the end of file on a device expands the file with new data.

RBF and similar random access devices using fixed-length records (sectors) must often preread a sector before writing it unless the entire sector is being written.

`Writeln`

`_os_writeln()` is the counterpart of `_os_readln()`. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing is also performed. After a carriage return, for example, SCF usually outputs a line feed character and nulls (if appropriate).

Device Driver Modules

Device driver modules perform basic low-level physical I/O functions. For example a basic function of the disk driver is to read or write a physical sector. The driver is not concerned about files and directories, which are handled at a higher level by the OS-9 file manager. Because device drivers are re-entrant, one copy of the module can simultaneously support multiple devices using identical I/O controller hardware.

This section describes the general characteristics of OS-9 device drivers. If you are developing or modifying a device driver, read the OS-9 *Porting Guide*.

Basic Functional Driver Requirements

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

The name by which the device is known to the system is the name of the device descriptor module. OS-9 copies some of the information contained in the device descriptor module to the logical unit and path descriptor data structure for easy access by the drivers.

A device driver is actually a package of subroutines called by a file manager in system state. Device driver functions include:

- initializing device controller hardware and related driver variables as required
- reading standard physical units (a character or sector depending on the device type)
- writing standard physical units (a character or sector depending on the device type)
- returning specified device status
- setting specified device status
- de-initializing devices, assuming the device will not be used again unless re-initialized
- processing device interrupts generated during driver execution

All drivers must conform to the standard OS-9 memory module format. The module type code is `MT_DEVDRVR`. Drivers should have the system state bit set in the attribute byte of the module header. Currently, OS-9 does not make use of this, but future revisions will require all device drivers to be system-state modules.

Interrupts and DMA

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

- For character-oriented devices, set the controller to generate an interrupt on receipt of an incoming character and at the completion of transmission of an outgoing character. Both the input data and the output data should be buffered in the driver.
- For block-type devices (RBF and SBF), set the controller to generate an interrupt upon the completion of a block read or write operation. The driver does not need to buffer data because the driver is passed the address of a complete buffer. A Direct Memory Access (DMA) device, if available, significantly improves the data transfer speed.

Usually, the initialization subroutine of the device driver adds the relevant device interrupt service routine to the OS-9 interrupt polling system using the `_os_irq` system call. The controller interrupts are enabled and disabled by the data transfer routines (for example, `_os_read()` and `_os_write()`) as required. The termination subroutine disables the interrupt hardware and removes the device from the interrupt polling system.

The assignment of device interrupt priority levels can have a significant impact on system operation.

Generally, the smarter the device, the lower you can set its interrupt level. For example, a disk controller that buffers sectors can wait longer for service than a single-character buffered serial port. Assign the clock tick device the highest possible level to keep system time-keeping interference at a minimum. The following is an example:

```
High:    clock ticker

         "dumb" (non-buffering) disk controller

         terminal port

         printer port

Low:     "smart" (sector-buffering) disk controller
```

2 Random Block File Manager (RBF)

This chapter describes the OS-9 disk system file structure, record locking, and file security. It includes the following topics:

Overview

The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service request to random-access devices. Specifically, RBF is a file manager module that supports random-access, block-oriented mass storage devic es (disk systems, bubble memory systems, and high-performance tape systems). RBF can handle any number or type of such systems simultaneously. It is responsible for maintaining the logical physical file structures.

RBF supports a wide range of devices having different performance and storage capacities. Consequently, it is highly parameter driven. The physical parameters it uses are stored on the media itself. On disk systems, this information is written on the first sector of track number zero. The device drivers also use the physical parameters stored on sector 0. These parameters are written by the format program that initializes and tests the media.

Disk File Organization

RBF supports a tree-structured file system. The physical disk organization is designed to do the following:

- Use disk space efficiently.
- Resist accidental damage.
- Access files quickly.

Basic Disk Organization

OS-9 supports “power of two” block sizes ranging from 256 bytes to 32768 bytes. If a disk system is used that cannot directly support the specified block size, the driver module must divide or combine blocks to simulate the allowed size.

Disks are often physically addressed by track number, surface number, and block number. To eliminate hardware dependencies, OS-9 uses a logical block number (LBN) to identify each block without regard to track and surface numbering.

It is the responsibility of the disk driver module or the disk controller to map logical block numbers to track/surface/block addresses. The OS-9 file system uses LBNs from 0 to (n - 1), where n is the total number of blocks on the drive.

All block addresses discussed in this section refer to LBNs.

The `format` utility initializes the file system on blank or recycled media by creating the track/surface/block structure. `format` also tests the media for bad blocks and automatically excludes them from the file system.

Every OS-9 disk has the same basic structure. An identification block is located in logical block zero (LBN 0). It describes the physical and logical format of the storage volume (disk media). Each volume also includes a disk allocation map—indicating the free and allocated disk blocks, and a root directory. The identification block contains block offsets to the file descriptors of the disk allocation map and root directory.

Identification Block

LBN zero always contains the following identification block. In addition to a description of the physical and logical format of the disk, the identification block contains the volume name, date and time of creation, and additional information. If the disk is a bootable system disk, it also includes the starting LBN and size of the sysboot file.

```
typedef struct idblock {
      u_int32    rid_sync,        /* ID block sync pattern */
                 rid_diskid,    /* disk ID number (pseudo random) */
                 rid_totblocks;   /* total blocks on media */
      u_int16    rid_cylinders,   /* number of cylinders */
                 rid_cyl0size     /* cylinder 0 size in blocks */
                 rid_cylsize,     /* cylinder size in blocks */
                 rid_heads,       /* number of surfaces on disk */
                 rid_blocksize,   /* the size of a block in bytes */
                 rid_format,      /* disk format
                           Bit 0: 0 = single side
                                  1 = double side
                           Bit 1: 0 = single density
                                  1 = double density
                           Bit 2: 0 = single track (48 TPI)
                                  1 = double track (96 TPI) */
                 rid_flags,       /* various flags */
                 rid_unused1;     /* 32 bit padding */
      u_int32    rid_bitmap,      /* block offset to bitmap FD */
                 rid_firstboot,   /* block offset to debugger FD */
                 rid_bootfile,    /* block offset to bootfile FD */
                rid_rootdir; /* block offset to root directory FD */
      u_int16    rid_group,       /* group owner of media */
                 rid_owner;       /* owner of media */
      time_t     rid_ctime,       /* creation time of media */
                 rid_mtime;      /* time of last write to ID block */
      char       rid_name[32],    /* volume name */
                 rid_endflag,     /* big-/little-endian flag */
                 rid_unused2[3];  /* long word padding */
      u_int32    rid_parity;      /* ID block parity */
} idblock, *Idblock;
```

Allocation Map

The allocation map indicates which blocks have been allocated to files and which are free. Each bit in the allocation map represents a block on the disk. This means the allocation map varies in size according to the number of bits required to represent the system. If a bit is set, the block is either in use, defective, or nonexistent. `rid_bitmap` specifies the location of the allocation map file descriptor.

Root Directory

The root directory is the parent directory of other files and directories on the disk. It is accessed using the physical device name (such as `/d1`). The root directory file descriptor is specified in `rid_rootdir`.

Basic File Structure

OS-9 uses a multiple-contiguous-segment type of file structure. Segments are physically contiguous blocks used to store the file's data. If all the data cannot be stored in a single segment, additional segments are allocated to the file. This can occur if a file is expanded after creation or a number of contiguous free blocks is not available.

All files have a file descriptor block (FD). An FD contains a list of data segments with starting LBNs and sizes. This is also where information such as file attributes and time of last modification is stored.

The OS-9 segmentation method keeps file data blocks in as close physical proximity as possible to minimize disk head movement. Frequently, files (especially small files) have only one segment. This results in the fastest possible access time. Therefore, it is good practice to initialize the size of a file to the maximum expected size during or immediately after its creation. This enables OS-9 to optimize its storage allocation.

The file descriptor structure is made up of one or more physical blocks on the disk. Only extremely large or fragmented files use more than one file descriptor block. The last element in a file descriptor is a pair of links, one to the previous file descriptor block and one to the next file descriptor block. The end of the file descriptor list is indicated by a next pointer pointing to the first or root file descriptor block. The information section of the file descriptor block is only valid in the root file descriptor block. Only the system uses the file descriptor structure; you cannot directly access the file descriptor.

fd_stats

The following structure, defined in the header file `rbf.h`, describes the contents of a file descriptor block.

Declaration

```
typedef struct fd_stats {
      u_int32     fd_sync,      /* file descriptor sync field */
                  fd_parity,    /* validation parity */
                  fd_flag;      /* flag word */
      u_int16     fd_host,      /* file host owner */
                  fd_group,     /* file group number */
                  fd_owner,     /* file owner number */
                  fd_links;     /* number of links to FD */
      u_int32     fd_size;      /* size of file in bytes */
      time_t      fd_ctime,     /* creation timestamp */
                  fd_atime,     /* last access timestamp */
                  fd_mtime,     /* last modified timestamp */
                  fd_utime,     /* last changed timestamp */
                  fd_btime;     /* last backup timestamp */
      u_int16     fd_rev,     /* RBF revision that created the FD */
                  fd_unused;    /* spare */
} fd_stats;
```

Fields

`fd_sync`
: Identifies this block as a file descriptor block. It is set to `0xfdb0b0fd`.

`fd_parity`
: Contains a 32-bit vertical parity value for the file descriptor block. It is always updated to validate the file descriptor block contents, whether in memory or on disk, to ensure the accuracy of the file structure.

`fd_flag`
: Contains the attributes and permissions of the file.

Table 2-1. File Attributes and Permissions

Flag	Description
`FD_SMALLFILE`	File is small enough to fit in the file descriptor
`FD_DIRECTORY`	File is a directory
`FD_EXCLUSIVE`	Only one active open allowed

Table 2-1. File Attributes and Permissions (Continued)

Flag	Description
`PERM_OWNER_READ`	Read permission by owner
`PERM_OWNER_WRITE`	Write permission by owner
`PERM_OWNER_SRCH`	Search permission by owner
`PERM_OWNER_EXEC`	Execute permission by owner
`PERM_GROUP_READ`	Read permission by group
`PERM_GROUP_WRITE`	Write permission by group
`PERM_GROUP_SRCH`	Search permission by group
`PERM_GROUP_EXEC`	Execute permission by group
`PERM_WORLD_READ`	Read permission by world
`PERM_WORLD_WRITE`	Write permission by world
`PERM_WORLD_SRCH`	Search permission by world
`PERM_WORLD_EXEC`	Execute permission by world

All bits not defined above are reserved

`fd_host`
: Contains the host owner number of the user to which the file belongs

`fd_group`
: Contains the group number of the user to which the file belongs. This is initially set to the group number of the process creating the file. Only the owner of the file or a super user can change the group number

`fd_owner`
: Contains the owner number of the user to which the file belongs. This is initially set to the owner number of the process creating the file. Only the owner of the file or a super user can change the owner number

`fd_links`
: Contains the number of hard links to this file. A hard link is a directory entry pointing to this file

`fd_size`
: Contains the size of the file in bytes

`fd_ctime`
: Contains a time stamp representing the time when the file descriptor was initially created. This time stamp is never changed

`fd_atime`
: Contains a time stamp representing the time when the file was last accessed. This time stamp is updated whenever the file is opened, read, or written. If the file is a directory file, this field is not updated when it is searched by RBF

`fd_mtime`
: Contains a time stamp representing the time when the file was last modified. The time stamp is updated whenever a file is opened for write or a write is performed on the file

`fd_utime`
: Contains a time stamp representing the time when the file was last changed. The time stamp is updated whenever a write is performed on the file or the file descriptor data changes

`fd_btime`
: Contains a time stamp representing the last time a back up of the file was made. The backup program (`fsave`) updates the time stamp whenever a back up of the file is made

`fd_rev`
: Contains the edition number of the RBF file manager that created the file descriptor

`fd_unused`
: Reserved.

The remainder of the file descriptor block up to the last eight bytes is filled with segment descriptors, unless the file is a small file. A small file is one whose contents fits the area of the file descriptor that is usually reserved for segments.

The number of segment descriptors in the file descriptor block depends on the logical block size. The structure of a segment descriptor is shown here and defined in the header file `rbf.h`. The `seg_offset` field contains the LBN of the first block in this segment and the `seg_count` field contains the number of logical blocks in the segment.

```
typedef struct fd_segment {

     u_int32     seg_offset,     /* segment block offset */

                 seg_count;      /* segment block count */

} fd_segment;
```

The last part of the file descriptor block contains links to other file descriptors for a file. If there is only one file descriptor block for the file, these fields point to the one file descriptor block. The links structure is shown here and defined in the header file `rbf.h`.

```
typedef struct fd_links {

     u_int32     link_prev,     /* previous fd block */

                 link_next;     /* next fd block */

} fd_links;
```

Small Files

OS-9 RBF implements a class of files called small files. A file is considered small when its contents fit in the area of the file descriptor reserved for segments. A small file has the `FD_SMALLFILE` bit set in the `fd_flag` field. From a user's perspective, small files behave exactly like other files. RBF automatically changes a small file to a non-small file if the file grows too big to fit in the file descriptor block.

Logical Block Numbers

RBF maintains the file pointer and logical end-of-file used by application software and converts them to the logical disk block number using the data in the segment list. You do not have to be concerned with physical blocks. OS-9 provides fast random access to data stored anywhere in the file. All the information required to map the logical file pointer to a physical block number is packaged in the file descriptor block. This makes the OS-9 record-locking functions very efficient.

Segment Allocation

Each device descriptor module has a value called a segment allocation size, that specifies the minimum number of blocks to allocate to a new segment. Set this value so file expansions do not produce a large number of tiny segments. If the system uses a small number of large files, set this field to a relatively high value, and vice versa.

When a file is created, it has no data segments allocated. Write operations past the current end-of-file allocate additional blocks to the file. The first write is always past the end-of-file. Generally, subsequent file expansions are also made in minimum allocation increments.

An attempt is made to expand the last segment before adding a new segment.

If all of the allocated blocks are not used when the file is closed, the segment is truncated and any unused blocks are deallocated in the bitmap. For random-access databases that expand frequently by only a few records, the segment list rapidly fills with small segments. A provision has been added to prevent this from being a problem.

If a file (opened in write or update mode) is closed when it is not at end-of-file, the last segment of the file is not truncated. All programs dealing with a file in write or update mode must not close the file while at end-of-file, or the file loses its excess space. The easiest way to ensure this is to perform a `_os_eek(0)` before closing the file. This method was chosen because random access files are frequently somewhere other than end-of-file, and sequential files are almost always at end-of-file when closed.

Directory File Format

Directory files have the same structure as other files, except the logical contents of a directory file conform to the following conventions:

- A directory file consists of an integral number of 64-byte entries.
- The end of the directory is indicated by the normal end-of-file.
- Each entry consists of a field for the file name and a field for the address of the first file descriptor block of the file.

The structure of a directory entry is shown here and defined in the header file `rbf.h`. The file name field (`dir_uname`) contains the null terminated file name. The first byte is set to zero (a null string) to indicate a deleted or unused entry. The address field (`dir_fd_addr`) contains the LBN of the first file descriptor block.

```
#define MAXNAME    43                              /* size of name */

#define DIRENTSIZE 64                              /* size of directory
entry */

typedef struct dirent {

     char     dir_name[MAXNAME+1],                 /* name of file */

              dir_unused[DIRENTSIZE-MAXNAME-sizeof(u_int32)-1];

    u_int32     dir_fd_addr;                     /* where file's FD is */

} dirent;
```

When a directory file is created, two entries are automatically created: the dot (.) and dot dot (..) directory entries. These specify the directory and its parent directory, respectively.

Raw Physical I/O on RBF Devices

You can open an entire disk as one logical file. This enables you to access any byte(s) or block(s) by physical address without regard to the normal file system. This feature is provided for diagnostic and utility programs that must be able to read and write to ordinarily non-accessible disk blocks.

A device is opened for physical I/O by appending the "at" character (@) to the device name. For example, you can open the device `/d2` for raw physical I/O under the pathlist: `/d2@`.

Standard open, close, read, write, and seek system calls are used for physical I/O. A seek system call positions the file pointer to the actual disk physical address of any byte. To read a specific block, perform a seek to the address computed by multiplying the LBN by the logical block size. For example, to read physical disk block 3 on media with a logical block size of 256, a seek is performed to address 768 (256*3), followed by a read system call requesting 256 bytes.

If the number of blocks per track of the disk is known or read from the identification block, any track/block address can be readily converted to a byte address for physical I/O.

Use the special @ file in update mode with extreme care. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The @ file is considered different from any other file and only conforms to record lockouts with other users of the @ file.

Improper physical I/O operations can corrupt the file system. Take great care when writing to a raw device. Physical I/O calls also bypass the file security system. For this reason, only super users can open the raw device for write permit. Non-super users are only permitted to read the identification block (LBN 0). Attempts to read past this return an end-of-file error.

Raw Physical I/O on RBF Devices

Block Mode

RBF supports block mode on paths opened to the raw device. In block mode, blocks of the device beyond the first four gigabytes can be addressed. They are addressed by their block number, rather than their byte number.

If a path to the raw device is opened in block mode (`FAM_BLKMODE`), semantics of some system calls on that path change. Generally, where one would normally express a value in bytes, the value is specified in terms of blocks. For example, an `_os_seek`(path, 1) would seek to "block 1" instead of "byte 1". In addition, where RBF would normally return a byte number or a number of bytes, it returns a block number or number of blocks.

I/O calls (`_os_read()`, `_os_readln()`, `_os_write()`, and `_os_writeln()`) still take their counts in terms of bytes, but the number of bytes must be a multiple of the block size of the device. For example, if the block size of the device is 256 bytes, valid sizes for I/O would be 256, 512, etc.

Record Locking

Record locking is a general term referring to preserving the integrity of files that more than one user or process can access. This involves recognizing when a process is trying to read a record another process may be modifying and deferring the read request until the record is safe. This process is referred to as conflict detection and prevention. RBF record locking also handles non-sharable files and deadlock detection.

OS-9 record locking is transparent to application programs. Most programs may be written without special concern for multi-user activity.

Record Locking and Unlocking

Conflict detection must determine when a record is being updated. RBF provides true record locking on a byte basis. A typical record update sequence is as follows:

```
_os_read(path, count, buffer)   program reads record;
                                RECORD IS LOCKED
 .
 .                              program updates record
 .
_os_seek(position)              reposition to record
_os_write(path, count, buffer)  record is rewritten;
                                RECORD IS RELEASED
```

When a file is opened in update mode, any read operation locks out the record because RBF is not aware if the record may be updated. The record remains locked until the next read, write, or close operation occurs. Reading files opened in read or execute modes does not lock the record because records cannot be updated in these modes.

A subtle problem exists for programs using a database and occasionally updating its data. When you look up a particular record, the record may be locked out indefinitely if the program neglects to release it. This problem is characteristic of record locking systems and can be avoided by careful programming.

Only one portion of a file may be locked out at one time. If an application requires more than one record to be locked out, multiple paths to the same file may be opened with each path having its own record locked out. RBF notices the same process owns both paths and keeps them from locking each other out. Alternately, the entire file may be locked out, the records updated, and the file released.

Non-Sharable Files

You can lock files when an entire file is considered unsafe for use by more than one user. On rare occasions, it is necessary to create a nonsharable file. A non-sharable file can never be accessed by more than one process at a time.

To create a non-sharable file, set the exclusive access (`x`) bit in the file attribute byte. The bit can be set when the file is created, or later using the `attr` utility.

If the exclusive access bit has been set, only one process may open the file at a time. If another process attempts to open the file, an error (`EOS_SHARE`) is returned.

Most often, a file needs to be non-sharable only while a specific program is executing. To do this, open the file with the exclusive-access bit set in the access mode parameter.

One example might be when a file is being sorted. If the file is opened as a non-sharable file, it is treated as though it had an exclusive access attribute. If the file has already been opened by another process, an error (`EOS_SHARE`) is returned.

A necessary quirk of non-sharable files is they may be duplicated using the `_os_dup()` system call, or inherited. Therefore, a non-sharable file may actually become accessible to more than one process at a time. Non-sharable only means the file may be opened once. It is usually a bad idea to have two processes actively using any disk file through the same (inherited) path.

End of File Lock

An EOF lock occurs when you read or write data at the end-of-file. The end-of-file is kept locked until a read or write is performed that is not at end-of-file. EOF lock is the only case when a write call automatically locks out any of the file. This avoids problems that may otherwise occur when two users want to extend a file simultaneously.

An interesting and useful side effect occurs when a program creates a file for sequential output. As soon as the file is created, EOF lock is gained, and no other processes can "pass" the writer in processing the file.

For example, if an assembly listing is redirected to a disk file, a spooler utility may open and begin listing the file before the assembler writes the first line of output. Record locking always keeps the spooler one step behind the assembler, making the listing come out as desired.

Deadlock Detection

A deadlock can occur when two processes simultaneously attempt to gain control of the same two disk areas. If each process gets one area (locking out the other process), both processes can become stuck permanently, waiting for a segment that can never become free. This situation is a general problem not restricted to any particular record locking method or operating system.

If this occurs, a deadlock error (`EOS_DEADLK`) is returned to the process that detects the deadlock. The easiest way to avoid deadlock errors is to access records of shared files in the same sequences in all processes that may be run simultaneously. For example, always read the index file before the data file, never the data file before the index file.

When a deadlock error occurs, a program cannot simply retry the operation in error. If all processes used this strategy, none would ever succeed. At least one process must release control over a requested segment for any to proceed.

Record Locking Details for I/O Functions

Record locking details are described, by function, in the following subsections.

_os_open()/_os_create()

When opening files, the most important guideline to follow is not to open a file for update if it is only necessary to read. Files open for read only do not lock out records and generally help the system run faster. If shared files are routinely opened for update on a multi-user system, you may become hopelessly record locked for extended periods of time.

Use the special @ file in update mode with extreme care. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The @ file is considered different from any other file and only conforms to record lockouts with other users of the @ file.

_os_read()/_os_readln()

`_os_read()` and `_os_realn()` lock out records only if the file is open in update mode. The locked out area includes all bytes starting with the current file pointer and extending for the requested number of bytes.

For example, if a `_os_realn()` call is made for 256 bytes, exactly 256 bytes are locked out, regardless of how many bytes are actually read before a carriage return is encountered. EOF lock occurs if the bytes requested also include the current end-of-file.

A record remains locked until any of the following occur:

- another read is performed
- a write is performed
- the file is closed
- an `_os_ss_lock()` set status call is issued

Releasing a record does not normally release EOF lock. A read or write of zero bytes releases any record lock, EOF lock, or file lock.

_os_write/_os_writeln()

Write calls always release any record that has been locked out. In addition, a write of zero bytes releases EOF lock and file lock. Writing usually does not lock out any portion of the file unless it occurs at end-of-file, when it gains EOF lock.

_os_seek()

Seek does not effect record locking.

_os_setstat()

Two setstats have been included for the convenience of record locking:

`_os_ss_lock()`	Lock or release part of a file.
`_os_ss_ticks()`	Set the length of time a program waits for a locked record.

Refer to the *Ultra C Library Reference* manual for more information on these calls.

File Security

Each file has a group/user ID identifying the owner of the file. These are copied from the current process descriptor when the file is created. Usually a file's owner ID is not changed.

An attribute word is also specified when a file is created. The file's attribute word tells RBF in which modes the file may be accessed. Together with the file's owner ID, the attribute word provides (some) file security.

The attribute word has three sets of bits indicating whether a file may be opened for read, write, or execute by the owner, group, or public.

- An owner is a user with the same owner ID.
- The group includes all users with the same group ID.
- The public includes all users.

When a file is opened, access permissions are checked on all directories specified in the pathlist, as well as the file itself. If you do not have permission to search a directory, you cannot read any files in that directory.

A super user (a user with group ID of zero) may access any file in the system. Files owned by the super user cannot be accessed by users of any other group unless specific access permissions are set. Files containing modules owned by the super user must also be owned by the super user. If not, the modules contained within the file can not be loaded.

The RBF file descriptor stores the group/user ID in two 16-bit fields (`fd_group` and `fd_owner`).

The system manager must exercise caution when assigning group/user IDs.

Creating RBF Drivers and Descriptors

The following sections discuss how to create RBF drivers and descriptors.

Creating Disk Drivers

Creating a disk driver for your target system is similar to creating a console terminal driver as explained in Chapter 9 of the OS-9 Porting Guide. However, disk drivers are more complicated. You can use a Microware-supplied sample disk driver source file as a prototype.

If the target system has both floppy disks and hard disks, create the floppy disk driver first, unless they both use a single integrated controller. You can create the hard disk driver after the system is up and running on the floppy.

A test disk must exist with the correct type of OS-9 formatting. If you are using:

- an OS-9 based host system, you can make test disks on the host system.
- a cross-development system, you should obtain sample pre-formatted disks from Microware.

You should make a non-interrupt driver the first time to make your debugging task easier. Make a new download file that includes the disk driver and descriptor modules along with one or two disk-related commands (such as `dir` and `free`) for testing. If you are using the RomBug, include the driver's `.stb` module for easier debugging.

You can add the previously tested and debugged console driver and descriptor modules to your main system boot at this time. This minimizes download time as in the previous step.

Disk drivers make use of the RBF file manager. The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service requests to random-access devices. Specifically, RBF is a file manager module supporting random-access, block-oriented mass storage devices (disk systems, bubble memory systems, and high-performance tape systems). RBF can handle any number or type of such systems simultaneously. It is responsible for maintaining the logical and physical file structures.

When you write a device driver, do not include MPU/CPU specific code. This makes the device driver portable.

RBF supports a wide range of devices having different performance and storage capacities. Consequently, it is highly parameter driven. The physical parameters it uses are stored on the media itself. On disk systems, this information is written on the first sector of track number 0. The device drivers also use the physical parameters stored on sector 0. These parameters are written by the `format` program that initializes and tests the media.

Understanding SCSI Device Driver Differences

This section explains some unique aspects of SCSI device drivers. The basic premise of the SCSI system is to break the OS-9 driver into separate areas of functionality (high- and low-level). This enables different file managers and drivers to talk to their respective devices on the SCSI bus.

The device driver handles the high-level functionality. The device driver is the module called directly by the appropriate file manager. Device drivers deal with all target-controller-specific/device-class issues (for example, SCSI hard disks or tapes).

Hardware Configurations

To configure a high-level driver, complete the following steps:

Step 1. Prepare the command packets for the SCSI target device.

Step 2. Pass this packet to the low-level subroutine module.

The low-level subroutine module passes the command packet (and data if necessary) to the target device on the SCSI bus. The low-level code does not concern itself with the contents of the commands/data; it performs requests for the high-level driver. The low-level module also coordinates all communication requests between the various high-level drivers and itself. The low-level module is often an MPU/CPU specific module, so it can be written as an optimized module for the target system.

The device descriptor module contains the name strings for linking the modules together. The file manager and device driver names are specified in the normal way. The low-level module name associated with the device is indicated through the `ds_ldrvrnam` field in the device-specific portion of the device descriptor. This offset pointer points to a string containing the name of the low-level module.

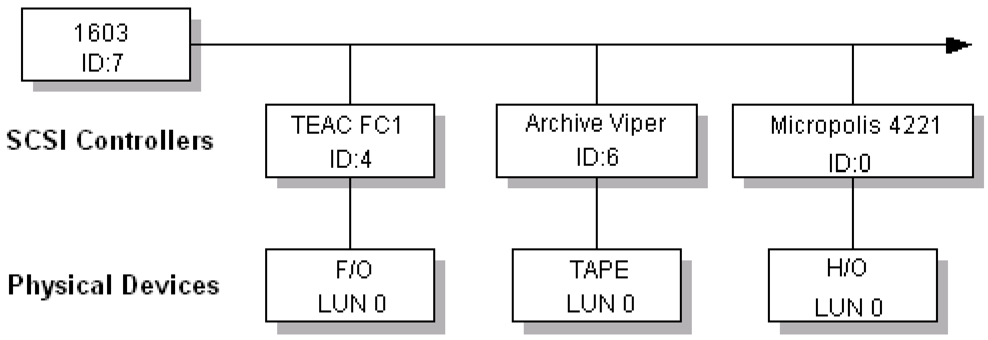
Example SCSI Software Configuration

An example system setup shows how drivers for disk and tape devices can be mixed on the SCSI bus without interference. The setup includes:

- Micropolis 4221 Hard Disk with embedded SCSI controller addressed as SCSI ID 0
- Archive Viper QIC tape drive with embedded SCSI controller addressed as SCSI ID 4
- TEAC SCSI floppy disk drive with embedded SCSI controller addressed as SCSI ID 6
- Host CPU
- MVME1603
- Uses NCR53C810 or NCR53C825 Interface chip
- ID of chip is SCSI ID 7

The hardware setup should look similar to that shown in the figure below:

Figure 2-1. SCSI Setup



The high-level drivers associated with this configuration are shown in Table 2-2.

Table 2-2. High-Level Drivers

Name	Handles
`RBTEAC`	TEAC SCSI floppy devices
`SBSCSI`	Archive VIPER tape device
`RBSCCS`	Hard disk device

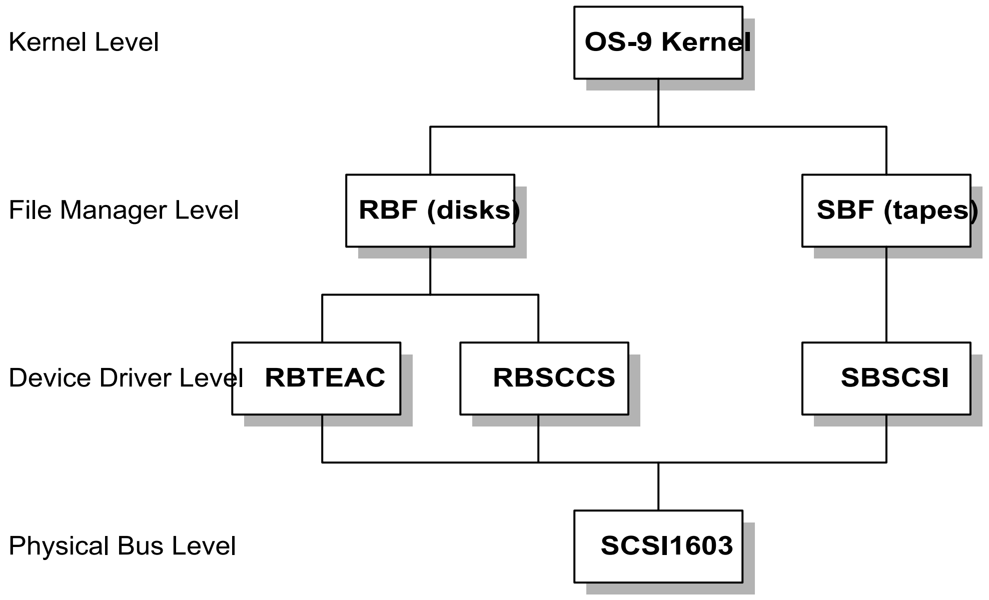
The low-level module associated with this configuration is shown in Table 2-3.

Table 2-3. Low-Level SCSI Subroutine Module

Name	Handles
`SCSI1603`	NCR53C8xx Interface on the MVME1603 CPU

A conceptual map of the OS-9 modules for this system would look like the following figure:

Figure 2-2. OS-9 Modules



A common reconfiguration occurs when you add additional devices of the same type as the existing device. For example, adding an additional disk to the SCSI bus on the MVME1603. To add a similar controller, Micropolis 4220, to the bus, you only need to create a new device descriptor. (The example ports have both `/h0` and `/h1` descriptors that demonstrate the use of additional SCSI disk controller devices.) There are no drivers to write or modify, as these already exist (`RBSCCI` and `SCSI1603`). You need to modify the existing descriptor for the `RBSCCS` device to reflect the second device's physical parameters (such as, SCSI ID) and change the actual name of the descriptor itself.

Testing the Disk Driver

Test the disk driver using the following procedure. (You can omit Steps 1 and 2 if the necessary system modules are in ROM.)

Step 1. After a reset, set the debugger's relocation register to the RAM address where you want the system modules (now including the console driver) loaded.

Step 2. Download the system modules, but do not insert breakpoints.

Step 3. Set the debugger's relocation register to the RAM address where you want the disk driver and descriptor loaded. Ensure this address does not overlap the area where the system modules were previously loaded.

Step 4. Download the disk driver and descriptor modules, but do not insert breakpoints.

Step 5. Type `gb` to initiate the boot process. If a menu appears, select the Boot from ROM option (ro). The following message should appear:

```
An OS-9000 kernel was found at $xxxxxxxx
```

This is followed by a register dump and a RomBug prompt. If you do not see this message, the system modules were probably not downloaded correctly or were loaded into the wrong memory area.

Step 6. Type gb again. This executes the kernel's initialization code including the OS-9 module search. You should get another register dump and debug prompt.

Step 7. Insert breakpoints in the disk driver if needed.

Step 8. Type gb again. This starts the system. You should see the following display:

```
Shell $
```

Step 9. Run the `dir` utility. If this fails, begin debugging by repeating this procedure with breakpoints inserted in the driver's `INIT`, `GETSTAT`, `SETSTAT`, and `READ` routines.

Creating RBF Device Drivers

RBF-type device drivers support any random access storage device that reads and writes data in fixed size blocks (for example, disks or RAM memories). The file manager handles all file system processing and passes the driver a data buffer and a logical block number (LBN) for each read or write operation.

Write calls to the driver initiate the block write operation and, if required, a prior "seek" operation. For interrupt driven systems, the controller generates an interrupt when the data has been written from the buffer on to the disk. The driver must suspend itself until the interrupt occurs.
DMA operation is preferred if available. If the "verify" flag is set in the path descriptor (`pd_vfy`), the block should be read back and verified.

Drivers for hard disks are relatively simple for two reasons:

- The driver typically works with an intelligent controller.
- The disk format is fixed.

For example, most SCSI type hard disk controllers directly accept OS-9's logical sector number as the physical sector address.

Floppy disk drivers are more complicated. They work with less capable disk controllers and often must handle a variety of disk sizes.

Disk drivers keep a table in the logical unit static variable storage area containing current track addresses and disk format information for each drive (unit). The track addresses are used for controllers with explicit seek commands to determine if the head must be moved prior to a read or write operation. The format data part of each table entry selects density, number of sides, etc.

The `INIT` routine obtains some initialization data from the device descriptor module. Each disk media has similar format information recorded on LBN zero (the `format` utility puts it there). Whenever block zero of a floppy disk is read, the drive's device static storage is updated with the information actually read. This is how the driver automatically adapts to different disk formats. Initialization of the static storage must occur prior to access of any other block on the drive.

RBF Device Driver Storage Definitions

RBF-type device driver modules contain a package of subroutines that perform block 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.

IOMAN allocates a driver static storage area for each driver and port combination (that may control several drives). The size of this storage area is specified in the device driver module header (m_data). RBF requires some of this storage area. The device driver may use the remainder in any manner. IOMAN also allocates a logical unit static storage area for each drive on a controller. The size of this storage area is specified by the device descriptor module (m_data). The structure of logical unit static storage is described earlier in this chapter.

The format of the part of driver static storage required by RBF is shown here and defined in the header file rbf.h. This is the dispatch table pointed to by the v_dr_stat field of the device list described in the previous chapter.

```
typedef struct rbf_drv_stat {
     u_int32      funcs,            /* number of functions */
         (*v_init)(),      /* address of driver init routine */
         (*v_read)(),      /* address of driver read routine */
         (*v_write)(),     /* address of driver write routine */
         (*v_getstat)(),   /* address of driver getstat routine */
         (*v_setstat)(),   /* address of driver setstat routine */
         (*v_term)();      /* address of device terminate routine */
     lock_id  v_drvr_rsrc_id;   /* the driver's resource lock ID */
     process_id   v_busy,           /* process using the device */
                  v_wake;           /* for use by the driver */
} rbf_drv_stat, *Rbf_drv_stat;
```

RBF Device Driver Subroutines

As with all device drivers, RBF device drivers use a standard executable memory module format with a module type of MT_DEVDRVR.

Within the driver's global static storage resides the dispatch table to the driver functions. Each function should return SUCCESS if the operation was successful. Otherwise, it should return an appropriate error code.

Table 2-4. RBF Subroutines

Function	Description
GETSTAT	Get device status
INIT	Initialize device and its static storage area
IRQ SERVICE ROUTINE	Service device interrupts
READ	Read sector(s)
SETSTAT	Set device status
TERMINATE	Terminate device
WRITE	Write sector(s)

GETSTAT
Get Device Status

Syntax

```
error_code getstat(
   void              pb,
   Rbfpd             pd,
   Dev_list          dev);
```

Description

These routines are wildcard calls used to get the device's operating parameters as specified for the OS-9 `getstat` service requests.

Usually all `GetStat` codes return with an `EOS_UNKSVC` (UnKnown Service Request) error.

Parameters

`pb`	is the status parameter block.
`pd`	is the path descriptor.
`dev`	is the device list entry.

INIT

Initialize Device and its Static Storage Area

Syntax

```
error_code init (Dev_list dev);
```

Description

The INIT routine must:

1. Check for previous initialization. It must allocate a lock for the driver in the driver static storage and place it in `v_drvr_rsrc_id`.
2. Initialize device control registers (enable interrupts if necessary).
3. If the driver uses interrupts, place the IRQ service routine on the IRQ polling list by using the `_os_irq()` service request.
4. If events are to be used for interrupt signaling, the event should be created and its ID placed in the driver static storage.

Parameters

`dev`	is the device list entry.

IRQ SERVICE ROUTINE

Service Device Interrupts

Syntax

```
error_code irq(Rbf_drvr_stat drvstat);
```

Description

Although this routine is not included in the device driver module branch table and is not called directly by RBF, it is a key routine in interrupt-driven device drivers. Its function is as follows:

1. Poll the device. If the interrupt is not caused by this device, the interrupt service routine should return with an `EOS_NOTME` error code.
2. Service device interrupts.
3. Inform the driver that the interrupt has occurred. This could involve either performing an event set system call or sending a signal, depending on the driver implementation. If the signal method is used, the interrupt service routine must clear the `v_wake` flag in the driver static storage area to notify the driver that the interrupt has indeed occurred.
4. When the IRQ service routine finishes servicing an interrupt, it must return `SUCCESS`. `SUCCESS` is defined in the `const.h` header file.

The IRQ service routine is passed one parameter. This parameter is specified when the driver calls `_os_irq()` to install the service routine on the interrupt polling table. This value is placed in the global pointer register. See the *Using Ultra C/C++* manual for the API (global register) used for your processor. This variable should be a pointer to the driver static storage. However, the driver can use this parameter for anything useful.

Parameters

`drvstat`	is the driver static storage.

READ

Read Sector(s)

Syntax

```
error_code read(
   u_int32                  blks,
   u_int32                  blkaddr,
   Rbfpd                    pd,
   Dev_list                 dev);
```

Description

The READ routine must:

1. Get the buffer address from `pd_buf` in the path descriptor.
2. Verify the drive number from `pd_drv` in the path descriptor.
3. Compute the physical disk address from the logical block number.
4. Seek to the physical track requested.
5. Read block(s) from the disk into the buffer.
6. Wait for the command to finish.

OS-9 drivers typically use the OS-9 event system to wait for interrupts. The driver read/write routine executes an event wait and the interrupt service routine issues an event signal or event set to inform the driver that the interrupt has occurred. Drivers can also use the more traditional sleep and signal method. To do this, the driver copies the current process ID from `v_busy` in the driver static storage to `v_wake` and does an indefinite sleep (a sleep for 0 ticks). The interrupt service routine then sends a wake up signal to the sleeping process using the ID stored in `v_wake`.

Drivers do not have to be interrupt driven. A driver can simply poll the device waiting for command completion, but this disrupts time sharing performance. If the disk controller cannot be interrupt-driven, a programmed I/O transfer must be performed.

Whenever logical sector zero is read, the `idblock` section must be copied to the drive table of logical unit static storage.

If bit number 1 in the `pd_cntl` field is clear, RBF only requests one sector reads. If the bit is set, RBF may request up to `pd_xfersize` bytes of data to be read. RBF divides `pd_xfersize` by the block size to determine the maximum number of blocks that can be transferred. `pd_xfersize` is defined in the path descriptor options section of the device descriptor.

Parameters

`blks`	is the number of blocks to transfer.
`blkaddr`	is the starting block address.
`pd`	is the path descriptor.
`dev`	points to the device list entry.

SETSTAT

Set Device Status

Syntax

```
error_code setstat(
   void                       pb,
   Rbfpd                      pd,
   Dev_list                   dev);
```

Description

These routines are wildcard calls used to get the device's operating parameters as specified for the OS-9 `setstat` service requests.

Typical RBF drivers have routines to handle the `SS_WTRK` and `SS_RESET` `setstat` calls. Usually all `getstat` calls and other `setstat` calls return with an `EOS_UNKSVC` (UnKnown Service Request) error.

Parameters

`pb`	is the status parameter block.
`pd`	is the path descriptor.
`dev`	is the device list entry.

TERMINATE
Terminate Device

Syntax

```
error_code term(Dev_list dev);
```

Description

This routine is called when a device is no longer in use in the system. This is defined as when the link count of its device table entry becomes zero (see `_os_attach()` and `I_DETACH`).

The TERM routine must:

1. Wait until any pending I/O has completed.
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.
4. Delete any events used by the driver.
5. Return the lock allocated by the driver in the init routine.

Parameters

`dev`	is the device list entry.

WRITE
Write Sector(s)

Syntax

```
error_code write(
   u_int32                    blks,
   u_int32                    blkaddr,
   Rbfpd                      pd,
   Dev_list                   dev);
```

Description

The WRITE routine must:

1. Get the buffer address from `pd_buf` in the path descriptor.
2. Verify the drive number from `pd_drv` in the path descriptor.
3. Compute the physical disk address from the logical block number.
4. Seek to the requested physical track.
5. Write buffer(s) to the disk.
6. Wait for the command to complete.
7. If `pd_vfy` in the path descriptor is equal to zero, read the data back and verify that it is written correctly. We recommend that the compare loop be as short as possible to keep the necessary block interleave value to a minimum.

OS-9 drivers typically use the event system to wait for interrupts. The driver read/write routine executes an event wait and the interrupt service routine issues an event signal or pulse to inform the driver that the interrupt has occurred. Drivers can also use the more traditional sleep and signal method by copying the current process ID from `v_busy` in the driver static storage to `v_wake`. Next, it does an indefinite sleep (a sleep for 0 ticks). The interrupt service routine then sends a wake up signal to the sleeping process using the ID stored in `v_wake`.

Drivers do not have to be interrupt driven. A driver can poll the device waiting for command completion, but this hampers time sharing performance. If the disk controller cannot be interrupt-driven, a programmed I/O transfer must be performed.

If bit 1 in `pd_cntl` is clear, RBF only requests one block writes. If the bit is set, RBF may request up to `pd_xfersize` bytes of data to be written. RBF divides `pd_xfersize` by the block size to determine the maximum number of blocks that can be transferred. `pd_xfersize` is defined in the path descriptor options section of the device descriptor.

Parameters

`blks`	is the number of blocks to transfer.
`blkaddr`	is the starting block address.
`pd`	is the path descriptor.
`dev`	points to the device list entry.

Using RBF Device Descriptor Modules

The RBF device descriptor consists of four parts:

- The OS-9 module header
- The common information required by IOMAN for all descriptors
- The path descriptor options
- The logical unit static storage

Two of these parts are contained in this structure (defined in `rbf.h`):

```
typedef struct rbf_desc {
  dd_com             dd_descom;
  rbf_path_opts      dd_pathopt;
} rbf_desc, *Rbf_desc;
```

The table below explains the RBF device descriptor structure.

Table 2-5. RBF Device Descriptor Structure

Name	Description
`dd_descom`	This is the common information structure IOMAN requires to be in all device descriptors.
`dd_pathopt`	This structure contains the RBF path descriptor options and information IOMAN uses to initialize the device. RBF copies this information into the path descriptor when a file is opened or created.

Logical Unit Static Storage Initialization

IOMAN initializes logical unit static storage from the device descriptor using a declaration of the following structure. This structure is defined in `rbf.h`.

```
typedef struct rbf_lu_stat {
   rbf_drv_info       v_driveinfo;       /* the drive's information */
   u_char             v_vector,         /* the interrupt vector */
                      v_irqlevel,       /* the interrupt level */
                      v_priority,        /* the interrupt priority */
                      v_unused;         /* unused byte */
   rbf_lu_opts        v_luopt;          /* logical unit options */
   u_int32            v_reserved[2];    /* reserved */
} rbf_lu_stat;
```

Table 2-6. RBF Logical Unit Static Storage Structure

Name	Description
`v_driveinfo`	**Disk Drive Information** RBF maintains information about the media in use in this field. A full description of this structure follows this discussion.
`v_vector`	Interrupt Vector This is the vector number of the device interrupt.

Table 2-6. RBF Logical Unit Static Storage Structure (Continued)

Name	Description
`v_irqlevel`	**Interrupt Level** This is the hardware priority of the device interrupt.
`v_priority`	**Interrupt Priority** This is the software (polling) priority of the device interrupt.
`v_luopt`	**Device Options** This is the device options section. A full description of this structure follows the discussion on Disk Drive Information.
`v_reserved`	**Reserved** Space reserved for future expansion.

Disk Drive Information

Because RBF supports a wide variety of format options for disk media, it maintains information about the current media being processed in the logical unit static storage for the drive. The structure definition of the drive information is shown here and a description of each field follows. This structure is defined in the header file `rbf.h`.

These values should not be changed from the defaults defined in the descriptor source file.

```
typedef struct rbf_drv_info {
  idblock               v_0;            /* standard ID block */
  lk_desc               v_file_rsrc_lk;  /* the file list resource lock */
  Rbf_path_desc         v_filehd;        /* list of open files on drive */
  lk_desc               v_free_rsrc_lk;  /* free list resource lock */
  struct freeblk        *v_free,       /* pointer to free list structure */
                        *v_freesearch;   /* start search for free space */
    u_int32             v_diskid;      /* disk ID number */
    fd_segmentv_mapseg;      /* the bitmap segment */
    Idblock             v_bkzero;      /* pointer to block zero buffer */
    u_int32             v_resbit,    /* reserved bitmap block # (if any) */
                        v_trak;        /* current track number */
    u_int32             v_softerr,     /* recoverable error count */
                        v_harderr;     /* non-recoverable error count */
    struct cachedriv    *v_cache;      /* drive cache information ptr */
    lk_desc             v_crsrc_lk;    /* cache resource lock */
    u_int16             v_numpaths;   /* # of open paths on this device */
    u_char              v_zerord,      /* block zero read flag */
                        v_init;        /* drive initialized flag */
    Rbf_path_opts       v_dopts;       /* copy of the default opts */
    u_char              v_endflag,     /* big/little endian flag */
                        v_dumm2[3];    /* reserved */
    lk_desc             v_fd_free_rsrc_lk; /* FD free list lock*/
    Fdl_list            v_fd_free_list; /* free FD block structures */
    lk_desc             v_blks_rsrc_lk;    /* free block list lock */
    Blockbuf            v_blks_list;     /* list of free block buffers */
   u_int32              v_reserved[4];     /* reserved */
} rbf_drv_info;
```

The following table gives the RBF drive information structure.

Table 2-7. RBF Drive Information Structure

Name	Description
`v_0`	**ID Block Structure** This is a copy of the `idblock` structure from the identification sector of the media. The device driver must copy this information from the identification sector every time it is read.
`v_file_rsrc_lk`	**Open File List Lock Descriptor** Lock descriptor structure for locking the open file list.
`v_filehd`	**List of Path Descriptors** This field points to a list of path descriptors, representing the open files on the drive.
`v_free_rsrc_lk`	**Resource List Lock Descriptor** Lock descriptor structure for locking the allocatable resources list.
`v_free`	**List of Allocatable Resources** This field points to a data structure, representing the areas on the media free for allocation. RBF searches this data structure when it allocates space for a file.
`v_freesearch`	**Beginning of Free Memory** This field points to the part of the `v_free` data structure for RBF to start searching when it allocates space for a file.
`v_diskid`	**Disk ID** RBF copies the `diskid` field from the `idblock` and stores it in this field. It is used to detect when disks have been changed in a disk drive.
`v_mapseg`	**Allocation Bitmap Segment Information** This field contains the segment information for the RBF allocation map. RBF does not set this field until it needs the allocation map (for an allocation or de-allocation operation).
`v_bkzero`	**Identification Section Pointer** This is a pointer to a buffer containing the identification sector. Only the driver uses this field. RBF never accesses this field.
`v_trak`	**Current Track/Cylinder** This is the track/cylinder over which the head is currently positioned. Only the driver uses this field. RBF never accesses this field.
`v_softerr`	**Recoverable Error Count** This is the number of recoverable errors that have occurred on the drive and media. Only the driver uses this field. RBF never accesses this field.
`v_harderr`	**Non-Recoverable Error Count** This is the number of non-recoverable errors that have occurred on the drive and media. Only the driver uses this field. RBF never accesses this field.

Table 2-7. RBF Drive Information Structure (Continued)

Name	Description
`v_cache`	**Data Cache Pointer** This field points to the data caching structure, if caching is being used on the drive.
`v_crsrc_lk`	**Cache Data Lock Descriptor** Lock descriptor structure for locking the disk cache data structure.
`v_numpaths`	**Open Paths On Device** This is the number of open paths on the device.
`v_zerord`	**Block 0 Read Flag** RBF drivers use this flag to determine whether or not there is a valid sector 0 buffered. RBF never accesses this field.
`v_init`	**Initialized Drive Flag** This flag indicates that the device has been initialized. RBF drivers use this field to prevent themselves from initializing a device more than once.
`v_dopts`	**Copy of Path Descriptor Options** This is a copy of the path descriptor options. These are detailed in the following section.
`v_endflag`	**Byte Ordering Flag** This flag indicates the byte ordering used by the processor: BIG_END processor uses most significant byte first order LITTLE_END processor uses least significant byte first order
`v_fd_free_rsrc_lk`	**FD Free List Lock Descriptor** Lock descriptor structure for locking the FD free list.
`v_fd_free_list`	**List of Free FD Block Structures** This field points to the list of free file descriptor block structures.
`v_blks_rsrc_lk`	**Free Block List Lock Descriptor** Lock descriptor structure for locking the free block list.
`v_blks_list`	**List of Free Block Buffers** This field points to the list of free block buffers used for buffering data blocks.
`v_reserved`	**Reserved for Future Enhancements**

Disk Device Options

This section describes the definitions of the device options for RBF-type devices. The structure definition of the device options is shown here. This structure is defined in the header file `rbf.h`. IOMAN copies the device options from the device descriptor module into the logical unit static storage when the device is attached.

```
typedef struct rbf_lu_opts {
    u_char     lu_stp,          /* step rate */
                lu_tfm,          /* DMA transfer mode */
                lu_lun,          /* drive logical unit number*/
                lu_ctrlrid;      /* controller ID */
    u_int32    lu_totcyls;      /* total number of cylinders */
    u_int32    lu_reserved[4];  /* reserved for future expansion */
} rbf_lu_opts, *Rbf_lu_opts;
```

Table 2-8. RBF Disk Device Option Structure

Name	Description
`lu_stp`	**Step Rate (floppy disks)** This location contains a code that sets the head stepping rate used with the drive. Set the step rate to the fastest value the drive is capable of to reduce access time. These are the values commonly used: • `0 STEP_30MS` • `1 STEP_20MS` • `2 STEP_12MS` • `3 STEP_6MS`
`lu_tfm`	**DMA Transfer Mode** This is hardware specific. If available, the byte can be set for use of DMA mode. DMA requires only a single interrupt for each block of characters transferred in an I/O operation. It is much faster than methods that interrupt for each character transferred.
`lu_lun`	**Drive Unit Number** This number is used in the command block to identify the drive to the controller. The driver uses this number when specifying the device.
`lu_ctrlrid`	**SCSI Controller ID** This is the ID number of the controller attached to the drive. The driver uses this number when communicating with the controller.
`lu_reserved`	**Reserved for Future Enhancements**
`lu_totcyls`	**Cylinders On Device** This value is the actual number of cylinders on a partitioned drive. The driver uses this value to correctly initialize the drive.

Path Descriptor Options Table

The structure definition of the RBF path descriptor options is shown here. This structure is defined in the header file `rbf.h`.

```
typedef struct rbf_path_opts {
    u_int32    pd_sid,             /* number of surfaces */
               pd_vfy,             /* 0=verify disk writes */
               pd_format,          /* device format */
               pd_cyl,             /* number of cylinders */
               pd_blk,             /* default blocks/track */
               pd_t0b,          /* default blocks/track for trk0/sec0 */
               pd_sas,             /* segment allocation size */
               pd_ilv,             /* block interleave offset */
               pd_toffs,           /* track base offset */
               pd_boffs,           /* block base offset */
               pd_trys,            /* # tries */
               pd_bsize,           /* size of block in bytes */
               pd_cntl,            /* control word */
               pd_wpc,             /* first write precomp cylinder */
               pd_rwr,         /* first reduced write current cylinder */
               pd_park,            /* park cylinder for hard disks */
               pd_lsnoffs,         /* lsn offset for partition */
               pd_xfersize;    /* max transfer size in terms of bytes */
               pd_reserved[4]; /* reserved for future enhancements */
} rbf_path_opts, *Rbf_path_opts;
```

The following table lists the path descriptor options for RBF.

Table 2-9. RBF Path Descriptor Options Table Structure

Name	Description
`pd_sid`	Heads or Sides* This indicates the number of surfaces for a disk unit.
`pd_vfy`	Write Verification This field indicates whether a write is verified by a re-read and compare. If `pd_vfy` is: 0 Verify disk write 1 No verification NOTE: Write verify operations are generally performed on floppy disks but not hard disks because of the lower soft error rate of hard disks.

Table 2-9. RBF Path Descriptor Options Table Structure (Continued)

Name	Description
pd_format	**Disk Type*** OS-9 supports the following format definitions. These are defined in rbf.h: FMT_DBLTRK0 — Track 0 is double density. FMT_DBLBITDNS — Device is double bit density. FMT_DBLTRKDNS — Device is double track density. FMT_DBLSIDE — Device is double sided. FMT_EIGHTINCH — Drive is eight inch. FMT_FIVEINCH — Drive is five inch. FMT_THREEINCH — Drive is three inch. FMT_HIGHDENS — Device is high density. FMT_STDFMT — Device is standard format. FMT_REMOVABLE — Media can be removed. FMT_HARDISK — Device is a hard disk.
pd_cyl	**Cylinders** This is the number of cylinders per disk.
pd_blk	**Blocks/Track*** This is the number of blocks per track on all tracks except track 0.
pd_t0b	**Blocks/Track 0*** This is the number of blocks per track for track 0. This may be different than pd_blk (depending on the specific disk format).
pd_sas	**Segment Allocation Size** This value specifies the default minimum number of sectors to be allocated when a file is expanded.
pd_ilv	**Sector Interleave Factor*** Sectors are arranged on a disk in a certain sequential order (1, 2, 3, etc., 1, 3, 5, etc.). The interleave factor determines the arrangement. For example, if the interleave factor is 2, the sectors would be arranged by 2's (1, 3, 5, etc.) starting at the base sector (refer to pd_soffs).
pd_toffs	**Track Base Offset*** This is the offset to the first accessible track number. Because Track 0 is often a different density, Track 0 is sometimes not used as the base track.
pd_boffs	**Sector Base Offset*** This is the offset to the first accessible sector number. Sector 0 is not always the base sector.

Table 2-9. RBF Path Descriptor Options Table Structure (Continued)

Name	Description
`pd_trys`	**Number of Tries** This is the number of times a device tries to access a disk before returning an error.
`pd_bsize`	Logical Block Size* This is the logical block size in bytes.
`pd_cntl`	**Control Word** This is the control word. It may currently contain: CTRL_FMTDIS — Disables formatting of the device. CTRL_MULTI — Device is capable of multi-sector transfers. CTRL_AUTOSIZE — Device size can be obtained from device. CTRL_FMTENTIRE — Device requires only one format command. CTRL_TRKWRITE — Device needs a full track buffer for format.
`pd_wpc`	**Write Precompensation Cylinder** This number determines at which cylinder to begin write precompensation.
`pd_rwr`	**Reduced Write Current Cylinder** This number determines at which cylinder to begin reduced write current.
`pd_park`	**Park Cylinder** This is the cylinder at which to park the hard disk's head, when the drive is to be shut down.
`pd_lsnoffs`	**Logical Sector Offset*** This is the offset to be used when accessing a partitioned drive.
`pd_xfersize`	**Maximum Transfer Size** This is the maximum size of memory the controller can transfer at one time. The size is specified in bytes.

* This parameter is format specific.

Building RBF Device Descriptors

Making OS-9 device descriptors involves two steps:

Step 1. Modify the appropriate C macro definitions within the `RBF/<Driver>/config.des` for a specific device descriptor.

Step 2. Make the descriptor using the associated makefile.

The `config.des` file is organized so the macro definitions for a particular descriptor are grouped together. For example, the following section of `config.des` contains the macros that must be defined (this is, macros that do not have pre-defined defaults) for the RBF `ram` descriptor. They are grouped together within a C macro conditional.

```
/*****************************************************************
* Ram Device Default Definitions (All associated descriptors)    *
*****************************************************************/
/* Module header macros */
#define MH_EDIT     0x7

/* Device descriptor common macros */
#define PORTADDR    0
#define DRVR_NAME   “ram”
#define MODE        0xffff

/* rbf macros */
#define BLKSTRK   2048   /* multiplied by system wide BLKSIZE default   */
                        /* of 256 will equal 512 KByte ram disk.   */
/*****************************************************************
* End of Ram Device Default Definitions                          *
****************************************************************/

/*****************************************************************
* R0 Ram Descriptor Override Definitions                         *
**************************************************************/
#if defined (R0) /* R0 descriptor */
/* Module header macros */
#define MH_NAME_OVERRIDE    “r0”

/* Device descriptor common macros */
/* rbf macros */
#endif /* R0 descriptor */
```

Standard Device Descriptor Macros

This section discusses the standard macro definitions used for creating RBF device descriptors. Some of the macros have predefined values you can redefine in `RBF/<Driver>/config.des` file. Others must be defined for every device descriptor. Each discussion gives the name of the macro, an explanation of the macro, and an example definition (in many cases this is the default value set by Microware).

These five macros are common to RBF, SCF, and SBF descriptors.

Table 2-10. RBF, SCF, and SBF Common Descriptors

Name	Description and Example
`PORTADDR`	Controller Address This is the address of the device on the bus. This is the lowest address the device has mapped. Port address is hardware dependent. `#define PORTADDR    0xfffe4000`
`VECTOR`	Interrupt Vector This is the vector passed to the processor at interrupt time. Vector is hardware/software dependent. You can program some devices to produce different vectors. `#define VECTOR      80`
`IRQLEVEL`	Interrupt Level For the Device The number of supported interrupt levels is dependent on the processor being used. When a device interrupts the processor, the level of the interrupt is used to mask out lower priority devices. `#define IRQLEVEL    4`
`PRIORITY`	Interrupt Polling Priority This value is software dependent. A non-zero priority determines the position of the device within the vector. Lower values are polled first. A priority of 1 indicates the device desires exclusive use of the vector. A priority of 0 indicates the device wants to be the first device on the polling list. OS-9 does not allow a device to claim exclusive use of a vector if another device has already been installed on the vector. Additionally, it does not allow another device to use the vector once the vector has been claimed for exclusive use. `#define PRIORITY    10`
`LUN`	Logical Unit Number of the Device More than one device can have the same port address. The logical unit number distinguishes the devices having the same port address. `#define LUN    2    /* drive number */`

RBF Specific Macro Definitions

The following macros are specific to RBF:

Table 2-11. RBF Macro Definitions

Name	Description and Example
`STEP`	Step Rate This specifies the step rate to use on the RBF device. The following values are commonly used: Value — 5" and 3" disks — 8" disks 0 — 30ms — 15ms 1 — 20ms — 10ms 2 — 12ms — 6ms 3 — 6ms — 3ms Only the device driver uses the step rate value. The particular driver must determine the correspondence between the step value code and the step rate used on the drive. `#define STEP        3`
`SIDES`	Number of Heads or Sides This defines the number of heads on the drive. For example, a double-sided floppy drive would have a `SIDES` value of 2. `#define SIDES       2`
`VERIFY`	Write Verification Flag If set to a non-zero value, `VERIFY` indicates a read after write verify is desired. A zero value indicates no verify should be performed. It is up to the device driver to perform the verify. `#define VERIFY   0    /* no verify */`

Table 2-11. RBF Macro Definitions (Continued)

Name	Description and Example

`FORMAT`	Driver Format
	This defines the format of the drive described by the device descriptor. The definitions of the bits in the format word (16 bits) are defined in `rbf.h`:
	`#define FMT_DBLTRK0     0x0001`
	`   /* track 0 is double density */`
	`#define FMT_DBLBITDNS   0x0002`
	`   /* dev is double bit density */`
	`#define FMT_DBLTRKDNS   0x0004`
	`   /*dev is double track density*/`
	`#define FMT_DBLSIDE     0x0008`
	`   /* device is double sided */`
	`#define FMT_EIGHTINCH   0x0010`
	`   /* drive is eight inch */`
	`#define FMT_FIVEINCH    0x0020`
	`   /* drive is five inch */`
	`#define FMT_THREEINCH   0x0040`
	`   /* drive is three inch */`
	`#define FMT_HIGHDENS    0x1000`
	`   /* device is high density */`
	`#define FMT_STDFMT      0x2000`
	`   /* device is standard format */`
	`#define FMT_REMOVABLE   0x4000`
	`    /* media can be removed */`
	`#define FMT_HARDISK     0x8000`
	`   /* device is a hard disk */`
	`#define FORMAT`
	`FMT_STDFMT+FMT_FIVEINCH+FMT_DBLSIDE+FMT_DBLTRKDNS+`
	`FMT_DBLKTRK0`

`CYLNDRS`	Number of Cylinders This defines the number of cylinders on the drive. `#define CYLNDRS     80`
`BLKSTRK`	Blocks Per Track This defines the number of blocks per track on the drive on all tracks but track 0. `#define BLKSTRK     16`
`BLKSTRK0`	Blocks Per Track 0 This defines the number of blocks per track on track 0. Some floppy disk formats use a track 0 format that is different from the rest of the media so at least track 0 can be read. `#define BLKSTRK0    16`
`SEGSIZE`	Minimum Segment Allocation This defines the minimum number of blocks RBF should allocate when it is enlarging files. `#define SEGSIZE     1`

Table 2-11. RBF Macro Definitions (Continued)

Name	Description and Example
`INTRLV`	Block Interleave Factor This defines the physical interleave used when formatting the disk media. `#define INTRLV      2`

`DMAMODE`	DMA Transfer Mode
	This defines the type of DMA to be performed when transferring data to or from the disk device. Only the device driver uses this value.
	`#define DMAMODE    0`
	`/* DMA transfer mode */`

`TRKOFFS`	Track Offset
	This defines the track offset to use when accessing the device. If a track offset of one is used, for example, logical block 0 is the first block on side 0 of track (cylinder) one.
	`#define TRKOFFS   1`
	`/* one track offset */`

`BLKOFFS`	Block Offset
	This defines the offset to use when obtaining the physical block number for a device. A value of 1 indicates blocks are numbered from 1 to `BLKSTRK`. A value of 0 indicates blocks are numbered from 0 to `BLKSTRK` - 1.
	`#define BLKOFFS   1`
	`/* one block offset */`

`BLKSIZE`	Block Size
	This defines the size in bytes of the blocks used on the media.
	`#define BLKSIZE   256`
	`/* size of a block */`

`CONTROL`	Format Control Flags
	This defines the settings of various flags affecting the control of the device. The definitions of the flags are defined in `rbf.h`:
	`#define CTRL_FMTDIS    0x0001`
	`   /* device cannot be formatted */`
	`#define CTRL_MULTI     0x0002`
	`   /* can transfer multi sectors */`
	`#define CTRL_AUTOSIZE  0x0004`
	`   /* device can find its size */`
	`#define CTRL_FMTENTIRE 0x0008`
	`   /* can format entire device */`
	`#define CTRL_TRKWRITE  0x0010`
	`   /* do track writes for format */`
	`#define CONTROL        CTRL_MULTI`
	`   /* control word */`

`TRYS`	Number of Retries Before Error This defines the number of retries that should be performed before returning an error. `#define TRYS        7`

Table 2-11. RBF Macro Definitions (Continued)

Name	Description and Example
`SCSILUN`	SCSI Logical Unit Number This defines the SCSI logical unit number to be used by a device. Only the device driver uses this value. It can be used for things other than the SCSI logical unit number in the case of non-SCSI drivers. `#define SCSILUN     2`
`PRECOMP`	First Cylinder for Write Precompensation This defines the starting cylinder for write precompensation. Only the driver uses this value. `#define PRECOMP     CYLNDRS`
`REDWRITE`	First Cylinder for Reduced Write Current This defines the starting reduced write current cylinder. Only the driver uses this value. `#define REDWRITE    CYLNDRS`
`PARKCYL`	Park Cylinder This defines the cylinder where the read/write heads of the drive should be placed when an `_os_ss_sqd() setstat` is performed. Only the driver uses this value. `#define PARKCYL     0`
`LSNOFFS`	Logical Block Offset This defines the logical block offset to be used when accessing the device. This value is added to the logical block address RBF passes to the driver. Only the driver uses this value. `#define LSNOFFS     1`
`TOTCYLS`	Total Number of Cylinders on Drive This defines the total number of physical cylinders on the drive. This value is useful when working with physical drives that have been split in to a number of logical drives. Only the driver uses this field. `#define TOTCYLS     80`
`CTRLRID`	SCSI Controller ID This defines the SCSI controller ID for the device being accessed. Only the driver uses this field. You can use it for other purposes on non-SCSI devices. `#define CTRLRID     0`
`DRIVERNAME`	Name of Driver This defines the name of the RBF driver used to access the device described by the descriptor. `#define DRIVERNAME  "rb5400"`

Device Specific Non-Standard Definitions

In addition to the standard fields described in `rbf.des`, you can add specific definitions for particular driver/descriptor combinations. It is usually done to accommodate specific RBF drivers.

Complete the following steps for adding device specific information to a descriptor:

Step 1. Create an `editmod` source file with the structure definition of the additional information. For example, in `rbsccs.des`:

```
struct dev_specific {
    pointer u_int32 ds_ldrvnam = ldrvnam;
    u_int32 ds_scsiopts, "SCSI options"
};

string ldrvnam, "SCSI low-level driver name";
```

Step 2. Change the driver's header file to indicate the driver has device specific information:

```
#define DEV_SPECIFIC
#include <rbsccs.edm>                    /* include the editmod generated
header file */
typedef struct dev_specific dev_specific;  /* create dev_specific type */
```

Step 3. Add the header generation entry to the makefile for the driver. For example,

```
   rbsccs.h : rbsccs.edm

   rbsccs.edm : rbsccs.des
       $(EDITMOD) -h=dev_specific -o=rbsccs.edm rbsccs.des
```

Step 4. Ensure the driver's header file is included by `confi.des` when the descriptor is made. Add an `#include` statement if necessary.

After following these steps, make the descriptor using the descriptor makefile.

3 Sequential Character File Manager (SCF)

This chapter describes how to create an SCF device descriptor and device driver. The following sections are included:

Overview

The Sequential File Manager (SCF) is a re-entrant subroutine package for I/O service requests to devices that operate on a character-by-character basis (terminals, printers, modems, etc.). SCF can support any number of SCF-type devices. In addition, SCF contains input and output editing functions to aid in typical line-oriented operations.

The I/O service requests applicable to SCF are listed below:

`I_ATTACH`	`I_CLOSE`
`I_CREATE`	`I_DETACH`
`I_DUP`	`I_GETSTAT`
`I_OPEN`	`I_READ`
`I_READLN`	`I_SETSTAT`
`I_WRITE`	`I_WRITLN`

The I/O service requests, when made to SCF, return the appropriate error code.

SCF is responsible for arbitrating I/O requests to devices and performing line editing functions. SCF device drivers are responsible for the actual transfer of data to and from the device hardware. The device driver transfers data to and from the unit's input/output buffer. SCF transfers the data to and from the units input/output buffer to the process's data buffer.

SCF device drivers that support hardware with interrupt capability transfer data to and from the unit's buffers asynchronously. The device driver is driven by interrupts from the device and needs no direct communication with the file manager to pass data between the unit's buffers and the hardware. An interrupt driven device driver is only directly called by SCF when the device is attached (to initialize the hardware), when the device is detached (to deinitialize the hardware), when the transmittor interrupts need to be enabled (when data is available for transmission), and for some select getstat/setstat service requests. SCF calls the device driver for each character read/written to the de vice in the case that the device is being operated in polled mode.

Creating an SCF Driver/Descriptor

This section summarizes the steps required to build a device descriptor for a new board. You will be referred to more detailed procedures for the specific steps involved in writing an SCF device driver and building a descriptor if Microware does not supply one you can use.

Step 1. Create a `<driver>` directory in the port-specific `<target>/SCF` directory where `<driver>` is the name of the serial device driver chip on your board.

Step 2. Create `DRVR` and `DESC` directories in the `<target>/SCF/<Driver>` directory, along with makefiles, to build the drivers and descriptors. You can use the example makefiles as a reference.

Step 3. Check the Microware-supplied driver and driver-specific descriptor sources included in the `MWOS/OS9000/SRC/IO/SCF/DRVR` directory for one based on the same target device your platform uses.

Step 4. If you find a driver matching the chip on your board, check the makefiles (copied from examples in Step 2) to make certain they point to the correct source files for the device driver. Proceed to Step 6.

Step 5. Create a new directory in `MWOS/OS9000/SRC/IO/SCF/DRVR` for the device driver.

Step 6. Build a new device descriptor for the driver.

Step 7. Set up the proper configuration labels for the device within the `systype.h` file for the driver and the configuration files for the descriptor.

Creating SCF Device Drivers

This section describes the data structures and subroutines comprising an SCF device driver. The first section describes the driver's static storage structure definition and the second section describes the subroutines required for an SCF driver.

Before you write a device driver, you should understand how the driver uses the device descriptor. For this information, refer to the Using SCF Device Descriptor Modules section.

SCF Device Driver Static Storage

This section describes the device driver's static storage structure definition. The structure definition of the driver static storage is found in `scf.h` and shown on the following page. This structure contains the information SCF needs to initialize and call the device driver.

Like all other OS-9 device drivers, SCF device drivers use a standard executable memory module format with a module type of device driver. Every driver maintains a driver static storage area for each device with a unique port address. The driver static storage area always contains the following five items:

1. The first seven long words of the driver's dispatch table structure must contain the address of the standard driver functions. SCF drivers may declare additional variables separate from this structure, but it is critical that this structure be identified as the sharable portion of the driver's static storage by equating the name of the structure with the `_m_share` label. This portion of every SCF driver static storage must be the same.
2. A variable used by drivers to keep track of the number of times the driver has been attached. This variable can determine when to properly terminate the device.
3. A pointer to the device list entry for the specific device.
4. The number of interrupt service routines for the driver.
5. A table of interrupt service routine entries containing the hardware vector offset of the associated interrupt and the address of the service routine completes the driver static storage.

SCF assumes the first interrupt entry in the table is the input interrupt service routine.

The static storage of the driver is a combination of the driver static storage structure and any other variables the driver declares.

IOMAN allocates and initializes the driver's entire static storage at attach time and also performs the following functions:

- Locates the driver's dispatch table structure within the driver's static storage information by using the m_share field of the driver's module header.
- Adds this offset value to the beginning of the driver's static storage to locate the shared structure. This value is contained in the v_dr_stat field in the device list entry associated with the device and is used by SCF in calling the driver.

```
typedef struct scf_drvr_stat {
  error_code  (*v_init)(),
                                /* address of driver’s init function */
        (*v_read)(),            /* address of driver’s read function */
        (*v_write)(),           /* address of driver’s write function */
        (*v_getstat)(), /* address of driver’s get_status function */
        (*v_setstat)(),/* address of driver’s put_status function */
        (*v_terminate)(),/* address of driver’s terminate function */
        (*v_entxirq)();    /* address of driver’s "entxirq" function */
                              /* i.e. (enable transmitter interrupts) */
    Dev_list v_dev_entry;/* device list entry pointer for device */
                         /* (initialized by SCF before calling drvr) */
    u_int16  v_attached,/* driver attached flag (maintained by drvr) */
             v_rsrvd[7];             /* reserved for future use */
    u_int32  v_irqcnt;   /* number of interrupt service routines */
    irq_entry  v_irqrtns[8];
                                /* interrupt service routine entries */
} scf_drvr_stat;
```

The following table describes the device driver storage for SCF.

Table 3-1. SCF Device Driver Static Storage

Name	Description
v_init	This field contains the address of the driver's initialization routine. The initialization routine is responsible for performing the actual initialization of the device hardware. SCF calls this routine when an _os_attach() service request is made.
v_read	This field contains the address of the driver's read routine. The driver's read routine is only called if the driver's input operates in polled mode. For more information, refer to the v_pollin field of the logical unit static storage structure definition (Table 3-4).

Table 3-1. SCF Device Driver Static Storage (Continued)

Name	Description
`v_write`	This field contains the address of the driver's write routine. The driver's write routine is only called if the driver's output operates in polled mode. For more information, refer to the `v_pollout` field of the logical unit static storage structure definition (Table 3-4).
`v_getstat`	This field contains the address of the driver's get status routine. The driver's get status routine is only called for `getstat` service requests that are defined to call the driver and unknown `getstat` function codes.
`v_setstat`	This field contains the address of the driver's set status routine. The driver's set status routine is only called for `setstat` service requests that are defined to call the driver and unknown `setstat` function codes.
`v_terminate`	This field contains the address of the driver's terminate routine. The terminate routine is responsible for performing the actual de-initialization of the device hardware. SCF calls this routine when an `_os_detach()` service request is made.
`v_entxirq`	This field contains the address of the driver's enable transmit interrupts routine. The enable routine is responsible for enabling the device's transmitter interrupts so the device can begin its asynchronous output. SCF only calls the enable routine when data is available for transmission and the transmit interrupts are disabled. For more information, refer to the `v_outhalt` field in the logical unit static storage structure definition (Table 3-4).
`v_dev_entry`	This field points to the device list entry for this device.
`v_attached`	The driver uses this field to keep track of the number of attach operations performed on the device. The driver should increment it every time the init routine is called, and decrement it for every terminate call. This allows the driver to know when it should truly initialize/de-initialize the hardware.
`v_irqcnt`	This fields specifies the number of interrupt service routines required by the device driver.

Table 3-1. SCF Device Driver Static Storage (Continued)

Name	Description
`v_irqrtns`	This array contains the addresses and associated vector numbers of the interrupt service routines of the driver. The driver may use this array to install its interrupt service routines on the system's interrupt polling table. The first entry (if any) is the read IRQ. The second entry (if any) is the write IRQ. This array contains the addresses and associated vector number offsets (from the base vector number of the device) of the interrupt service routines of the driver. The base vector number is usually zero. However, for some smart devices, there can be multiple IRQs. The actual vector number value the driver uses to install the routine on the system polling table is the sum of the vector number in the logical unit static storage (`v_vector`) and the routine vector offset.
`v_rsrvd`	This array is reserved for future use.

SCF Device Driver Entry Subroutines

The standard driver subroutines and their parameters follow:

Table 3-2. SCF Subroutines

Function	Description
`ENABLE TRANSMITTER INTERRUPTS`	Enable the device's "ready-to-transmit" interrupts.
`GETSTAT`	Get device status.
`INIT`	Initalize device hardware.
`IRQ SERVICE ROUTINE`	Service device interrupts.
`READ`	Read next character.
`SETSTAT`	Set device status.
`TERMINATE`	Terminate device.
`WRITE`	Write a character.

ENABLE TRANSMITTER INTERRUPTS

Enable the Device's Ready to Transmit Interrupts

Syntax

```
error_code entxirq(Dev_list device_entry);
```

Description

The enable transmitter interrupts routine is called by SCF and the driver when there is data in the output buffer and the `v_outhalt` field of the output device's logical unit static storage indicates the ready to transmit interrupts are disabled. The init routine should initialize this field and the interrupt service routine(s) should maintain it properly. Also, the enable transmitter interrupts routine should flag that the transmitter interrupt is enabled by setting the `OH_IRQON` bit of the `v_outhalt` field.

Parameters

`device_entry`
    points to the device list entry.

GETSTAT

Get Device Status

Syntax

```
error_code getstat(
  I_getstat_pb                  ctrl_block,
  Scf_path_desc                 path_desc,
  Dev_list                      device_entry);
```

Description

These routines are wildcard calls used to get the device parameters specified by the `getstat` service requests. Many SCF-type requests are handled by IOMAN or SCF. Any `getstat` functions not handled by them are passed to the device driver. If the function code specified in the control block is not recognized by the driver, the driver returns an `EOS_UNKSVC` (unknown service code) error.

Parameters

`ctrl_block`
: is the `I_GETSTAT` control block.

`path_desc`
: points to the path descriptor.

`device_entry`
: points to the device entry.

INIT
Initialize Device Hardware

Syntax

```
error_code init(Dev_list device_entry);
```

Description

The INIT routine must:

1. Install driver interrupt service routine(s) on the system interrupt polling table using the `_os_irq()` service request.
2. Initialize the device control registers with the functionality specified by the logical unit options section.
3. Output a null byte to get the transmitter interrupts activated. The transmitter interrupts should not actually be enabled until later when SCF has data to output and calls the driver's "enable-transmitter interrupts" routine.

Parameters

`device_entry`
    is the device list entry for the device.

IRQ SERVICE ROUTINE

Service Device Interrupts

Syntax

```
error_code input_irq(Dev_list device_entry);
```

```
error_code output_irq(Dev_list device_entry);
```

Description

The interrupt service routine is not included in the driver's entry point table and not called directly by SCF. It functions as follows:

1. Query device to determine if it caused the interrupt. If the device did not cause the interrupt, exit immediately with an `EOS_NOTME` error.
2. Service the device interrupt (receive/transmit data). This routine puts its data into or get its data from the buffers defined in the logical unit static storage.
3. Wake up any process waiting for I/O to complete by checking the `v_wake` field of the logical unit static storage. It sends a wakeup signal to the process specified by this field and then clears the field.
4. If the device is ready to send (assuming it is servicing an output interrupt) and the output buffer is empty, it disables the device's ready- to-transmit interrupts. It also flags the interrupts as disabled by clearing the `OH_IRQON` bit and setting the `OH_EMPTY` bit of the `v_outhalt` flag field of the logical unit static storage.
5. If a pause character is received, sets the `v_pause` field of the logical unit static storage of the output device to a non-zero value.
6. If a keyboard interrupt or keyboard quit character is received, sends the associated signal to the process specified in the `v_lproc` field of the logical unit static storage.
7. If an X-ON or X-OFF character is received, enables or disables transmitter interrupts.
8. If the input buffer has reached the "high water mark" as specified by the `v_maxbuff` field of the logical unit static storage and X-OFF is enabled, prepares to send an X-OFF character.

Parameters

`device_entry`
    points to the device list entry.

READ
Read Next Character

Syntax

```
error_code read(
  Scf_path_desc                  path_desc,
  Dev_list                       device_entry);
```

Description

The READ routine for drivers that have interrupt driven input returns without error. The read routine for drivers with polled input performs the same functions as an input interrupt service routine (X-ON/X-OFF flow control, keyboard interrupt, keyboard quit) except it polls the hardware for the next character.

Parameters

`path_desc`
: points to the path descriptor.

`device_entry`
: points to the device list entry.

SETSTAT
Set Device Status

Syntax

```
error_code setstat(
  I_setstat_pb                     ctrl_block,
  Scf_path_desc                    path_desc,
  Dev_list                         device_entry);
```

Description

Setstats are wildcard calls that set the device parameters specified by the `setstat` service requests. Many SCF-type requests are handled by IOMAN or SCF. Any `setstat` functions not handled by them are passed to the device driver. If the function code specified in the control block is not recognized by the driver, the driver returns an `EOS_UNKSVC` (unknown service code) error.

Parameters

`ctrl_block`
: is the `I_SETSTAT` control block.

`path_desc`
: points to the path descriptor.

`device_entry`
: points to the device entry.

TERMINATE
Terminate Device

Syntax

```
error_code terminate(Dev_list device_entry);
```

Description

The terminate routine performs the following functions:

1. De-initializes the hardware, disabling the device interrupts.
2. Removes the interrupt service routine(s) from the system interrupt polling table.

Parameters

`device_entry`
points to the device list entry.

WRITE

Write a Character

Syntax

```
error_code write(
  Scf_path_desc                     path_desc,
  Dev_list                          device_entry);
```

Description

The write routine for drivers that have interrupt driven output returns without error. The write routine for drivers with polled output performs the same functions as an output interrupt service routine except it polls the hardware to transmit the next character.

Parameters

`path_desc`
: points to the path descriptor.

`device_entry`
: points to the device entry.

Using SCF Device Descriptor Modules

The SCF device descriptor consists of four parts:

- OS-9 module header
- common information required by IOMAN for all descriptors
- path descriptor options
- logical unit static storage

Along with the common IOMAN information, the `scf_desc` structure contains the offset for the name of an output device to be used, if different from the input device. SCF examines this structure when processing an `OPEN` request.

This structure is defined in `scf.h`:

```
typedef struct scf_desc {
    dd_com
        dd_descom;      /* common device descriptor variables */
    u_int32
        dd_outdev;      /* alternate output device name offset */
    u_int16
        dd_rsvd_scf[2]; /* reserved space */
} scf_desc;
```

Table 3-3. SCF Device Descriptors

Name	Description
`dd_descom`	This is the common information structure IOMAN requires be in all device descriptors.
`dd_outdev`	This is the offset to the name of the output device to be used instead of the input device. If this field is non-zero, SCF attaches to this device, initializes it, and issues an `SS_OPEN setstat` request to the device's driver.

SCF Logical Unit Static Storage

This section describes the definitions of the logical unit (device) static storage area for SCF-type devices. The structure definition of the device static storage is found in `scf.h`. IOMAN copies the initial values from the device descriptor module into the logical unit static storage when a path to the device is opened. This structure contains the important variables used by the device driver and SCF to communicate and transfer data.

Device Static Storage Structure Definition Example

```
typedef struct scf_lu_stat {
  hardware_vector      v_vector;       /* IRQ vector number */
  u_char      v_irqlevel,         /* IRQ interrupt level */
              v_priority,         /* IRQ polling priority */
              v_pollin,           /* polled input flag; 1=polled, 0=IRQ
                                  driven */
              v_pollout,          /* polled output flag; 1=polled, 0=IRQ
                                  driven */
              v_inhalt,           /* input halted flag */
              v_hangup,       /* set non-0 when data carrier is lost */
              v_outhalt;        /* output IRQ's disabled when non-zero */
  u_int16     v_lu_num,           /* logical unit number */
              v_wait;           /* indicates process is waiting on I/O */
  u_int32     v_irqmask,          /* Interrupt mask word */
              v_savirq_fm,        /* previous interrupt mask word (SCF
                                  only) */
              v_savirq_dv,        /* prev. interrupt mask word (driver
                                  only) */
              v_savirq_ll;        /* reserved for future use */
  process_id  v_wake,           /* ID of process waiting I/O operation */
              v_busy,           /* ID of process currently using device */
              v_lproc,            /* # of last process to use this unit */
              v_sigproc[3],       /* process to signal on SS_SENDSIG
                                  request; signal code; associated
                                  (system) path # */
              v_dcdoff[3],        /* process to signal on SS_DCOFF
                                  request; signal code; associated
                                  (system) path # */
              v_dcdon[3];       /* process to signal on SS_DCON request;
                                signal code; associated (system) path # */
  Scf_lu_stat  v_outdev;          /* output device's static storage
                                  pointer */
  u_int32     v_pdbufsize,        /* SCF's path buffer size for this
                                  device */
              v_maxbuff;          /* input buffer maximum (high water
                                  mark) */
  u_int32     v_insize,           /* size of input buffer */
              v_incount;          /* number of bytes in input buffer */
  u_char      *v_inbufad,         /* input buffer address */
              *v_infill,          /* input buffer next-in pointer */
              *v_inempty,         /* input buffer next-out pointer */
              *v_inend;           /* input buffer end of buffer pointer */
  u_int32     v_outsize,          /* size of output buffer */
              v_outcount;         /* number of bytes in output buffer */
  u_char      *v_outbufad,        /* output buffer address */
              *v_outfill,         /* output buffer next-in pointer */
              *v_outempty,        /* output buffer next-out pointer */
              *v_outend;         /* output buffer end of buffer pointer */
  lock_id     v_lockid;           /* I/O lock identifier */
  u_int32     v_use_cnt;          /* logical unit user count */
  u_int32     v_resrvd[5];        /* reserved space */
  Scf_path_opts  v_pdopt;         /* ptr to path descriptor options
                                  section */
  scf_lu_opts    v_opt;           /* logical unit options section */
#ifdef DEV_SPECIFICS
  DEV_SPECIFICS                  /* driver specific static variables */
#endif
} scf_lu_stat;
```

The following table describes the storage fields for SCF.

Table 3-4. SCF Logical Unit Static Storage Fields

Name	Description
`v_vector`	**Interrupt Vector** This field contains the associated interrupt vector number for the device. Note: The OS-9 *Configuration Reference* uses `hardware_vector`.
`v_irqlevel`	**Interrupt Level** This field contains the interrupt level of the device.
`v_priority`	**Interrupt Priority** This field contains the polling priority of the device.
`v_pollin`	**Polled Input Flag** This field indicates whether the device's input operates in interrupt or polled mode. If the driver uses polled mode, SCF calls the driver's `READ` routine for every character. A non-zero value indicates polled mode. A zero value indicates interrupt driven input.
`v_pollout`	**Polled Output Flag** This field indicates whether the device's output operates in interrupt or polled mode. If the driver uses polled mode, SCF calls the driver's `WRITE` routine for every character. A non-zero value indicates polled mode. A zero value indicates interrupt driven output.

Table 3-4. SCF Logical Unit Static Storage Fields (Continued)

Name	Description
`v_inhalt`	**Input Halted Flag** This field indicates whether or not input to the device has been halted. It is non-zero if an X-OFF character has been sent and input halted.
`v_hangup`	**Data Carrier Lost Flag** This field is non-zero when the *data carrier* line has been lost, indicating a lost connection.
`v_outhalt`	**Output Halt Flag** This field indicates the status of output from the device. SCF uses this field to decide when to call the driver's "enable transmit IRQ" routine to begin output. Bits 2 - 4 are undefined. Bits 5 and 6 are user-definable. Bits 0, 1, and 7 are defined as follows: Bit 0 (0x01) — indicates an X-OFF has been received and output has been halted. Bit 1 (0x02) — indicates the output buffer is empty and output has been halted. Bit 7 (0x80) — indicates transmitter interrupts are enabled. It is important that the device driver clears this bit whenever definitions for these bits are in the `scf.h` header file
`v_lu_num`	**Logical Unit Number** This field contains the logical unit number. NOTE: The OS-9 *Configuration Reference* uses `v_lun`.
`v_wait`	**I/O Wait Flag** This field indicates whether a process is waiting for I/O on this logical unit. Definitions for this field are located in the `scf.h` header file. The values of this field are defined as follows: `0` — No processes waiting on the device. `1` — A process is waiting on input to the device. `2` — A process is waiting on output from the device.
`v_irqmask`	**Interrupt Mask** This field contains the interrupt mask used for masking interrupts to the level of the device. NOTE: Interrupts should be masked as little as possible and only for critical sections of the device driver.
`v_savirq_fm`	**Previous Interrupt Status (SCF use)** SCF uses this field for saving the current state of the interrupt status register prior to masking interrupts.

Table 3-4. SCF Logical Unit Static Storage Fields (Continued)

Name	Description
`v_savirq_dv`	**Previous Interrupt Status (Driver use)** SCF device drivers use this field for saving the current state of the interrupt status register prior to masking interrupts.
`v_wake`	**Waiting Process ID** This field contains the process identifier of any process waiting for the device to complete I/O. `0` indicates there is no process waiting.
`v_busy`	**Current Process ID** This field contains the process identifier of the process currently using the device. SCF uses this field to prevent more than one process from using the device at a time. NOTE: `v_busy` is always equal to `v_lproc` or is zero.
`v_lproc`	**Last Process ID** This field contains the process identifier of the last process to use the device. The interrupt service routine sends this process the proper signal when an "interrupt" or "quit" character is received.
`v_sigproc`	**Signal Process Information (for data ready)** This field contains the process identifier, the signal code to send, and the associated system path number for the process that made an `SS_SENDSIG setstat` call (send signal on data ready).
`v_dcdoff`	**Signal Process Information (for DCD false)** This field holds the process identifier, the signal code to send, and the associated system path number for the process that made an `SS_DCOFF setstat` call (send signal on DCD false).
`v_dcdon`	**Signal Process Information (for DCD true)** This field holds the process identifier, the signal code to send, and the associated system path number for the process that made an `SS_DCON setstat` call (send signal on DCD true).
`v_outdev`	**Output Device Static Storage Pointer** This points to the logical unit static storage structure of the output (echo) device. In most cases, a device is its own echo device. However, it may not be, as in the case of a keyboard and a memory mapped video display.
`v_pdbufsize`	**Path Buffer Size** This field contains the size of the path buffer SCF uses for this device.
`v_maxbuff`	**Maximum Data For Path Buffer** This field is a high water marker for the path buffer. The device driver should send an X-OFF character to the transmitter when the path buffer fills up to this point.

Table 3-4. SCF Logical Unit Static Storage Fields (Continued)

Name	Description
`v_insize`	**Device Input Buffer Size** This field contains the size of the input buffer for this device (logical unit).
`v_incount`	**Current Byte Count in Input Buffer** This field contains the number of bytes currently in the input buffer. The device driver updates this field as it places characters in the input buffer. SCF updates this field when it removes characters from the input buffer.
`v_inbufad`	**Beginning of Input Buffer Pointer** This field contains a pointer to the beginning of the input buffer for this logical unit.
`v_infill`	**Next Data Input Pointer (to Input Buffer)** This field contains a pointer to the "next-in" position for the input buffer for this logical unit. The device driver uses and maintains this pointer to place characters in the input buffer.
`v_inempty`	**Next Data Output Pointer (from Input Buffer)** This field contains a pointer to the "next-out" position for the input buffer for this logical unit. SCF uses and maintains this pointer to remove characters from the input buffer.
`v_inend`	**End of Input Buffer Pointer** This field contains a pointer to the end of the input buffer for this logical unit.
`v_outsize`	**Output Buffer Size** This field contains the size of the output buffer for this logical unit.
`v_outcount`	**Current Byte Count in Output Buffer** This field contains the number of bytes currently in the output buffer. SCF updates this field as it places characters in the output buffer. The device driver updates this field as it removes characters from the output buffer.
`v_outbufad`	**Beginning of Output Buffer Pointer** This field contains a pointer to the beginning of the output buffer for this logical unit.
`v_outfill`	**Next Data Input Pointer (to Output Buffer)** This field contains a pointer to the next-in position for the output buffer for this logical unit. SCF uses and maintains this pointer to place characters in the output buffer.
`v_outempty`	**Next Data Output Pointer (from Output Buffer)** This field contains a pointer to the next-out position for the output buffer for this logical unit. The device driver uses and maintains this pointer to remove characters from the output buffer.

Table 3-4. SCF Logical Unit Static Storage Fields (Continued)

Name	Description
v_outend	**End of Output Buffer Pointer** This field contains a pointer to the end of the output buffer for this logical unit.
v_lockid	**Resource Lock ID** This field contains the resource lock identifier for this logical unit. SCF uses this field to arbitrate exclusive access to this logical unit.
v_use_cnt	**Logical Unit User Counter** This field can be used by the driver to record the number of users using a given logical unit. This provides better control over devices supporting more than one unit.
v_pdopt	**Path Descriptor Option Pointer** This field contains a pointer to the path descriptor options section for this path.
v_opt	**Logical Unit Options** This field is the structure containing the logical unit options for this logical unit. These options are described following this section.
DEV_SPECIFICS	**Device Specific Variable MACRO** This is a C language macro. The author of a device driver can expand this macro to include additional variables in the logical unit static storage structure. The additional field can be defined in a header file for the device driver being written. The fields are included in the structure when the device descriptor for the logical unit is created.

SCF Logical Unit Static Storage Options

This section describes the definitions of the device options (logical unit options) for SCF-type devices. The structure definition of the device options is shown here. This structure is defined in `scf.h`. IOMAN copies the device options from the device descriptor module into the logical unit static storage when a path to the device is attached. The device options may be changed afterwards using the `SS_LUOPT` function of `I_GETSTAT` and `I_SETSTAT` service requests or from the keyboard using the `xmode` utility.

```
typedef struct scf_lu_opts {
  u_int16     v_optsize;       /* size of logical unit options section */
  u_char      v_class,         /* device type; 0 = SCF */
              v_err,           /* accumulated errors */
              v_pause,         /* immediate pause request */
              v_line,          /* lines left until end of page */
              v_intr,          /* keyboard interrupt character */
              v_quit,          /* keyboard quit character */
```

```
             v_psch,            /* keyboard pause character */
             v_xon,             /* X-ON character */
             v_xoff,            /* X-OFF character */
             v_baud,            /* baud rate */
             v_parity,          /* parity */
             v_stopbits,        /* stop bits */
             v_wordsize,        /* word size */
             v_rtsstate,        /* RTS state: disable = 0; enable = non-
                                zero */
             v_dcdstate,        /* current state of DCD line */
             v_reserved[9];     /* reserved for future use */
} scf_lu_opts;
```

Table 3-5. SCF Logical Unit Static Storage Options

Name	Description
`v_optsize`	**Options Section Size** This field specifies the size of the logical unit options section.
`v_class`	**Device Type (DT_SCF = 0)** This field specifies the device type. This should be zero for SCF. The device types are defined in the `io.h` header file.
`v_err`	**Accumulated Errors** This field is used to accumulate I/O errors. Typically, the IRQ service routine uses it to record errors so they can be reported later when SCF calls one of the device driver routines.
`v_pause`	**Pause Flag** This field tells SCF when there is an immediate pause request from the input device. It causes SCF to suspend output from `_os_writeln()` until a character is entered from the input device.
`v_line`	**Lines Before End of Page** This field contains the number of lines left to output until a page pause occurs (end-of-page).
`v_intr`	**Keyboard Interrupt Character** This field specifies the keyboard interrupt character. When a keyboard interrupt character is entered, a keyboard interrupt signal is sent to the last user of this unit. It terminates the current I/O request (if any) with an `EOS_BSIG` error. This field is normally set to `<Ctrl>C`.
`v_quit`	**Keyboard Quit Character** This field specifies the keyboard quit character. When a keyboard quit character is entered, a keyboard quit signal is sent to the last user of this unit. It terminates the current I/O request (if any) with an `EOS_BSIG` error. This field is normally set to a `<Ctrl>E`.

Table 3-5. SCF Logical Unit Static Storage Options (Continued)

Name	Description
`v_psch`	**Keyboard Pause Character** This field specifies the keyboard pause character. When this character is entered during output, output is suspended before the next end-of-line. This also deletes any "type ahead" input for `_os_readln()`.
`v_xon`	**X-ON Character** This field specifies the transmit on (X-ON) character. When this character is received, output is resumed, assuming it was suspended by a transmit off character. NOTE: X-ON and X-OFF are required for software handshaking for some devices.
`v_xoff`	**X-OFF Character** This field specifies the transmit off (X-OFF) character. When this character is received, output is suspended until a transmit on character is received. NOTE: X-ON and X-OFF are required for software handshaking for some devices.
`v_baud`	**Baud Rate** This field sets the baud rate as follows:

0x00 = Hardwired	0x10 = 19200 baud
0x01 = 50 baud	0x11 = 31250 baud
0x02 = 75 baud	0x12 = 38400 baud
0x03 = 110 baud	0x14 = 57600 baud
0x04 = 134.5 baud	0x15 = 64000 baud
0x05 = 150 baud	0x16 = 115200 baud
0x06 = 300 baud	0x17 = 230400 baud
0x07 = 600 baud	0x18 = 460800 baud
0x08 = 1200 baud	0x19 = 921600 baud
0x09 = 1800 baud	0x1a = 26800 baud
0xA = 2000 baud	0x1b = 153600 baud
0xB = 2400 baud	0x1c = 307200 baud
0xC = 3600 baud	0x1d = 614400 baud
0xD = 4800 baud	0x1e = 1228800 baud
0xE = 7200 baud	0xff = External
0xF = 9600 baud	

Table 3-5. SCF Logical Unit Static Storage Options (Continued)

Name	Description
`v_parity`	**Parity** This field specifies the parity to be used. 0 = no parity 1 = odd parity 2 = even parity 3 = mark parity 4 = space parity
`v_stopbits`	**Stop Bits** This field specifies the number of stop bits to be used. 0 = 1 stop bit 1 = 1 1/2 stop bits 2 = 2 stop bits
`v_wordsize`	**Bits Per Character** This field specifies the number of bits per character.
`v_rtsstate`	**RTS Line State** This field controls the state of the RTS line. It is useful for drivers wanting to use hardware handshaking. When this field is zero, the RTS line is disabled. When it is non-zero, it is enabled.
`v_dcdstate`	**Current DCD Line State** This field indicates the state of the DCD Line.

SCF Path Descriptor

This section describes the definitions of the path descriptor for SCF-type devices. The structure definition of the path descriptor is shown here. This structure is defined in `scf.h`. SCF initializes the path descriptor options section from the specified device descriptor module when a path is opened to an SCF device. The path descriptor options can later be changed using the `_os_gs_popot()`/`_os_ss_popt()` service requests or from the keyboard using the `tmode` utility.

```
typedef struct scf_path_desc {
  struct      pathcom pd_common;  /* common path descriptor structure */
  Dev_list    pd_outdev;          /* device tbl pointer for echo device */
  u_char      *pd_ubuf,           /* user buffer base address */
              *pd_pbuf,           /* path buffer base address */
              *pd_pbufpos;        /* current path buffer position */
  u_int32     pd_endobuf,         /* end of buffer position */
              pd_curpos,          /* cursor position counter */
              pd_reqcnt,      /* number of bytes requested by the caller */
              pd_evl;             /* readln end of visible line counter */
  u_char      pd_echoflag,   /* flag if echoing output is ok for this device */
              pd_lost;            /* non-zero if path has become dead */
                                  /* (ie: data-carrier-detect lost) */
  u_int16     pd_reserved[7];     /* reserved space */
  scf_path_opts    pd_opt;        /* SCF path descriptor options */
} scf_path_desc;
```

The following table lists the SCF path descriptor fields.

Table 3-6. SCF Path Descriptor Fields

Name	Description
`pd_common`	**Common Path Descriptor Variables** This field is the structure containing the path descriptor variables IOMAN requires for all path descriptors. These variables are described in the first chapter of this manual.
`pd_outdev`	**Device Table Pointer for Echo Device** SCF uses this field for calling the echo device to echo input and to output characters in polled mode.
`pd_ubuf`	**User Buffer Base Address** This field saves the user's buffer pointer on `_os_read()`, `_os_readln()`, `_os_write`, and `_os_writeln` requests.
`pd_pbuf`	**Path Buffer Base Address** This field points to the input buffer for the path. It is the buffer associated with input and its editing functions.
`pd_pbufpos`	**Current Path Buffer Position** This field points to the current input position in the path buffer.
`pd_endobuf`	**End of Buffer Position** This field contains the last character position of the path buffer.
`pd_curpos`	**Cursor Position Counter** SCF uses this field to maintain the current location of the cursor on input.
`pd_reqcnt`	**Number of Bytes Requested** This field contains the total number of bytes requested on a read or readln request.
`pd_evl`	**End of Visible Line Counter** SCF uses this field to maintain the logical end-of-visible line when performing the editing functions of a `_os_readln` request.
`pd_echoflag`	**Echo Output Flag** A non-zero value in this field indicates echoing input is enabled.
`pd_lost`	**Data Carrier Detect Lost Flag** A non-zero value in this field indicates a transition of the data carrier detect line. This is useful for modem support.
`pd_opt`	**Path Descriptor Options** This field is the structure containing the path descriptor options. These options are described in the following section.

SCF Path Descriptor Options Section

The structure definition of the path descriptor options is shown here. This structure is defined in the header file `scf.h`. You can update the path descriptor options using the `_os_gs_popt()`/`_os_ss_popt()` system calls or the `tmode` utility.

```
typedef struct scf_path_opts {
  u_int16     pd_optsize,         /* path options table size */
              pd_extra;           /* reserved for future use */
  inmap_entry  pd_inmap[32];/* Input control character mapping table */
  u_char      pd_eorch,          /* end of record character (read only) */
              pd_eofch,           /* end of file character */
              pd_tabch,           /* tabulate character (0 = none) */
              pd_bellch,          /* bell character (for input line
                                  overflow) */
              pd_bspch;           /* backspace echo character */
  u_char      pd_case,            /* case 0 = both ~0 = upper case only */
              pd_backsp,          /* backspace 0 = backspace ~0 =
                                  backspace,space,backspace */
              pd_delete,          /* delete 0 = carriage return, line fee
                                  ~0 = backspace over line */
              pd_echo,            /* echo 0 = no echo */
              pd_alf,            /* auto-linefeed 0 = no auto line feed */
              pd_pause,           /* pause 0 = no end of page pause */
              pd_insm;            /* insert mode 0 = type over ~0 = insert
                                  at cursor */
  u_char      pd_nulls,           /* end of line null count */
              pd_page,            /* lines per page */
              pd_tabsiz,          /* tabulate field size */
              pd_err,             /* most recent I/O error status */
              pd_rsvd[2];         /* reserved */
  u_int32     pd_col,             /* current column number */
              pd_time;            /* time out value for unblocked reads */
  Dev_list    pd_deventry;        /* Device table address (copy) */
} scf_path_opts;
```

The following table lists the SCF path descriptor options.

Table 3-7. SCF Path Descriptor Options

Name	Description
`pd_optsize`	**Path Descriptor Options Size** This is the total size of the SCF path options section.
`pd_inmap`	**Control Character Mapping Table** This is the input control character mapping table. It maps input control characters to the input line editing functions or user-defined control strings (break sequences). The control mapping table is described in detail following this section.
`pd_eorch`	**End of Record Character** This is the end-of-record character—the terminating character entered on each line for the `_os_readln` system call. Output lines from `_os_writeln` calls are terminated when this character is sent. Normally, the end-of-record character is set to "`\x0d`". NOTE: If the end-of-record character is set to `0`, `_os_readln` calls never terminate.
`pd_eofch`	**End of File Character** This is the end-of-file character. SCF returns an end-of-file error for `_os_read` and `_os_readln` system calls when this is the first (and only) character input. It can be disabled by setting this value to `0`.
`pd_tabch`	**Tab Character** This is the tabulate character. In `_os_writeln` calls, SCF expands this character to spaces to make tab stops at column intervals specified by the `pd_tabsiz` field. NOTE: SCF does not know the effect of control characters on particular terminals. Therefore, it may expand tabs incorrectly if they are used.
`pd_bellch`	**Bell Character** This is the bell sound character. In `_os_readln` calls, SCF echoes this character to the terminal once for every character input after the input buffer has filled. It is only useful for terminals with sound capability. It can be disabled by setting this value to `0`.
`pd_bspch`	**Backspace Character** This is the backspace "output" echo character. This is the backspace character SCF echoes when it is performing an editing function requiring a backspace, such as move cursor left.
`pd_case`	**Case Mode** This field indicates the casing mode SCF should use for input and output characters. When this field is non-zero, SCF converts all characters in the range `a..z` to `A..Z`.

Table 3-7. SCF Path Descriptor Options (Continued)

Name	Description
`pd_backsp`	**Destructive Backspace Flag** This field indicates whether backspacing (move cursor left) is destructive or non-destructive. If it is `0`, a move cursor left input control character causes SCF to echo a `pd_bspch` character. If it is non-zero, SCF echoes `pd_bspch`, space, `pd_bspch`.
`pd_delete`	**Delete Line Function** This field specifies how SCF implements the delete line editing function. If it is `0`, SCF deletes the line by backspace-erasing the line. If it is non-zero, SCF deletes the line by echoing a carriage return/line feed.
`pd_echo`	**Echo Flag** This field determines whether or not SCF echoes input characters. If it is non-zero, SCF echoes input characters. If it is `0`, SCF does not echo input characters.
`pd_alf`	**Line Feed Flag** This is the automatic line feed flag. If it is `0`, a line feed character is echoed after every end-of-record character output by the `_os_writeln` service request.
`pd_pause`	**Page Pause Flag** This field is the end of page pause indicator. If it is non-zero, an auto page pause occurs upon reaching a full screen of output. Refer to `pd_page` for information on setting the page length.
`pd_insm`	**_os_readln Input Mode** This field determines the insert mode for `_os_readln` calls. If it is `0`, input is in type-over mode. If it is non-zero, input characters are inserted at the cursor position and all characters to the right of the cursor are shifted to the right.
`pd_nulls`	**Padding Characters** This field specifies the number of null padding characters (always `'\0'`) to be echoed after a carriage return/line feed sequence.
`pd_page`	**Lines per Page** This field specifies the number of lines per page or screen.
`pd_tabsiz`	**Tab Size** This field specifies the tab size.
`pd_err`	**I/O Error Status** This field contains the most recent I/O error status.
`pd_col`	**Current Column Position** This field contains the current column position of the cursor.

Table 3-7. SCF Path Descriptor Options (Continued)

Name	Description
`pd_time`	**Time Out For _os_read, _os_readln** This field specifies the time out value (in ticks) for unblocked `_os_read` and `_os_readln` calls. When this field is set to 1 tick, these calls return the characters currently available in the user's input buffer.
`pd_deventry`	**Device Table Entry Address** This field contains the address of the device table entry for the path.

SCF Control Character Mapping Table

This table maps input control characters to the input line editing functions or user-defined control strings. Each entry in the field directly corresponds to the control character ASCII value in ascending order. The following control characters are mapped in this table: `0x01 - 0x1F` and `0x7F`.

Each entry in the table has the following format:

```
typedef struct inmap_entry {
  u_int16     type,               /* character mapping type */
              func_code;          /* SCF editing function code */
  u_int32     size;               /* size of associated string */
  void        *string;            /* pointer to associated string */
} inmap_entry;
```

Table 3-8. SCF Control Character Mapping Table

Name	Description	
type	**Mapping Type** The control character mapping type can be one of three values:	
	`IGNORE`	This control character is removed from the data stream.
	`PASSTHRU`	This control character is passed on without editing.
	`EDFUNCTION`	This control character is removed from the data stream and an editing function is performed in its place.

Table 3-8. SCF Control Character Mapping Table (Continued)

func_code	**Editing Function Code** If the type field is defined as `EDFUNCTION` and size is zero, `func_code` must be defined. This field can be any of the following function codes:

`MOVLEFT`	0x00	move cursor to the left (formerly `pd_bsp`)
`MOVRIGHT`	0x01	move cursor to the right
`MOVBEG`	0x02	move cursor to the beginning of the line
`MOVEND`	0x03	move cursor to the end of the line
`REPRINT`	0x04	reprint the current line to cursor position
`TRUNCATE`	0x05	truncate the line at the cursor position
`DELCHRL`	0x06	delete character to the left
`DELCHRU`	0x07	delete character under the cursor
`DELWRDL`	0x08	delete word to the left
`DELWRDR`	0x09	delete word to the right
`DELINE`	0x0A	delete the entire line
`UNDEF1`	0x0B	undefined (reserved)
`MODETOGL`	0x0C	insert mode toggle (type over vs. insert)
`UNDEF2`	0x0D	undefined (reserved)
`ENDOREC`	0x0E	end of record (read-only)
`ENDOFILE`	0x0F	end of file

size	**Size of Editing Function String** This field specifies the size of the editing function string to echo to the terminal. If this field is specified as `0`, an editing function built into SCF is executed to perform the editing function. If this field is non-zero, the string pointed to by `string` is echoed to the terminal.
string	**Editing Function String Pointer** This field points to the character string to be echoed to the terminal. This is only used if size is non-zero.

Default Mapping Table

The following control character mappings are defined in `scf.des`. They are used whenever a new device descriptor is created.

Table 3-9. SCF Default Mapping

Identifier	Function Type	Function Code	Size	String
`0x01 <Ctrl> A`	`EDFUNCTION`	`MOVEND`	`0`	`NULL`
`0x02 <Ctrl> B`	`EDFUNCTION`	`MOVLEFT`	`0`	`NULL`
`0x03 <Ctrl> C`	`IGNORE`	`0`	`0`	`NULL`
`0x04 <Ctrl> D`	`EDFUNCTION`	`DELCHRU`	`0`	`NULL`
`0x05 <Ctrl> E`	`IGNORE`	`0`	`0`	`NULL`
`0x06 <Ctrl> F`	`EDFUNCTION`	`MOVRIGHT`	`0`	`NULL`
`0x07 <Ctrl> G`	`PASSTHRU`	`0`	`0`	`NULL`

Table 3-9. SCF Default Mapping (Continued)

Identifier	Function Type	Function Code	Size	String
0x08 <Ctrl> H	EDFUNCTION	DELCHRL	0	NULL
0x09 <Ctrl> I	EDFUNCTION	MODETOGL	0	NULL
0x0A <Ctrl> J	PASSTHRU	0	0	NULL
0x0B <Ctrl> K	EDFUNCTION	TRUNCATE	0	NULL
0x0C <Ctrl> L	EDFUNCTION	DELWRDL	0	NULL
0x0D <Ctrl> M	EDFUNCTION	ENDOREC	0	NULL
0x0E <Ctrl> N	PASSTHRU	0	0	NULL
0x0F <Ctrl> O	PASSTHRU	0	0	NULL
0x10 <Ctrl> P	EDFUNCTION	REPRINT	0	NULL
0x11 <Ctrl> Q	IGNORE	0	0	NULL
0x12 <Ctrl> R	EDFUNCTION	DELWRDR	0	NULL
0x13 <Ctrl> S	IGNORE	0	0	NULL
0x14 <Ctrl> T	PASSTHRU	0	0	NULL
0x15 <Ctrl> U	PASSTHRU	0	0	NULL
0x16 <Ctrl> V	PASSTHRU	0	0	NULL
0x17 <Ctrl> W	IGNORE	0	0	NULL
0x18 <Ctrl> X	EDFUNCTION	DELINE	0	NULL
0x19 <Ctrl> Y	PASSTHRU	0	0	NULL
0x1A <Ctrl> Z	EDFUNCTION	MOVBEG	0	NULL
0x1B <Esc>	EDFUNCTION	ENDOFILE	0	NULL
0x1C	PASSTHRU	0	0	NULL
0x1D	PASSTHRU	0	0	NULL
0x1E	PASSTHRU	0	0	NULL
0x1F	PASSTHRU	0	0	NULL
0x7F <Delete>	EDFUNCTION	DELCHRU	0	NULL

Building SCF Device Descriptors

Making OS-9 device descriptors involves two steps:

1. Modifying the appropriate C macro definitions within the `SCF/<Driver>/config.des` for a specific device descriptor.
2. Making the descriptor using the associated makefile.

The `config.des` file is organized so the macro definitions for a particular descriptor are grouped together. For example, the following example of `config.des` contains the macros that must be defined (i.e. macros that do not have pre-defined defaults for the SCF `term` descriptor). They are grouped together within a C macro conditional:

```
/* Device descriptor common macros */

#define LUN         1

#define DRVR_NAME   “sc7110”


/* scf macros */

#define IRQLEVEL    0
```

```
#define PRIORITY    5
#define INPUT_TYPE IRQDRIVEN
#define OUTPUT_TYPE IRQDRIVEN
/*********************************************************************
 * End of Sc7110 Device Default Definitions      *
*********************************************************************/
/*********************************************************************
* Term_t1 Sc7110 Descriptor Override Definitions (descriptor for Com1)*
*********************************************************************/
#if defined (TERM_T1)
/* Module header */
#define MH_NAME "term"

/* Device descriptor common macros */
#define PORTADDR    0x80000480

/* scf macros */
#define VECTOR 0x4c

#endif /* TERM_T1 */
/******************************************************************
```

Usually a few fields for every descriptor type must be defined in order to make the descriptor (for example, port address, vector, IRQ level). However, most of the fields of the descriptor structures have pre-defined values. Consequently, you do not need to redefine them. These values seldom change from descriptor to descriptor. If a change in the operational characteristics of a device is desired, redefine the standard macro for the target field in `config.des` and make the descriptor.

Once you have edited the `config.des` file, edit the appropriate makefile by adding the appropriate dependencies and command lines to the makefile. When you have added these lines, make the descriptor. The following is a typical cross hosted make command sequence.

```
$ chd SCF/SC68901/DESC
$ os9make
```

The makefile invokes the `editmod` utility to create the descriptor. `EditMod` generates device descriptors from description files.

The information in this chapter describes how `editmod` can be used to create device descriptors. `editmod` can also list or edit the contents of a device descriptor. For more information about this utility, refer to the *Utilities Reference* manual.

SCF Device Descriptor Macros

Table 3-10 and Table 3-11 contain the SCF macro definitions used for creating SCF device descriptors.

Table 3-10. RBF, SCF, and SBF Common Descriptors

Name	Description and Example
`PORTADDR`	**Controller Address** Address of the device on the bus. This is the lowest address the device has mapped. It is hardware-dependent. `#define PORTADDR    0xfffe4000`
`VECTOR`	**Interrupt Vector** This is the vector passed to the processor at interrupt time. It is hardware/software-dependent. Some devices can be programmed to produce different vectors. `#define VECTOR      80`
`IRQLEVEL`	**Interrupt Level For the Device** The number of supported interrupt levels is dependent on the processor being used (e.g. 1-7 on 680x0 type CPUs). When a device interrupts the processor, the level of the interrupt is used to mask out lower priority devices. `#define IRQLEVEL    4`
`PRIORITY`	**Interrupt Polling Priority** This value is software dependent. A non-zero priority determines the position of the device within the vector. Lower values are polled first. A priority of 1 indicates the device desires exclusive use of the vector; a priority of 0 indicates the device wants to be first on the polling list. OS-9 does not allow a device to claim exclusive use of a vector if another device has already been installed on the vector. Also, it does not allow another device to use the vector once the vector has been claimed for exclusive use. `#define PRIORITY    10`
`LUN`	**Logical Unit Number of the Device** More than one device may have the same port address. The logical unit number distinguishes the devices having the same port address. `#define LUN    2     /* drive number */`

The following macros are specific to SCF:

Table 3-11. SCF Macros

Name	Description

`MODE`	**Device Access Capabilities**
	This reflects the operational mode or capabilities of the device. Most SCF type devices use the default value. However, in some cases you should change `MODE` to include the `S_ISHARE` bit signaling the device is non-sharable. For example, an SCF serial printer port should be non-sharable.
	`#define MODE S_ISIZE|S_IREAD|S_IWRITE`
	`/* default device mode capabilities */`

`MAXBUFF`	**Maximum Data for the Input Buffer**
	This defines the *high water mark* of the input buffer. When the input buffer reaches the defined level, the sender is sent an X-OFF character to temporarily halt transmission.
	`#define MAXBUFF OUTSIZE-LOWCOUNT`
	`/* default maxbuff size */`

`INPUT_TYPE`	**Input Type Flag**
	This specifies whether input on the device is interrupt driven or polled. If the device is operated in polled mode, SCF calls the driver's read routine for every character. The two values defined for this field are:
	`#define IRQDRIVEN   0`
	`#define POLLED      1`
	They are defined in `scf.h`. The default for this macro is interrupt driven.
	`#define INPUT_TYPE IRQDRIVEN`

`OUTPUT_TYPE`	**Output Type Flag**
	This specifies whether output on the device is interrupt driven or polled. If the device is operated in polled mode, SCF calls the driver's write routine to transmit every character.
	`#define IRQDRIVEN   0`
	`#define POLLED      1`
	They are defined in `scf.h`. The default for this macro is interrupt driven.
	`#define OUTPUT_TYPE IRQDRIVEN`

`SCFBUFSIZE`	**Path Descriptor Buffer Size** This specifies the size of the path descriptor buffer for all paths opened to the device. The default is 256 bytes. `#define SCFBUFSIZE  256`
`INSIZE`	**Logical Unit Input Buffer Size** This specifies the size of the input buffer for the logical unit. The default is 256 bytes. `#define INSIZE  256`

Table 3-11. SCF Macros (Continued)

Name	Description
`OUTSIZE`	**Logical Unit Output Buffer Size** This specifies the size of the output buffer for the logical unit. The default is 256 bytes. `#define OUTSIZE  256`
`KYBDINTR`	**Keyboard Interrupt Function** This specifies the control key to use for the keyboard interrupt function. The default value is <control>C. `#define KYBDINTR  Ctrl_C`
`KYBDQUIT`	**Keyboard Quit Function** This specifies the control key to use for the keyboard quit function. The default value is <control>E. `#define KYBDQUIT  Ctrl_E`
`KYBDPAUSE`	**Keyboard Pause Function** This specifies the control key to use for the keyboard pause function. The default value is <control>W. `#define KYBDPAUSE  Ctrl_W`
`XON`	**XON Function** This specifies the control key to use for the X-ON protocol function. The default value is <control>Q. `#define XON  Ctrl_Q`
`XOFF`	**XOFF Function** This specifies the control key to use for the X-OFF protocol function. The default value is <control>S. `#define XOFF  Ctrl_S`

`UPC_LOCK`	**Character Case Function**
	This controls the casing of characters. A non-zero value converts input and output characters in the `a...z` to characters in the `A...Z` range. The default value is upper and lower casing.
	`#define UPC_LOCK  PLOFF`
	`/* default to upper and lower case */`

`BSB`	**Backspace Character Interpretation**
	This controls how SCF interprets a backspace character: as a destructive or non-destructive backspace. If this value is zero, SCF echoes a backspace character. If this value is non-zero, SCF echoes a backspace, space, backspace character sequence. The default is a destructive backspace.
	`#define BSB  PLON`
	`/* default to destructive backspace */`

`LINEDEL`	**Delete Line Function**
	This controls how SCF performs a delete line function. A zero value causes SCF to delete a line by backspacing over it. A non-zero value causes a carriage return/line feed sequence to be echoed to delete the line. The default is a destructive line delete.
	`#define LINEDEL  PLON`
	`/* default destructive delete line */`

Table 3-11. SCF Macros (Continued)

Name	Description

`AUTOECHO`	**Input Echo Function**
	This controls whether or not input characters are echoed as they are received. A non-zero value causes input to be echoed. A zero value flags no echoing. The default is echo on.
	`#define AUTOECHO  PLON`
	`/* default to echo on */`

`AUTOLF`	**Automatic Line Feed Function**
	This specifies whether or not carriage returns are to be automatically followed by line feed characters. The default is auto line feed on.
	`#define AUTOLF  PLON`
	`/* default to auto line feed on */`

`EOLNULLS`	**Nulls After End-Of-Line**
	This specifies the number of null ($00) padding bytes to be transmitted after a carriage return/line feed sequence. The default value is `0`.
	`#define EOLNULLS  0`
	`/* default to no end-of-line nulls */`

`PAGEPAUSE`	**Page Pause Function**
	This specifies whether or not the automatic page pause facility of SCF is active. A non-zero value causes an auto page pause upon reaching a full screen of output. The default is page pause on.
	`#define PAGEPAUSE  PLON`
	`/* default to page pause on */`

`PAGESIZE`	**Lines Per Page**
	This specifies the number of lines per screen (or page). The default value is twenty-four lines per page.
	`#define PAGESIZE  24`
	`/* default to 24 line/page */`

`TABSIZE`	**Spaces Per Tab**
	This specifies the number of spaces per tab. The default value is `4`.
	`#define TABSIZE  4`
	`/* default to 4 spaces/tab */`

`INSERTMODE`	**Input Mode Specification**
	A non-zero value causes input to operate in insert mode. This is, input characters are inserted in the current input line. A zero value causes input to operate in the type-over mode. The default is type-over mode. By using the associated control key, SCF allows you to enter insert mode.
	`#define INSERTMODE  PLOFF`
	`/* default to insert mode off */`

Table 3-11. SCF Macros (Continued)

Name	Description

`BAUDRATE`	**Baud Rate**
	This specifies the baud rate of the device. The default is 9600. The various standard baud rate macros are defined in the `scf.h` header file.
	`#define BAUDRATE  BAUD9600`
	`/* default to 9600 baud */`

`LUPARITY`	**Parity of Logical Unit**
	This specifies the parity node of the device. The default is no parity. The various standard parity macros are defined in the `scf.h` header file.
	`#define LUPARITY  NOPARITY`
	`/* default to no parity */`

`STOPBITS`	**Stop Bits**
	This specifies the number of stop bits to be used for transmission. The default number of stop bits is one.
	`#define STOPBITS  ONESTOP`
	`/* default to one stop bit */`

`WORDSIZE`	**Bits Per Character**
	This specifies the number of bits per character to be used for transmission. The default word size is eight bits per byte.
	`#define WORDSIZE  WORDSIZE8`
	`/* default to 8 bits/byte */`

`RTSSTATE`	**Request to Send Flag**
	This determines the state of the `request to send` line for hardware handshaking. The default state is disabled.
	`#define RTSSTATE  RTSDISABLED`
	`/* default to RTS disabled */`

SCF Control Character Mapping

You can also change the input control character mapping. This involves redefining the control character macro in `SCF/<driver>/config.des` as described previously. The default input control character mapping macros are located in the `scf.des` file.

Device Specific Non-Standard Definitions

Some SCF drivers require device-specific information to be defined within the logical unit static storage structure. The structure definition is needed for driver and descriptor creation in differing forms (C-source include file for the driver and `editmod` source for the descriptor).

Write the `editmod` form of the device-specifics record and adapt the driver's makefile to use `editmod` to generate the C-source include file from the `editmod` source.

The `sc85x30` example driver does this in `sc85x30.des`:

```
#define DEV_SPECIFIC
```

```
data struct device_specific_des {
/* Device specific static variables */
    u_int32    (“u_char *%s”) v_irqport, “device hardware irq register
pointer”;
    u_int32    (“u_char *%s”) v_port, “device hardware register pointer”;
    u_char     v_autovect, “autovector flag; 0=chip vector 1=autovector”;
}, “sc85x30 device specific storage”;
string drvr_name = “sc85x30”;
```

Next, the makefile uses `editmod` to create the `sc85x30.edm` file (example follows), that is included by the primary driver include file, `sc85x30.h`.

```
BUILD = $(EDITMOD) -v$(SDIR)
-mDEV_SPECIFICS=device_specific_des


$(SDIR)/sc85x30.edm: $(SDIR)/sc85x30.des $(MAKERS)
     $(BUILD) sc85x30.des -o$@
```

In addition to the standard fields described in `systype.h`, you can add specific definitions for particular driver/descriptor combinations.

4 The PC File Manager (PCF)

This chapter describes the PC File Manager.

Overview

PCF is a reentrant subroutine package that handles I/O service requests for random-access PC-DOS/MS-DOS disk devices. PCF can handle any number of such devices simultaneously, and is responsible for maintaining the defined logical file structure on the PC-DOS/MS-DOS disk drive.

PCF supports FAT12, FAT16, and FAT32 file formats. Long file names (called VFAT), introduced with the advent of Windows 95, are fully supported. PCF will automatically choose the correct FAT algorithms for the device that is accessed. When creating a FAT file system, FAT12 should be used for devices under 32MB in size and FAT16 should be used for devices under 2GB in size. The requirements of FAT32 increase overhead and will slow down disk access.

Getting Top Performance from PCF

While PCF has been designed to achieve as much performance as possible, there are a few steps that applications can take to insure that PCF achieves maximum throughput:

- **Initialize all PCF devices.** For performance reasons, PCF reads the entire disk's FAT into memory at open time. If the device is not initialized, the reading of the FAT can occur as many times as a file is opened on the device. To insure the FAT is read once per device, initialize the device before using it. This will decrease file open times, especially on slower devices such as floppy drives or large devices such as hard drives larger than 512MB.
- **Pre-extend files when writing.** One way of increasing write performance is to pre-extend the file's size by using the `_os_ss_size()` function. Note that the `FAM_SIZE` bit in `_os_create()` is not recognized by PCF.

Differences from RBF

While PCF maintains very good compatibility with existing OS-9 disk utilities, there are some subtle differences that should be noted.

- **There is no record locking.** Unlike RBF, PCF does not employ record locking on a file. However, to prevent conflicts between processes, device locking is used at each entry point of the PCF file manager. Since PCF lacks record locking, only one path to any given file on a PCF device should be open at any time.
- `FAM_SIZE` **is not recognized.** Under RBF, a typical way to pre-extend the size of a file at create time is to pass `FAM_SIZE` as a parameter to the `_os_create()` function; however, the PCF file manager does not recognize this parameter. If file pre-sizing is desired, use the `_os_ss_size()` function.
- **There is a different directory structure format.** If the application reads the directory raw and parses the entries, it must be written to accommodate the PCF directory format. It is highly recommended that an application which needs to read directory structure information use the portable functions: `opendir()`, `readdir()`, and `closedir()`. These functions are compatible with all OS-9 file storage managers.

5 Module File Manager (modman)

This chapter explains how to use the modman file manager to process I/O service requests to a simulated disk-based device and the configurable parameters. It includes the following topics:

- Overview
- Device Descriptor Modules

Overview

The modman file manager is a re-entrant subroutine package for I/O service requests to read-only random-access devices. modman simulates the directory and file structure of an RBF device. It is generally used to simulate a disk on a diskless system.

The directory heirarchy is derived from the module directory heirarchy. The directory contents are derived from the module directory contents. The file contents are derived from module contents. For example, creating a module directory will result in a new simulated directory. Loading or creating a module will result in a new simulated file.

Modman supports the automatic creation of a directory heirarchy. When modman is first initialized it reads a special configuration module that specify the desired directory structure. It then uses OS-9 module directories and small data modules to create a heirarchy. The small data modules serve as symbolic links to existing modules. The heirarchy contents and format module are created using the mar utility (documented in the *Utilities Reference* manual).

The format of the directory heirarchy specification module is a series of string pairs. The first string of the pair is the desired pathlist to the module. The second string of the pair is the module that should appear there. The strings are separated by white-space characters or NUL ('\0'). The first name may either begin with /<device name> or just the directory in which the module should appear. The second string can be either relative to the root module directory or absolute. If any of the following lines appeared in a directory structure configuration module specified in a modman device descritptor called /mm a module directory called INITMOD would be create at the root with a module called initlink that would refer to the init module in the root module directory:

```
/mm/INITMOD/initlink init

INITMOD/initlink /init

/mm/INITMOD/initlink /init

INITMOD/initlink init
```

The directory structure configuration module is terminated by two white-space characters in a row. Refer to the documentation for mar for information on automatically generating this module.

modman I/O System Calls

modman handles the following I/O system calls:

Table 5-1. Included I/O System Calls

_os_attach()	_os_chdir()	_os_close()	_os_create()
_os_delete()	_os_detach()	_os_dup()	_os_getstat()
_os_makdir()	_os_open()	_os_read()	_os_read()
_os_readln()	_os_seek()	_os_setstat()	_os_write()
_os_writeln()			

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

_os_attach()

If the modman device is already attached, modman simply increments the attach count. Otherwise, modman performs the following functions for each directory structure module specified in the device descriptor (while there are white-space, separated pathlist pairs):

- Create the directories necessary to reach the first item of the pair.
- Create a symbolic link module with the pathlist of the first item of the pair that refers to module named as the second item of the pair.

_os_chdir()

modman returns unknown service code (`EOS_UNKSVC`). You cannot change your current data nor execution directory to a modman device.

_os_close()

modman performs the following functions:

- Check the use count in the path descriptor. If the use count is non-zero, SUCCESS is returned.
- Free any memory allocated to maintain the path.
- Unlink from the associated module, if any.

_os_create()

modman returns unknown service code (EOS_UNKSVC). You cannot create modules with modman. Use other means of adding modules to the system (e.g. `_os_datmod()` or `_os_load()`) to create “files” on the modman device.

_os_delete()

modman returns unknown service code (EOS_UNKSVC). You cannot delete modules with modman. Use other means of removing modules from the system (e.g. `_os_unload()` or `_os_unlink()`) to remove “files” from the modman device.

_os_detach()

modman frees all memory used to maintain the device. The directory structure created by `_os_attach()` is left intact.

_os_dup()

modman returns SUCCESS, no functions beyond those performed by IOMan are necessary.

_os_getstat()

modman supports the following getstats:

Table 5-2. _os_getstat() Functions

Function	Description
`_os_gs_popt()`	Get the current path options
`_os_gs_dopt()`	Get the default path options
`_os_gs_fdinf()`	Get a copy of a specified file descriptor.
`_os_gs_ready()`	Test for data ready.
`_os_gs_fd()`	Get file descriptor.
`_os_gs_eof()`	Test for EOF condition.
`_os_gs_size()`	Determine file's size.
`_os_gs_pos()`	Determine current file position.
`_os_gs_fdaddr()`	Determine the file descriptor address.

All other `_os_getstat()` calls return EOS_UNKSVC.

_os_makdir()

modman returns unknown service code (EOS_UNKSVC). You cannot create module directories with modman. Use other means of creating module directories (such as the `makmdir` utility or `_os_makmdir()`) to create "directories" on the modman device.

_os_open()

modman performs the following functions:

- Ensure that no modes other than read or directory are used, all other combinations return EOS_BMODE (bad mode).
- Ensure that a full pathlist was specified because _os_chdir() is not supported.
- If directory mode is specified:
    - Ensure the directory name specified is not a module; if so, return EOS_FNA (file not accessible).
    - Ensure the directory name specified exists; if not, return EOS_PNNF.
    - Create a snapshot of the directory's contents in the format of an RBF directory.
    - Set the path descriptor up so that reads come from this simulated RBF directory.

Read mode performs the following functions:

- Ensure the file name specified is not a module directory; if so, return EOS_FNA (file not accessible).
- Follow symbolic links.
- Link to the specified module.

- Determine the beginning of the data portion of the module using thes rules:
  - If the module was created by the mkdatmod utility, take advantage of the information it writes in the module to determine the size of the data section.
  - If the module header's `m_share` value is non-zero, use it for module size.
  - If the module contains only ASCII data, seek to the end of the data portion and then back up, consuming any number of zero bytes.
  - Otherwise, consider everything from m_exec to the end of the module, minus the space for the CRC, the data portion of the file.

_os_read()

modman performs the following functions:

- Ensure that the read mode was specified when the path was opened.
- Determine if there is data left to read in the file or directory. If there is none, an end-of-file error (`EOS_EOF`) is returned.
- Copy the data from the module to the user's buffer starting at the current file position and going until the end of the file is encountered or the count specified by the user is exhausted.
- Return the number of bytes copied.

_os_readln()

modman performs the following functions:

- Ensure that the read mode was specified when the path was opened.
- Determine if there is data left to read in the file or directory. If there is none, an end-of-file error (`EOS_EOF`) is returned.
- Copy the data from the module to the user's buffer starting at the current file position and going until the end of the file is encountered or the count specified by the user is exhausted or a carriage return ('\x0d') is copied.
- Return the number of bytes copied.

_os_seek()

modman sets the current position in the path descriptor to the specified position.

_os_setstat()

modman supports the `_os_ss_symlink()` setstat. This setstat creates a symbolic link module to another module. All other `_os_setstat()` calls return EOS_UNKSVC.

_os_write()

modman returns unknown service code (EOS_UNKSVC). You cannot write to modules with modman. modman is designed to aid only in the emulation of a read-only file system. To change the contents of a file write into it via other means or unlink and old module and replace it with a new one.

_os_writeln()

modman returns unknown service code (EOS_UNKSVC). You cannot write to modules with modman. modman is designed to aid only in the emulation of a read-only file system. To change the contents of a file write into it via other means or unlink and old module and replace it with a new one.

Device Descriptor Modules

This section describes the path options and logical unit static storage structures in a modman device descriptor module.

The table below describes the path options section of the device descriptor.

Table 5-3. Path Options Structure

Name	Type	Description
`pd_reserved`	`u_int32 [4]`	**Reserved** Reserved for expansion of the path options.

This table describes the logical unit static storage section of the device descriptor table.

Table 5-4. Logical Unit Static Storage Structure

Name	Type	Description
`v_treemod_name`	`char *`	**Tree Module Name** This field is set to the name(s) of module directory structure modules. There may be any number of these directory structure modules. Their names are separated by white-space.
`v_linkcnt`	`u_int32`	**Attach Count** This field is used to keep track of the number of times that this locical unit has been attached to the system. The first and last detach are the only substantial actions. The first attach causes modman to parse the directory structure modules specified in `v_treemod_name`.
`v_reserved`	`u_int32 [4]`	**Reserved** Reserved for expansion of the logical unit static storage.