RT–11 Device Handlers Manual

Order Number AA–PE7VA–TC

August 1991

This manual describes the structure of device handlers, how to write your own device handler, and provides specific programming information about distributed RT–11 device handlers.

Revision/Update Information: This is a new manual for programmers; it is a complete revision of the information previously located in Chapters 7 and 10 of the *RT–11 Software Support Manual*.

Operating System: RT–11 Version 5.6

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, August 1991

The Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

S559

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

Chapter 2 Programming for Specific Devices

Figures

Tables

Preface

Document Structure

This manual is divided into the following two chapters:

- Chapter 1, Device Handlers, describes the recommended structure of device handlers and provides detailed information on how to write a device handler.
- Chapter 2, Programming for Specific Devices, alphabetically presents programming information for specific distributed device handlers.

Audience

This manual is written for those users of the RT–11 operating system who want to understand distributed device handlers and write their own device handlers.

Conventions

The following conventions are used in this guide.

Convention	Meaning

Black print	In examples, black print indicates output lines or prompting characters that the system displays. For example:
	`.BACKUP/INITIALIZE DL0:F*.FOR DU1:WRK`
	`Mount output volume in DU1:; continue? Y`

Red print	In examples, red print indicates user input.
Braces ({ })	In command syntax examples, braces enclose options that are mutually exclusive. You can choose only one option from the group of options that appear in braces.
Brackets ([ ])	Square brackets in a format line represent optional parameters, qualifiers, or values, unless specified otherwise.
Lowercase characters	In command syntax examples, lowercase characters represent elements of a command for which you supply a value. For example: `DELETE filespec`

Convention	Meaning
UPPERCASE characters	In command syntax examples, uppercase characters represent elements of a command that should be entered exactly as given.
RET	RET in examples represents the RETURN key. Unless the manual indicates otherwise, terminate all commands or command strings by pressing RET.
CTRL/*x*	CTRL/*x* indicates a control key sequence. While pressing the key labeled Ctrl, press another key. For example: CTRL/C

Associated Documents

Basic Books

- *Introduction to RT–11*
- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*
- *RT–11 Commands Manual*
- *RT–11 Quick Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

- *RT–11 Automatic Installation Guide*
- *RT–11 Installation Guide*
- *RT–11 System Generation Guide*

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 System Internals Manual*

- *RT–11 Volume and File Formats Manual*
- *DBG–11 Symbolic Debugger User's Guide*

Chapter 1
Device Handlers

The term *device handler* can mean three things, depending on the context in which it is used. A device handler can be:

- The source program

  This is a .MAC file that is distributed with RT–11 or you write.

- The file image

  This is a .SYS file that is distributed with RT–11 or the assembled and linked source program you write.

- The memory image

  This is the part of the file image that resides in memory; the memory resident portion of the device handler. Not all of the file image is normally loaded in memory. The first block (block 0) of the file image, for example, is temporarily loaded when the monitor requires information that is stored in handler block 0. The memory resident portion of the device handler begins at block 1 of the file image. Therefore, block 1 of the file image is the beginning of the memory image.

To write a device handler, you first need to know what points to consider in the planning stage. These points are listed and cross-referenced in the first sections of this chapter. The points that have not been treated elsewhere in this manual are then described in detail. Device handler structure and a skeleton outline of a typical handler are covered here. After this, details are given on the optional features available to handlers and their implementation. Optional features include internal queuing, SET options, device I/O timeout support, special functions, error logging, and special services available in mapped systems.

To write a bootstrap for a system device, you first need to know the differences between a standard handler and a system device handler. These differences are discussed in several sections before the final sections of the chapter, where you will find explained the assembly, installation, testing, and debugging procedures for the new handler.

Be sure to also read Chapter 5 of the *RT–11 System Internals Manual*, as that chapter can help you decide whether you need to write an in-line interrupt service routine or a device handler.

1.1 How to Plan a Device Handler

The most important part of writing a device handler is taking the time to plan the whole process carefully. Follow these guidelines:

- Get to know your device
- Study the structure of a standard device handler
- Study the skeleton device handler
- Think about using the special features
- Study the sample handlers
- Prepare a flowchart of the device handler
- Write the code
- Install, test, and debug the handler

1.1.1 Get to Know Your Device

Learning about the characteristics of your device and the bus interface is crucial to writing a handler that works correctly. Review the appropriate material in Chapter 5 of the *RT–11 System Internals Manual* so that you can answer all the pertinent questions about your device before you attempt to write a handler for it.

1.1.2 Study the Structure of a Standard Device Handler

Section 1.2 describes the structure of a standard device handler. Read this section carefully; your handler should conform to this structure.

1.1.3 Study the Skeleton Device Handler

Section 1.2.10 contains a skeleton outline of a standard device handler. You can use this outline as a starting point when you begin to write your own handler.

1.1.4 Think About Using the Special Features

Sections 1.4 through 1.10 describe the special features available to device handlers. Read these sections carefully to determine whether any features are applicable to your handler.

1.1.5 Study the Sample Handlers

Appendix A contains assembly listings of three RT–11 device handlers (DL, DX, and XL) with extensive explanatory comments. Study these listings until you feel comfortable with the organization of the handlers, and you understand how they implement some of the special features. Obtain listings of handlers for other devices that resemble yours; you may be able to use some of the code that is already written.

1.1.6 Prepare a Flowchart of the Device Handler

Preparing a flowchart for your handler can help you plan the contents of the various sections. Flowcharting can also help you spot loose ends and errors in your programming logic. Unfortunately, flowcharts are not much help in pointing out potential race conditions. (A race condition is a situation in which two or more asynchronous processes attempt to modify the same data structure at the same time; as a result, the data structure is corrupted and the integrity of the processes is compromised.) Therefore, when you design the handler, examine every step carefully and keep in mind what would happen if an interrupt occurred at each instruction. This kind of planning can help you avoid race conditions later.

1.1.7 Write the Code

If you have followed the recommended steps so far, writing the code for the device handler should be relatively simple. You must write Position-Independent Code (PIC) for the handler. Review the chapter on PIC code in the *PDP–11 MACRO–11 Language Reference Manual* if you are not already familiar with it. Copy as much code as possible from the commented device handlers in Appendix A or from other reliable sources. Start with a general outline that conforms to the structure presented in Section 1.2 and then add details to reflect the specifics of your particular device. When you have thoroughly checked the code for logic errors and it assembles properly, you are ready to test and debug it.

1.1.8 Install, Test, and Debug the Handler

Sections 1.14 and 1.15 show how to install a new device handler and how to begin testing and debugging it.

1.2 Structure of a Device Handler

For ease of explanation and understanding, the RT–11 handler source program is described as having the following six sections:

- Preamble

  The preamble is the information section of the source. Much of the information you put in the preamble as arguments to macro parameters and as system conditionals is associated with symbols that are used by macros in other handler sections. The macros you use in the preamble section create many of the handler's data structures and further define the handler.

- Header

  The header section is where you code the beginning of the memory resident portion of the handler.

- I/O initiation

  The I/O initiation section contains the first executable instructions; the code to get the handler ready to perform data transfers. The I/O initiation section is able to use data structures and symbols that were defined in the previous sections and defines further handler characteristics.

- Interrupt service

  The interrupt service section is the heart of the handler. It contains the code that processes interrupts as they are received from the device. It handles aborts and manages the handler queue.

- I/O completion

The I/O completion section contains code to inform the monitor of the success or failure of the interrupt processing and perform appropriate actions depending on success or failure.

- Handler termination

  The handler termination section is the tail of the handler. It contains code to build tables and handler service routines. Being at the end of the handler, it defines a symbol that is used to determine the size of the handler.

The complexity of the coding you must write is reduced because the RT–11 system macro library (SYSMAC.SML) provides device handler macros to generate much of the required code.

You should read and think about the following points before working through this section:

- Although the various macro parameters are listed and briefly described in this chapter, you should consult the *RT–11 System Macro Library Manual* for complete parameter argument descriptions. Refer to that manual as you read this chapter.

  Some of the macros that you use to write a device handler are interdependent. For example, the device status word is created from symbols that SYSMAC.SML equates based on arguments you supply to .DRDEF parameters. Those symbols are then used by .DRBEG to create the device status word and store it into the handler's block 0.

- RT–11 distributes a library of the system data structures (SYSTEM.MLB), described in the *RT–11 System Internals Manual*. In this section, the symbols that identify handler data structures and the elements in those structures are as defined in SYSTEM.MLB. If your device handler is assembled with SYSTEM.MLB, you can use those symbols and need not define them explicitly in your handler.

- As you work through the parts of this section, you should look at the skeletal device handler in Section 1.2.10. The skeletal handler illustrates the overall structure.

  For examples of specific handler structure, look at the sample device handlers in Appendix A.

  Also refer to Table 1–11, which illustrates the layout of a device handler .SYS file image.

1.2.1 Preamble Section

Begin the device handler source file with the preamble section. Include a .MCALL directive for the .DRDEF macro and any other macros you use that this chapter does not explicitly mention. Also in the preamble, you should define system conditionals that you will use later.

As shown in the skeletal handler, Figure 1–1, you include macros in the preamble section that build various data structures and define symbols. The following macros can be used in the preamble section:

- .DRDEF

  Provides the primary definition of the device handler and is the only mandatory device macro. Many of the values you supply as arguments to .DRDEF's parameters are equated during assembly to symbols that are then used by other handler macros.

- .DREST

  Provides information about the handler, which is stored in block 0 of the handler's file image.

- .DRPTR

  Points to handler service routines that can be run when the handler is loaded, unloaded, fetched, and released. Those routines do not reside in memory (keeping the memory resident portion of the handler smaller), but are read into and executed from the USR buffer.

- .DRSPF

  Defines which special functions the handler supports.

- .DRINS

  Points to any installation checking code and defines how the handler CSRs are to be displayed.

- .DRSET

  Defines the handler SET commands.

As you work through this section, look at Table 1–3 to see which offsets in block 0 are written by those macros.

1.2.1.1 .DRDEF Macro

Use the .DRDEF macro near the beginning of your device handler. In the following list of functions performed by .DRDEF, *dd* represents the device name you specify in the macro's *name* parameter. The .DRDEF macro's functions are to:

- Issue .MCALL directives for all handler-related macros
- Provide default values for the key system conditionals
- Invoke the .QELDF macro to define queue element offsets
- Define bit patterns for device characteristics
- Define *dd*DSIZ as the device size in blocks
- Define *dd*$COD as the device identification
- Set up the device status word from information in *dd*DSIZ and *dd*$COD
- Provide default values for the device CSR in *dd*$CSR and vector in *dd*$VEC
- Make the symbols *dd*$CSR and *dd*$VEC global

- Indicate whether the handler supports extended device units
- Indicate whether the handler supports DMA (direct memory access)
- Define the required number of permanent UNIBUS mapping registers if this handler supports DMA on UNIBUS processors
- Indicate whether the handler requires serialized I/O request satisfaction

The format of the .DRDEF macro call is as follows:

Macro Call: .DRDEF name,code,stat,size,csr,vec
[,UNIT64=str][,DMA=str][,PERMUMR=n][,SERIAL=str]

name	is the two-letter handler name, stored in H.HAN (offset 0 of handler block 0) by .DREST.
code	is the device identifier byte, stored in H.DSTS (offset 56 of handler block 0) by .DRBEG.
stat	is the device status bit pattern, stored in H.DSTS (offset 56 of handler block 0) by .DRBEG.
size	is the device size, stored in H.DSIZ (offset 54 of handler block 0) by .DRBEG.
csr	is the default value for the device's control and status register, stored in H.ICSR (offset 176 of block 0) by .DRBEG. To suppress storing a value in 176, specify *NO* as the argument to *csr*.
vec	is the default value for the device's interrupt vector, stored in H1.VEC (offset 0 of block 1) by .DRBEG.
UNIT64=str	is the number of device units to be supported by this handler, stored in H.UNIT (offset 76 of handler block 0) by .DRDEF.
DMA=str	indicates whether this handler supports direct memory access, stored in symbol DV2.DM of H1.FLG (offset 10 of block 1) by .DRBEG.
PERMUMR=n	indicates this handler should be assigned *n* permanent UNIBUS mapping registers, stored in H.64UM (offset 100 of handler block 0) by .DRDEF.
SERIAL=str	indicates handler requires serialized I/O completion, stored in symbol HF2.SR of H1.FG2, (offset 16 of block 1) by .DRBEG.

The .DRDEF macro also issues the .MCALL directive for the following macros:

.DRAST	.DRBEG	.DREST	.DRFIN
.DRBOT	.DREND	.DRINS	.DRSPF
.DRSET	.DRVTB	.FORK	.QELDF
.DRTAB	.DRUSE		

In addition, if you assemble your handler with the conditional TIM$IT set to 1, .DRDEF issues a .MCALL directive for the .TIMIO and .CTIMIO macros.

1.2.1.1.1 System Conditionals

RT–11 source files make extensive use of conditional assembly directives. Sections of source code are included or omitted at assembly time, based on the value of conditional symbols. For example, RT–11 uses the conditional ERL$G to indicate whether routines for error logging should be assembled.

If you use conditional symbols in your handler, they should conform to RT–11 standard usage by setting the conditional equal to 0 to indicate that the feature it represents is not to be included and by setting the conditional to 1 to include the feature. (Note that RT–11 uses only the values 0 and 1 to indicate absence or presence of a feature.) See the *PDP–11 MACRO–11 Language Reference Manual* for information on the conditional assembly directives (.IF EQ, .IF NE, and so on).

The .DRDEF macro sets to 0 the system generation conditionals TIM$IT (for device timeout), MMG$T (for extended memory support), and ERL$G (for error logging), if you do not define them in a prefix file at assembly time. In addition, if the symbols have values other than 0, .DRDEF sets them to 1.

1.2.1.1.2 Queue Element Offsets

The .DRDEF macro invokes .QELDF to define queue element offsets and define symbols for those offsets.

As shown in Table 1–1, the size of a queue element is determined by whether or not a monitor supports mapping.

Unmapped Monitors

For unmapped monitors, each queue element contains 16_8 bytes.

Mapped Monitors

Device handlers in a mapped environment require two more words of information to locate the actual user buffer in physical memory. The offsets, Q.PAR and Q.MEM, are values for PAR1 that, when combined with the user virtual buffer address (Q.BLKN), provide the physical address of the buffer.

Q.PAR and Q.MEM initially contain the same PAR1 value. The value in Q.PAR varies from Q.MEM only with UNIBUS Mapping Register (UMR) support; if the UMR handler UB is loaded, Q.PAR becomes a relocation constant to load UMRs. Q.MEM remains the PAR1 displacement bias for CPU memory management (MMU) address values. If there is no UMR support, Q.PAR and Q.MEM continue to contain the same PAR1 value. Therefore, you should use Q.MEM as the PAR1 displacement bias because it is not affected by the presence of UMR support.

Table 1–1: Queue Element Offsets

Name	Offset	Meaning
With All Monitors:		
Q.LINK	0	Link to next queue element

Table 1–1 (Cont.): Queue Element Offsets

Name	Offset	Meaning
Q.CSW	2	Pointer to channel status word
Q.BLKN	4	Physical block number
Q.FUNC	6	Special function code
Q.JNUM	7	Job number
Q.UNIT	7	Device unit number
Q.BUFF	^O10	User virtual buffer address
Q.WCNT	^O12	Word count
Q.COMP	^O14	Completion routine code
With Unmapped Monitors:		
Q.ELGH	^O16	Length of queue elements
	^O20–^O24	Reserved
With Mapped Monitors:		
Q.PAR	^O16	Is initially PAR1 value. See text above
Q.MEM	^O20	Is always PAR1 value. See text above
	^O22	Reserved
Q.ELGH	^O24	Length of queue elements

Since the handler usually deals with queue element offsets relative to offset Q.BLKN, the .QELDF macro also defines the following symbolic offsets:

Symbolic Offset	From Q.BLKN
Q$LINK	–4
Q$CSW	–2
Q$BLKN	0
Q$FUNC	2
Q$JNUM	3
Q$UNIT	3
Q$BUFF	4
Q$WCNT	6
Q$COMP	^O10

Symbolic Offset	From Q.BLKN
Q$PAR	^O12
Q$MEM	^O14

1.2.1.1.3 Symbol Definitions

Use direct assignment statements to define symbols that you will use later in the handler. Typically, the definitions include the device registers and other useful internal symbols. Some examples from the DY handler for mapped monitors follow:

```
;  FIXED OFFSETS EQUATES (.FIXDF)

        $PNPTR  =:      000404  ;RMON OFFSET OF PNAME TABLE
        P1$EXT  =:      000432  ;RMON OFFSET OF $P1EXT ADDRESS
        $H2UB   =:      000460  ;RMON OFFSET OF UB ENTRY VECTOR PTR
        MMG$T = 1

;  EXTENDED MEMORY SUBROUTINE OFFSETS FROM $P1EXT  (.P1XDF)

        $MPMEM  =:      -22.    ;OFFSET TO MAP KT-11 VIRTUAL TO PHYSICAL

        NOUMRS = 1              ; NUMBER OF PERMANENT UMRS REQUIRED

; DY CHARACTERISTICS

        DDNBLK  = DYDSIZ*2              ;DOUBLE DENSITY SINGLE-SIDED

        DYNREG  = 3                     ;# OF REGISTERS TO READ FOR ERROR LOG.
        RETRY   = 8.                    ;RETRY COUNT
        SPFUNC  = 100000                ;SPECIAL FUNCTIONS FLAG
                                        ; (IN COMMAND WORD)
; SPECIAL FUNCTION CODES

        SIZ$FN  = 373                   ;373 - GET DEVICE SIZE
                                        ;374 - UNUSED
        WDD$FN  = 375                   ;375 - WRITE WITH DELETED DATA
        WRT$FN  = 376                   ;376 - WRITE ABSOLUTE SECTOR
        RED$FN  = 377                   ;377 - READ ABSOULTE SECTOR
;NOTE: if you add a SPFUN code here also add it to .DRSPF
```

The .DRDEF macro also defines the following symbols for you:

```
        HDERR$ = 1          ;HARD ERROR BIT IN THE CSW
        EOF$   = 20000      ;END OF FILE BIT IN THE CSW
```

1.2.1.1.4 Device-Identifier Byte

The low byte of the device status word, the device-identifier byte, identifies each device in the system. You specify the correct device identifier as the *code* argument to .DRDEF. The values are defined in octal and listed under .DRDEF in the *RT–11 System Macro Library Manual*.

To create device-identifier codes for devices that are not already supported by RT–11, start by using code 377_8 for the first device, 376 for the second, and so on. This procedure should avoid conflicts with codes that RT–11 will use in the future for new hardware devices.

1.2.1.1.5 Device Status Word

The device status word identifies each unique physical device in an RT–11 system and provides other information about it, such as whether it is random or sequential access. The .DRDEF macro sets up symbols based on the parameter arguments for *code* and *stat*. The .DRBEG macro takes those symbols, builds the device status word, and stores it in block 0 of the handler file at the offset H.DSTS and in the $STAT table when the device is installed. The .DSTATUS programmed request can return this value to a running program.

Table 1–2 shows the meaning of the bits in the device status word. Except for ABTIO$ and HNDLR$, all bits have an individual meaning. The meaning of ABTIO$ and HNDLR$ is determined by their combination; they should be thought of as a pair. More information on the ABTIO$/HNDLR$ pair is found in Sections 1.3.1 and 1.3.2.

Table 1–2: Device Status Word

Bit	Symbol	Meaning
0–7	—	Device-identifier byte (see Section 1.2.1.1.4)
8	VARSZ$	0 = SF.SIZ (special function code 373) requests are invalid for this handler
		1 = SF.SIZ (code 373) requests (return volume size) are valid for this handler
9	ABTIO$†	0 = Handler is not entered at abort entry point on normal program exits
		1 = Handler is entered at abort entry point whenever a program terminates
10	SPFUN$	0 = .SPFUN requests are invalid
		1 = Handler accepts .SPFUN requests
11	HNDLR$‡	0 = Enter handler at abort entry point only if there is an active queue element belonging to the aborted job
		1 = Enter handler at abort entry point on all aborts
12	SPECL$	1 = Special directory-structured device (examples are MS and MU)
13	WONLY$	1 = This is a write-only device
14	RONLY$	1 = This is a read-only device
15	FILST$	0 = This is a sequential-access device (examples are LP, LS, MS)
		1 = This is a random-access device (examples are DU and DY)

†ABTIO$ works in combination with HNDLR$. See Section 1.3.1.
‡HNDLR$ works in combination with ABTIO$. See Section 1.3.1.

The bit combinations for handlers that internally queue I/O requests are described in Section 1.4. See Section 1.9 for details on special devices (such as magtape).

All device handlers that have bit 15 set are assumed to be RT–11 file-structured devices by most of the system utility programs.

An easy way to define the device status word is to use the symbols for the bit patterns that .DRDEF defines for you. Thus, you can create the *stat* argument by ORing together the appropriate symbols from the list below.

```
FILST$ == 100000 ;File-structured random access
RONLY$ ==  40000 ;Read-only
WONLY$ ==  20000 ;Write-only
SPECL$ ==  10000 ;Special directory structured device
HNDLR$ ==   4000 ;Enter handler on abort
SPFUN$ ==   2000 ;Accepts special functions
ABTIO$ ==   1000 ;Always take abort entry
VARSZ$ ==    400 ;Handler supports variable-size volumes
```

For example, form the *stat* argument for the DY, MS, and LS handlers as follows:

- For DY: FILST$!SPFUN$!VARSZ$
- For MS: SPECL$!SPFUN$
- For LS: WONLY$!SPECL$

1.2.1.1.6 Device Size Word

The *size* argument for the .DRDEF macro defines *dd*DSIZ to be the size of the device in 256-word blocks. The .DRDEF macro stores the value of *dd*DSIZ in H.DSIZ, offset 54 in the handler's block 0.

The .DSTAT programmed request returns the value of the device size word to a running program. For examples of the .DRDEF macro, see the device handler listings in Appendix A.

1.2.1.2 .DREST Macro

The .DREST macro places device specific information about the handler into handler block 0:

- The device class and any variation
- The presence of bad-block replacement information
- How the handler can be installed, loaded, and mounted

The format of the .DREST macro call is as follows:

Macro Call: .DREST [CLASS=str][,MOD=str][,DATA=dptr]
[,TYPE=str][,REPLACE=rptr][,STAT2=symb]

CLASS=str stores the class symbol in H.CLAS, offset 20 in handler block 0.

MOD=str stores the classification modifier in H.MOD, offset 21 in handler block 0.

DATA=dptr stores an internal table file address in H.DATA, offset 72 in handler block 0.

TYPE=str — stores an internal table device classification in H.TYPE, offset 70 in handler block 0.

REPLACE=rptr — stores a pointer to a bad-block replacement table in H.REPL, offset 32 in handler block 0.

STAT2=symb — stores a second status word in H.STS2, offset 36 in handler block 0.

See Section 1.2.1.9 for more information on the contents of handler block 0, including those offsets written by .DREST. For information on using the .DREST macro, see the *RT–11 System Macro Library Manual*.

1.2.1.3 .DRINS Macro

The .DRINS macro sets up the installation code area in the handler's block 0:

- Defines the display CSR addresses (displayed by RESORC)
- Defines the installation CSR addresses (used by INSTALL command) and monitor bootstrap
- Defines system device (INSSYS) and data device (INSDAT) installation entry points

  INSSYS is located at symbol H.ISY, offset 202. INSDAT is located at symbol H.IDK, offset 200.

The format of the .DRINS macro call is as follows:

Macro Call: .DRINS name,<csr,csr,...>

name — is the device handler name.

If *name* is preceded by a minus sign (-), it indicates that the specified CSR is for display purposes only; there is no installation CSR for this invocation of .DRINS.

csr — creates a symbolic reference to a CSR for this device. The first (or only) specified is both the installation CSR and the first display CSR. The .DRBEG macro stores the installation CSR in H.ICSR, offset 176 in block 0. The .DRINS macro stores the first display CSR in H.DCSR, offset 174 in handler block 0. (You must also specify *csr* = *NO* in .DRDER for this to take effect.)

If more than one CSR is specified, the second and any subsequent in the list are the secondary (and subsequent) display CSRs. Those are written to offset 172, 170, and so forth. The list is terminated with a word containing a zero value. (There remains a single installation CSR.)

See Section 1.2.1.9 for more information on the contents of handler block 0, including those offsets written by .DRINS. For information on using the .DRINS macro, see the *RT–11 System Macro Library Manual*.

1.2.1.4 .DRPTR Macro

The .DRPTR macro sets up pointers to handler service routines that can assist the handler when it is fetched, loaded, released, or unloaded.

The pointers are located in handler block 0. The service routines are not normally located in handlere block 0 and are not located in the handler memory image. When called, any service routine is read from the handler file image into the shared area of the USR and used by the handler.

The format of the .DRPTR macro call is as follows:

Macro Call: .DRPTR [FETCH=n][,RELEASE=n][,LOAD=n][,UNLOAD=n]

FETCH=n	stores a pointer to a fetch service routine in H.FETC, offset 2 in handler block 0.
RELEASE=n	stores a pointer to a release service routine in H.RELE, offset 4 in handler block 0.
LOAD=n	stores a pointer to a load service routine in H.LOAD, offset 6 in handler block 0.
UNLOAD=n	stores a pointer to an unload service routine in H.UNLO, offset 10 in handler block 0.

See Section 1.2.1.9 for more information on the contents of handler block 0, including those offsets written by .DRPTR. For information on using the .DRPTR macro, see the *RT–11 System Macro Library Manual*.

1.2.1.5 .DRSPF Macro

The .DRSPF macro defines a handler's support for special functions. As explained in the *RT–11 System Macro Library Manual*, two methods can be used to create that support.

The format of the .DRSPF macro call is as follows:

Macro Call: .DRSPF arg[,arg2][,TYPE=n]

Up to three groups of special functions can be described in symbols H.SPF1, H.SPF2, and H.SPF3, beginning at offset 22 in handler block 0. Any further groups require the *extension table method*, which are stores in the pointer symbol H.SPFX at offset 30 in handler block 0. The offset H.SPFX points to that extension table of other supported special functions.

See Section 1.2.1.9 for more information on the contents of handler block 0, including those offsets written by .DRSPF. For information on using the .DRSPF macro, see the *RT–11 System Macro Library Manual*.

1.2.1.6 .DRTAB Macro

The .DRTAB macro is normally reserved for use by Digital. Although .DRTAB is described in the *RT–11 System Macro Library Manual*, you should use .DRUSE in your handler.

1.2.1.7 .DRUSE Macro

The .DRUSE macro defines a list of data tables for the device handler. There are three levels of definition.

1. You write a data table (or tables) at some file address (or addresses) in your device handler. You invoke .DRUSE enough times to define each data table. To invoke .DRUSE, see the *RT–11 System Macro Library Manual*.
2. At a file address, the .DRUSE macro creates a descriptor table of those data tables. The descriptor table is described in Section 1.2.8.7.
3. The .DRUSE macro places a pointer to the descriptor table file address in H.USER, at offset 106 in the handler's block 0.

The format of the .DRUSE macro call is as follows:

Macro Call: .DRUSE type,addr,size

type	stores the value of *type* at symbol DT.ID in the descriptor table
addr	stores the value of *addr* as symbol DT.PTR in the descriptor table
size	stores the value of *size* as symbol DT.SIZ in the descriptor table

1.2.1.8 .DRSET Macro

The .DRSET macro must be invoked in the preamble section of the device handler. Invoking .DRSET and the structure of the SET tables it creates are described in Section 1.5.2.

1.2.1.9 Information in File Image Block 0

Table 1–3 describes the contents of block 0 of the assembled handler file image. This is the informational block and is not normally loaded into memory.

The symbol names in the table are those used in the distributed system definition library file, SYSTEM.MLB. The macros are those that actually write the offset; they are not necessarily the originating macro. Where appropriate, the description indicates where you can find more information about the offset, its contents, or the structure pointed to by an address in the offset.

Table 1–3: Contents of .SYS Image Block 0

Offset	Symbol	Macro	Description
000000	H.HAN	.DREST	Handler identifier in RAD50
	H.HANV		Value for H.HAN (RAD50 HAN)
000002	H.FETC	.DRPTR	Pointer to a FETCH service routine; See Section 1.2.8.1
000004	H.RELE	.DRPTR	Pointer to a RELEASE service routine; See Section 1.2.8.1
000006	H.LOAD	.DRPTR	Pointer to a LOAD service routine; See Section 1.2.8.1

Table 1–3 (Cont.): Contents of .SYS Image Block 0

Offset	Symbol	Macro	Description
000010	H.UNLO	.DRPTR	Pointer to an UNLOAD service routine; See Section 1.2.8.1
000012–000016			Reserved
000020	H.CLAS	.DREST	Device classification; See Section 1.2.1.2
000021	H.MOD	.DREST	Device classification modifier; See Section 1.2.1.2
000022	H.SPF1	.DRSPF	First special function (index method) list; See Section 1.2.8.2
000024	H.SPF2	.DRSPF	Second special function (index method) list; See Section 1.2.8.2
000026	H.SPF3	.DRSPF	Third special function (index method) list; See Section 1.2.8.2
000030	H.SPFX	.DRSPF	Pointer to further special functions (extension table method); See Section 1.2.8.2
000032	H.REPL	.DREST	Pointer to bad-block replacement table; See Section 1.2.8.3
000034			Reserved
000036	H.STS2	.DREST	Second status word; See Section 1.2.8.5
000040–000050			SYSCOM area for runnable handlers.
000052	H.SIZ	.DRBEG	Handler size (*dd*END–*dd*STRT)
000054	H.DSIZ	.DRBEG	Device size (*dd*DSIZ); See Section 1.2.1.1.6
000056	H.DSTS	.DRBEG	Device status word (*dd*STS); See Section 1.2.1.1.5
000060	H.GEN	.DREND	Result of standard SYSGEN conditionals OR'd with the value of the FORCE= parameter; See Section 1.2.8.6
000061			Reserved
000062	H.BPTR	.DRBOT	Pointer to the primary bootstrap; See Section 1.11.2.2
000064	H.BLEN	.DRBOT	Bootstrap size in bytes; See Section 1.11.2.1
000066	H.READ	.DRBOT	Pointer to the bootstrap read routine; See Section 1.11.2.5
000070	H.TYPE	.DRTAB .DREST	If contains value –1, indicates written by .DRTAB (only Digital distributed handlers)—otherwise: If contains a RAD50 value, indicates invoked by .DREST and is the device type classification for an internal table

Table 1–3 (Cont.): Contents of .SYS Image Block 0

Offset	Symbol	Macro	Description
000072	H.DATA	.DRTAB .DREST	If H.TYPE written by .DRTAB, then H.DATA is a pointer to the list of handler data table descriptors. If H.TYPE written by .DREST, then H.DATA is the file address of the internal data tables. See Section 1.2.8.7
000074	H.DLEN	.DRTAB .DREST	Size in bytes of total list of handler data table descriptors; See Section 1.2.8.7
000076	H.UNIT	.DRDEF	Pointer to extended device-unit ownership table
000100	H.64UM	.DRDEF	Letter name of extended device-unit handler and device characteristics for UMR support; See Section 1.2.8.8
000102– 000104			Reserved
000106	H.USER	.DRUSE	Pointer to the file address of the handler data descriptor table; See Section 1.2.8.7
000110– 000173		.AUDIT .MODULE	Information written by those two macros. Terminated by –1. This list and the display CSR list cannot overlap.
000164– 000174	H.DCSR	.DRINS	Display CSRs read by RESORC. If more than one, each written into previous offset; See Section 1.2.1.3
000176	H.ICSR	.DRDEF .DRINS	Installation CSR; See Sections 1.2.1.1 and 1.2.1.3.
000200	H.IDK	.DRINS	Data device installation entry point (INSDAT); See Section 1.2.1.3
000202	H.ISY	.DRINS	System device installation entry point (INSSYS); See Section 1.2.1.3
000204– 000377			Installation code; See Section 1.14.3.5
000400– 000777	H.SET	.DRSET	SET code; See Section 1.5.2

1.2.2 Header Section

The second part of an RT–11 device handler is the header section. The header section is the beginning of the memory resident portion of the handler and starts at the base of file image block 1. In the header section, you invoke the .DRBEG macro to build a data structure of variable size at the beginning of the handler's memory image. This macro also stores information in the handler file at offsets 52 through 60 of block 0, and creates some global symbols.

The data you set up in the header section is used when the handler is brought into memory with the .FETCH programmed request or LOAD monitor command. The contents of location 176, described below, are used by the bootstrap when it checks for the presence of device hardware at handler installation time.

As shown in the skeletal handler, Figure 1–1, you include macros in the preamble section that build various data structures and define symbols. The following macros can be used in the header section:

- .DRBEG

  Defines the handler queue entry point and provides other information about the handler. Writes locations in the handler file image blocks 0 and 1.

- .DRVTB

  Defines multiple vectors if the handler supports more than one interrupt vector.

1.2.2.1 .DRBEG Macro

The .DRBEG macro sets up offsets in block 0 and the header information in block 1. This macro also generates the appropriate global symbols for your handler. Before you invoke .DRBEG, invoke .DRDEF to define various symbols that .DRBEG uses internally. The format for .DRBEG is as follows:

.DRBEG name[,SPFUN=spsym][,NSPFUN=nspsym]

name is the two-character device name.

spsym is the label on the list of DMA standard special functions. Sets HF2.SD in offset H1.FG2 of handler block 1.

nspsym is the label on list of DMA nonstandard special functions. Sets HF2.ND in offset H1.FG2 of handler block 1.

For examples of .DRBEG, see the DL handler listing in Appendix A and the UB example in Chapter 2.

1.2.2.2 Multivector Handlers: .DRVTB Macro

An RT–11 device handler can service multiple controllers where each controller has an interrupt vector. The handler can also service a device that has more than one vector.

Device handlers support a single vector through the .DRDEF macro's *vec* parameter. A device handler that supports multiple vectors must contain the .DRVTB macro. Invoke the .DRVTB macro once for each vector. Each invocation creates a table with three entries. The table for each vector consists of the vector location, the interrupt entry point, and the Processor Status, or PS, value.

You can invoke .DRVTB anywhere between the .DRBEG macro and the .DREND (or .DRBOT) macro, as long as it does not interfere with the flow of control within the handler. You must invoke this macro once for each vector, and the macro calls must appear one after the other in the handler.

The format of the .DRVTB macro is as follows:

.DRVTB name,vec,int[,ps]

name is the two-character device name. Specify it on the first .DRVTB call; leave this argument blank on all subsequent calls.

vec is the location of the vector; it must be between 0 and 474. The first vector is usually *dd*$VEC. The value must be a multiple of 4. The .DRBEG stores the value for *dd*$VEC in H1.VEC, offset 0 of block 1.

int is the symbolic name of the interrupt handling routine; it must appear elsewhere in the handler. It generally takes the form *dd*INT, where *dd* represents the two-character device name. The .DRBEG stores the value for *dd*INT in H1.ABT, offset 2 of block 1.

ps is an optional value you can use to specify the low-order four bits of the new Processor Status word in the interrupt vector. If you omit this argument, it defaults to 0.

An example of a handler that can use two vectors is the DY handler, when that handler is built to support a second controller. The following example shows the source lines and the code the macros generate:

```
.IF NE DYT$O                          ; If we support two controllers:
.DRVTB  DY,DY$VEC,DYINT               ; DY$VEC symbol for first vector table
.DRVTB  ,DY$VC2,DYINT                 ; DY$VC2 symbol for second vector table
                        .ASSUME . LE DYSTRT+1000
.ENDC ;NE DYT$O
```

Generates:

```
.IF NE DYT$O
.DRVTB  DY,DY$VEC,DYINT
.WORD   DY$VEC&^C3.,DYINT-.,^o340!0,^o100000
.DRVTB  ,DY$VC2,DYINT
.WORD   DY$VC2&^C3.,DYINT-.,^o340!0,^o100000
.ASSUME . LE DYSTRT+1000
.ENDC ;NE DYT$O
```

In the example above, the priority bits of the PS are always set to PR7, even if you omit the *ps* argument.

PS Condition Codes

In the .DRVTB macro, only the condition code bits of the *ps* argument are significant. These can be useful if you have a common interrupt service entry point for two or more vectors and you need to determine through which vector the interrupt occurred. For example, the skeletal handler (Figure 1–1) has a single interrupt entry point for its two vectors. For the handler to determine the source of the interrupt, one is serviced with the carry bit clear and the other (*INT2*), when the carry bit is set.

1.2.2.3 Information in File Image Block 1

The following table describes the contents of block 1 of the assembled handler file image that are written by the .DRBEG macro. This is the first block that is normally loaded into memory and is therefore block 0 of the handler memory image.

The symbol names used in the table are from the distributed system definition library file, SYSTEM.MLB. All defined offsets are written by .DRBEG but .DRBEG is not the originating macro for all locations. As appropriate, the description indicates where you can find more information about each offset, its contents, or the structure pointed to by an address in the offset.

Table 1–4: Contents of .SYS Image Block 1

Offset	Symbol	Macro	Description
001000	H1.VEC	.DRBEG	Either the device vector if a single vector device or an offset to the table of vectors for multivector devices (*dd*STRT)
001002	H1.ABT	.DRBEG	Offset to the interrupt service entry point
001004	H1.HLD	.DRBEG	Priority (340)
001006	H1.LQE	.DRBEG	Pointer to the last queue element (*dd*LQE)
001010	H1.CQE	.DRBEG	Pointer to the current queue element (*dd*CQE) in handler memory image
001010	H1.FLG	.DRBEG	Flag word (in handler file image); See Section 1.2.9.1
001012	H1.NOP	.DRBEG	NOP instruction OR'd with flags; See Section 1.2.9.2
001014	H1.BR	.DRBEG	Branch instruction (optional)
001016	H1.FG2	.DRBEG	Second flag word (optional); See Section 1.2.9.3
001020	H1.SCK	.DRBEG	Pointer to SPFUN address check routine (optional)
001022	H1.SDF	.DRBEG	Pointer to standard DMA SPFUN table (optional)
001024	H1.LDT	.DRBEG	Pointer to LD translation table (optional)
001026	H1.NDF	.DRBEG	Pointer to nonstandard DMA SPFUN table (optional)

1.2.3 I/O Initiation Section

The I/O initiation section contains the first executable instructions of the handler and must follow the call to .DRBEG. The purpose of the code in this section is to start a data transfer. Remember that you must write Position-Independent Code (PIC) for the handler.

When a program issues a programmed request that requires device I/O, such as .READ or .WRITE, control first passes to the Resident Monitor, which then calls the device handler for the peripheral device with the CALL instruction. The monitor calls the handler at the handler's sixth word—that is, the first word immediately after the five-word data header. The monitor makes the call whenever a new queue element becomes the first element in a handler's queue. This situation occurs when an element is added to an empty queue, or when an element becomes first in a queue because a prior element was released. If any parameters in the I/O request are invalid for the device (for example, the block number is too large, the unit number is too high, and so on), the handler should proceed immediately to the I/O completion section and signal a hard (fatal) error.

The I/O initiation code executes at processor priority 0 in system state, which means that no context switch can occur, no completion routines can run, and any traps to 4 and 10 cause a system fatal halt. All registers are available for you to use in this section. The fifth word of the handler header, *dd*CQE, contains a pointer to the current queue element at its third word, Q.BLKN.

The queued I/O system guarantees that requests for data transfers are serialized so that RT–11 device handlers need not be re-entrant. Therefore, you can minimize the size of a handler by mixing, rather than separating, the pure code and the data segments.

1.2.3.1 Guidelines for Starting the Data Transfer

Since the purpose of the I/O initiation section is to start up the data transfer, you must now supply the instructions to do this. The following steps (from the RK handler) represent guidelines for a generalized I/O initiation section:

1. You should have already decided how many times the handler will retry a transfer should an error occur. Initialize a retry counter by moving the maximum number of retries to it. The following two lines of code illustrate this step:

```
        MOV     #RKCNT,(PC)+            ;RKCNT = MAXIMUM # OF RETRIES
RETRY:  .WORD   0                       ;THE RETRY COUNTER
```

2. Put the pointer to the current queue element into a register, and get the device unit number and the block number for the transfer from the queue element. The following lines of code illustrate this.

```
        MOV     RKCQE,R5                ;GET CURRENT QUEUE ELEMENT POINTER
        MOV     @R5,R2                  ;PICK UP BLOCK NUMBER
        MOV     Q$UNIT-1(R5),R4         ;GET REQUESTED UNIT NUMBER
        ASR     R4                      ;SHIFT UNIT NUMBER
        ASR     R4                      ; TO HIGH 3 BITS
        ASR     R4                      ;  OF LOW BYTE
        SWAB    R4                      ;PUT UNIT NUMBER IN HIGH 3 BITS
        BIC     #^C<DAUNIT>,R4   ;ISOLATE UNIT IN DRIVE SELECT BITS
```

3. Next, perform the steps to calculate the address on the device for the data transfer to begin. The instructions you use depend on the device's structure, of course. Once you have calculated the correct address, save it in a memory location. If you need to retry this transfer, you will not have to recalculate the address.

```
        .
        .
        .
        MOV     R3,(PC)+                ;SAVE ADDRESS IN DISKAD
DISKAD: .WORD   0                       ;SAVE CALCULATED ADDRESS HERE
```

4. Steps 1 through 3 outlined above are executed only once for each data I/O request from a running program. However, in case of a soft error, you may need to restart a transfer as part of the retry operation. So, by placing a label here to use as the retry entry point, you avoid repeating steps 1 through 3.

   The following steps can be performed more than once. They are executed once for the first I/O startup, and they can be executed again if an I/O error causes a retry.

   At this point, the handler should determine whether the I/O request is a read, a write, or a seek. It should then generate the appropriate op code for the operation and move it to the device control and status register. This step actually initiates the I/O transfer.

```
        CSIE =          100             ;INTERRUPT ENABLE
        FNWRITE =       12              ;WRITE
        CSGO =          1               ;GO BIT
        .
        .
        .
AGAIN:  MOV     RKCQE,R5                ;POINT TO QUEUE ELEMENT
        MOV     #CSIE!FNWRITE!CSGO,R3   ;ASSUME A WRITE
        MOV     #RKDA,R4                ;POINT TO DISK
        .                               ;ADDRESS REGISTER
        .
        .
```

5. Finally, return to the interrupted program by going through the monitor first. Then when the I/O transfer finishes, the device will interrupt, and control will pass to the handler at the interrupt entry point in the interrupt service section of the handler.

```
        RTS     PC                      ;AWAIT INTERRUPT
```

1.2.3.2 Transferring the Data

Data can be transferred between a device and the user buffer as individual bytes, words, or by direct memory access (DMA). How the data is transferred is largely determined by whether or you are using a mapped or unmapped monitor. This section describes transferring the three types of data into both unmapped and mapped memory.

1.2.3.2.1 Byte Transfer from the User Buffer to the Device

The following examples are from the XL handler and illustrate transferring a byte from the user buffer.

Unmapped Monitor

```
GNXTCH: MOV     XOCQE,R4                ;R4->current output queue element
        BEQ     10$                     ;None available...
        ADD     #Q$WCNT,R4              ;R4->word count
        TST     @R4                     ;Any characters left to output?
        BEQ     20$                     ;Nope, this request is complete
        INC     @R4                     ;Yes, now there is one less to do
        MOVB    @-(R4),R5               ;Get the byte to output
        INC     @R4                     ;bump pointer to next byte
```

Mapped Monitor

RT–11 provides the $GTBYT routine to perform the address translation between a user buffer in mapped memory and the device. The $GTBYT routine is described in more detail in Section 1.10.4.1.

Before the call:

R4 must point to Q.BLKN, the third word in the queue element.

After the call:

(SP), the first word on the stack, contains the next byte from the user buffer in the low byte. The contents of the high byte are not defined.

R4 is unchanged.

```
GNXTCH: MOV     XOCQE,R4                ;R4->current output queue element
        BEQ     10$                     ;None available...

        TST     Q$WCNT(R4)              ;Any characters left to output?
        BEQ     20$                     ;Nope, this request is complete
        INC     Q$WCNT(R4)              ;Yes, now there is one less to do
        CALL    @$GTBYT                 ;Get the byte to output
        MOV     (SP)+,R5
```

The buffer address (Q.BUFF) in the queue element is updated by 1. If Q.BUFF is greater than 20077, a 1 is added to Q.PAR and Q.MEM and Q.BUFF is reduced by 100.

1.2.3.2.2 Byte Transfer from the Device to the User Buffer

The following examples are from the XL handler and illustrate transferring a byte into the user buffer.

Unmapped Monitor

```
30$:
        ADD     #Q$WCNT,R4              ;R4->Word count
        MOVB    R5,@-(R4)               ;Return the character
        INC     (R4)+                   ;Bump the buffer pointer
        DEC     (R4)                    ;Is transfer complete?
                                        ;  (z-bit=1 if so)
```

Mapped Monitor

RT–11 provides the $PTBYT routine to perform the address translation between a user buffer in mapped memory and the device. The $PTBYT routine is described in Section 1.10.4.2.

Before the call:

R4 must point to Q.BLKN, the third word in the queue element.

The byte to transfer to the user buffer must be on the top of the stack. The character must be in the low byte of the stack's first word. The high byte is unpredictable.

After the call:

The word containing the character to transfer is removed from the stack and transferred to the user buffer.

R4 is unchanged.

```
30$:
        MOVB    R5,-(SP)                        ;Put character here for PUTBYT
        CALL    @$PTBYT                         ;Call the routine
        DEC     Q$WCNT(R4)                      ;Is transfer complete?
                                                ; (z-bit=1 if so)
```

The buffer address (Q.BUFF) in the queue element is updated by 1. If Q.BUFF is greater than 20077, a 1 is added to Q.PAR and Q.MEM and Q.BUFF is reduced by 100.

1.2.3.2.3 Word Transfer from the Device to the User Buffer

The handler may have to change a word in user memory. The following examples are taken from the DY handler and return a word of size information.

Unmapped Monitor

```
; DRIVER IS DUAL DENSITY ONLY

        BIS     #CSDN,R4        ;ALWAYS USE DOUBLE DENSITY
        CMPB    R1,#SIZ$FN      ;SPECIAL SIZE FUNCTION?
        BNE     3$              ;NO, CONTINUE
        MOV     #DDNBLK,@(R5)+  ;RETURN DOUBLE DENSITY SIZE
        JMP     DYDONE          ;DONE WITH SIZE OPERATION
```

Mapped Monitor

RT–11 provides the routine $PTWRD to perform the address translation between the device and a user buffer. The $PTWRD routine is described in Section 1.10.5.

```
; DRIVER IS DUAL DENSITY ONLY

        BIS     #CSDN,R4        ;ALWAYS USE DOUBLE DENSITY
        CMPB    R1,#SIZ$FN      ;SPECIAL SIZE FUNCTION?
        BNE     3$              ;NO, CONTINUE
        MOV     #DDNBLK,-(SP)   ;RETURN DOUBLE DENSITY SIZE
        MOV     DYCQE,R4        ;CURRENT QUEUE ELEMENT
        CALL    @$PTWRD         ;STORE SIZE IN BUFFER
        JMP     DYDONE          ;DONE WITH SIZE OPERATION
```

The buffer address (Q.BUFF) in the queue element is updated by 2. If Q.BUFF is greater than 20077, a 1 is added to Q.PAR and Q.MEM and Q.BUFF is reduced by 100.

1.2.3.2.4 Non-DMA Transfers

The following examples are from the DY handler and illustrate getting a pointer to the user buffer for use in DMA transfer initialization.

Unmapped Monitor

```
3$:
        MOV     (R5)+,R0        ;GET THE USER'S BUFFER ADDRESS
        MOV     @R5,WRDCNT      ;GET WORD COUNT
        BPL     4$              ;POSITIVE MEANS READ, SO ALL SET UP
```

Mapped Monitor

RT–11 provides the $MPMEM routine to perform address translation for non-DMA transfers between the device and a user buffer. Non-DMA transfers are typically done with the MOV instruction. The $MPMEM routine is described in Section 1.10.3.1.

```
3$:
        CALL    @$MPPTR         ;CONVERT MAPPED ADDRESS TO PHYSICAL ADDRESS
        MOV     (SP)+,R0        ;GET PHYSICAL BUFFER ADDRESS LOW ORDER BITS
        MOV     R4,(PC)+        ;SAVE CURRENT COMMAND WORD
35$:    .BLKW
        MOV     (SP)+,R4        ;GET HIGH-ORDER ADDRESS BITS <21:18>
        BIT     #1700,R4        ;22-BIT ADDRESS SPECIFIED?
        BNE     DYERR           ;YES, NOT VALID FOR THIS CONTROLLER
        SWAB    R4              ;MOVE TO CORRESPONDING POSITIONS IN HIGH BYTE
        BIS     35$,R4          ;NOW MERGE COMMAND WORD WITH EXTENSION BITS

        MOV     @R5,WRDCNT      ;GET WORD COUNT
        BPL     4$              ;POSITIVE MEANS READ, SO ALL SET UP
```

1.2.3.2.5 DMA Transfers

The address translation for DMA transfers is performed by the $MPPHY routine, described in Section 1.10.3.2. A complete description of doing DMA transfers using UNIBUS mapping registers (UMRs) is in Section 2.13.

1.2.4 Interrupt Service Section

Control passes to the interrupt service section of the handler when a device interrupts, when the program requesting the I/O transfer aborts, or a .ABORT is issued for the channel. The code in this section must first determine if the data transfer had an error, if it was incomplete, or if it was complete, and then take the appropriate action. The same register usage restrictions that apply to the interrupt entry point also apply to the abort entry point. See Chapter 5 in the *RT–11 System Internals Manual* for information on interrupt service routines.

Your first step in coding the interrupt service section is to set up the interrupt entry point and the abort entry point by using the .DRAST macro. (These entry points are sometimes referred to as the asynchronous trap entry points.) The default name for the interrupt entry point is *dd*INT, where *dd* is the device name. Under normal conditions, the handler is called at the interrupt entry point when an interrupt occurs. However, under some circumstances, the handler is called at the abort entry point located at *dd*INT–2. The various situations are discussed in the following sections.

1.2.4.1 .DRAST Macro

Use the .DRAST macro to set up the interrupt entry point and the abort entry point, and to lower the processor priority. The .DRDEF and .DRVTB macros fill in the structure at bootstrap (for the system device) or at .FETCH time (for a data device).

The format of the .DRAST macro is as follows:

.DRAST name,pri[,abo]

name is the two-character device name.

pri is the priority of the device, and the priority at which the interrupt service code is to execute.

abo is an optional argument that represents the label of the abort entry point. If you omit this argument, the macro expansion generates a RETURN instruction at the abort entry point. Either the branch to the specified label or the RETURN instruction is the word immediately preceding the interrupt entry point *dd*INT.

The following example from the DY handler shows the .DRAST macro call. In the example, DYABRT is the label for the abort routine which would generate the instruction BR DYABRT in the word preceding the interrupt entry point DYINT.

```
        .SBTTL  INTERRUPT ENTRY POINT

        .DRAST  DY,5,DYABRT           ; AST entry point
        BR      DYABRT                ; Jump to abort entry point
DYINT:: JSR     R5,@$INPTR            ; Jump to monitor INTEN code
        .WORD   ^C<5*^o40>&^o340      ; New priority
        .FORK   DYFBLK                ; Request fork level immediately
        JSR     R5,@$FKPTR            ; Jump to monitor fork code
        .WORD   DYFBLK-.              ; Offset to fork queue element
        CALL    SETDY                 ; Setup registers
        BMI     DYERR2                ; Check out the error and retry
INTDSP: JMP     @(PC)+                ; No error, return to called
INTRTN: .WORD   0                     ; : Address of waiting routine
```

The next example, from the RK handler, does not have an abort routine. Notice the instruction, RETURN, in the word immediately preceding the interrupt entry point RKINT.

```
        .DRAST  RK,5

        .GLOBL  $INPTR                  ;MAKE THIS SYMBOL GLOBAL
        RETURN                          ;JUST RETURN ON ABORT
RKINT:: JSR     R5,@$INPTR              ;JUMP TO MONITOR INTEN CODE
        .WORD   ^C<5*^O40>&^O340        ;NEW PRIORITY
```

1.2.4.2 Abort Entry Point

As described in Section 1.3, there are a number of situations that cause an abort in the queued I/O system. The response to the abort situation by the handler and RMON depends on the ABTIO$ and HNDLR$ bits in the device status word.

When an abort occurs, it is important to stop I/O on some devices. Character-oriented devices, such as the communications handler XL, fall into this category.

So, character-oriented devices generally contain an abort routine; the abort entry point is simply a branch instruction to that routine. The following lines are from the XL handler:

```
XLDONE:
   .
   .
   .
        BIC     #RC.IE,@XIS             ;Turn off input interrupts
   .
   .
   .
        RTS     PC                      ;Return to monitor
```

Other devices, such as disks, should be allowed to complete an I/O transfer attempt, even if an abort occurs. In fact, trying to abort in the middle of an operation can corrupt data or formatting information on a disk. So, instead of having a separate abort routine, most handlers for disks ignore an abort. Thus, a RETURN instruction is located at the abort entry point, which simply returns control to the monitor.

The abort entry point is always located at the word previous to the interrupt entry point (*dd*INT–2). If the optional .DRAST *abo* parameter is specified, the abort entry point is a branch instruction to the label specified as the *abo* parameter argument. If *abo* is not specified, the .DRAST macro expansion places a RETURN instruction at the abort entry point (*dd*INT–2).

If you use .FORK in your handler, there is a special procedure you must follow if an abort occurs. You must move 0 to F.BADR (the fork routine address, at offset 2) in the fork block. This prevents the monitor from attempting to execute a meaningless fork routine after the abort.

1.2.4.3 Lowering the Priority to Device Priority

When the interrupt occurs, the handler is entered at priority 7. As with interrupt service routines, the handler's first task is to lower the processor priority to the priority of the device, thus permitting more important devices to interrupt this service routine. Instead of using the .INTEN call, as in an interrupt service routine, use the .DRAST macro to lower the priority.

1.2.4.4 Guidelines for Coding the Interrupt Service Section

Since the purpose of this section is to evaluate the results of the last device activity, you must now supply the instructions to do this. Essentially, the code must determine if the transfer was in error, if it was incomplete, or if it was complete.

1. **If an Error Occurred**

   If an error occurred during the transfer, the handler must distinguish between a hard error and a soft error that might vanish if the operation is retried.

   If the error is hard, the handler should immediately exit through the I/O completion section after setting HDERR$ in the CSW.

If the error is soft, the handler should prepare to retry the transfer. It should decrement the count of available retries. Then, possibly at fork level, it should branch back to the I/O initiation section to restart the transfer. If the transfer has already been retried enough times (the retry count is 0), treat the failure as though it were a hard error. In that case, the handler should proceed to the I/O completion section after setting HDERR$ in the CSW.

Note that dropping to fork level is not strictly required to process an error. Whether or not to use .FORK depends on the length of time required for setting up the retry. The .FORK call is especially useful because it gives you use of R0 through R3, thus permitting you to use common routines for the retry. If you do not use .FORK, only R4 and R5 are available.

2. **Perform Retries at Fork Level**

As also described in the *RT–11 System Internals Manual*, the .FORK macro causes a return to the Resident Monitor, which dismisses the current interrupt. The code that follows .FORK executes at priority 0, rather than at device priority, after all other interrupts have been serviced, but before any jobs or their completion routines can execute. The code following .FORK executes, as does the main body of the interrupt service section of the handler, in system state. (This is the same state the I/O initiation section runs in.) Thus, context switching is prevented while the fork level code is executing, and any traps to 4 and 10 cause a system fatal halt.

The following example from the RK handler illustrates how the handler drops priority to fork level to retry data transfers after a soft error occurred. Fork level is ideal for performing the retries, since this may be a lengthy process. The .FORK call and its expansion are as follows:

```
        .FORK   RKFBLK                     ;THE FORK CALL

        JSR     R5,@$FKPTR                 ;(JUMP TO MONITOR FORK CODE)
        .WORD   RKFBLK - .                 ;(OFFSET TO FORK QUEUE ELEMENT)

RKRETR: CLRB    RETRY+1                    ;RESET A FLAG
        BR      AGAIN                      ;BRANCH INTO I/O INIT SECTION
```

3. **If the Transfer Was Incomplete**

In general, a transfer is considered to be incomplete when there are more characters or more blocks of data left to transfer. The handler should restart the device and exit with a RETURN instruction to wait for the next interrupt.

4. **If the Transfer Was Complete**

When the transfer is complete, the handler can simply exit through the I/O completion section.

1.2.5 I/O Completion Section

The I/O completion section provides a common exit path to inform the monitor that the handler is done with the current request, so that the monitor can release the current queue element.

The I/O completion section is an extension of the interrupt service section. Control passes from the interrupt service section to the I/O completion section when a data transfer completes, when a hard error is detected, or when a soft error condition exhausts the number of allowed retries.

(Note that you can branch directly to this section from the I/O initiation section if you immediately detect a hard error.)

1. **If an Error Occurred**

   There are two kinds of errors that cause control to pass to the I/O completion section: hard errors, which should cause a branch to this section immediately, and soft errors that have exhausted their allotted number of retries, which cause a branch to this section after the last retry fails. Treat both cases alike in handling the exit to the monitor.

   First, set the hard error bit (HDERR$), bit 0, in the Channel Status Word for the channel. The second word of the I/O queue element, Q.CSW, points to the Channel Status Word. Then jump to the I/O completion routine in the Resident Monitor. Use the .DRFIN macro, described below, to generate the code for this jump.

   The following lines of code are from the DY handler. They illustrate how the handler sets the hard error bit and jumps back to the monitor.

```
10$:    BIC     #<CSINIT!CSINT>,@DYCSA ;DISABLE FLOPPY INTERRUPTS
                                       ;AND INHIBIT DRIVE RESET
11$:    .DRFIN  DY              ;GO TO I/O COMPLETION
        .
        .
        .
DYERR:  MOV     DYCQE,R4        ;R4 -> CURRENT QUEUE ELEMENT
        BIS     #HDERR$,@-(R4)  ;SET HARD ERROR IN CSW
        BR      10$             ;EXIT ON HARD ERROR
```

2. **If the Transfer Was Complete**

   For a block-oriented device, such as a disk or diskette, the handler simply disables interrupts and performs the jump to the monitor. The .DRFIN macro generates the code to perform the jump.

   For a character- or word-oriented device, the procedure is slightly more complicated because the handler may have to report end-of-file to the job that requested the I/O transfer. When the handler actually detects the EOF condition on a READ operation, it should set an internal EOF flag, put the last character in the user's buffer, and then zero-fill the rest of the buffer. Then the handler should jump back to the monitor, as it would if EOF were not detected but the buffer had simply filled up. The handler waits until it is called again to signal EOF to the user.

This convention for indicating end-of-file makes character-oriented devices appear to programs as random-access devices, which is in keeping with the RT–11 philosophy of device independence.

.DRFIN Macro

Use the .DRFIN macro to generate the instructions for the jump back to the monitor at the end of the handler I/O completion section. The macro makes the pointer to the current queue element a global symbol, and it generates Position-Independent Code for the jump to the monitor. When control passes to the monitor after the jump, the monitor releases the current queue element.

The format of the .DRFIN macro is as follows:

.DRFIN name

name is the two-character device name.

For examples of the .DRFIN macro, see the handler listings in Appendix A.

1.2.6 Handler Termination Section

The purpose of the handler termination section is to declare some global symbols and to establish a table of pointers to locations in the Resident Monitor. The pointers are filled in by the bootstrap, if the handler is for the system device. Otherwise, they are filled in when the handler is made resident with .FETCH or LOAD. The termination section also provides a symbol to determine the size of the handler. Use the .DREND macro to generate the handler termination code.

1.2.6.1 .DREND Macro

The format of the .DREND macro is as follows:

.DREND name

name is the two-character device name.

In bootable handlers, the .DREND macro is invoked twice, once explicitly by the programmer and once implicitly by the .DRBOT macro. When .DRBOT is invoked, it implicitly generates a .DREND macro to close the memory resident part of the handler. You end the boot area with a second .DREND macro.

For examples of the .DREND macro, see the handler listings in Appendix A. The symbols defined by .DREND are shown in Table 1–11.

1.2.7 Pseudodevices

You can write a device handler for a pseudodevice (one that does not interrupt, and is not a mass storage device) to take advantage of the queued I/O system and the fact that handlers can remain memory resident. Examples of handlers for pseudodevices are NL (the null device), MQ (the message queue handler), SL (the single-line command editor), and UB (the UMR handler).

All the executable code of such a handler must appear in the I/O initiation section. The handler should then issue the .DRFIN macro call to terminate the operation and return the queue element. Since pseudodevices do not interrupt, the handler needs no interrupt service section and no .DRAST macro call.

1.2.8 Handler Data Structures Related to Block 0

The following sections describe data structures that relate to block 0 of the handler file image. The data structure can reside within block 0 or be pointed to by an address contained there.

1.2.8.1 Handler Service Routine Environment

This section describes the handler service routine entry environment and error processing. The routines are defined by the .DRPTR macro and located at a file address in the handler (See the .DRPTR section in the *RT–11 System Macro Library Manual*).

Handler Service Routine Entry Environment

The following registers and their contents constitute the handler service routine entry environment. These registers (R0 through R5) are set up by RT–11. All registers are available and none needs to be preserved.

Table 1–5: Handler Service Routine Entry Environment

Register	Contents
R0	Contains starting address of the current running handler service routine.
R1	Contains starting address of GETVEC routine if a CTI Bus-based processor. Otherwise, R1 contains the address of a routine that always returns carry set.

Table 1–5 (Cont.): Handler Service Routine Entry Environment

Register	Contents
R2	Contains the value $SLOT*2. That value is the length of the $PNAME table in bytes. You can use that value to locate information in the handler tables concerning this handler. The following table shows the order in memory and size in bytes, relative to $SLOT*2, of the pertinent handler tables and the contents of those tables:

Table	Size	Contents
$OWNER:	<$SLOT*2>*2	Ownership table; can be removed from (generated out of) monitors
$UNAM1:	<$SLOT*2>+4	Physical name of device table
$UNAM2:	<$SLOT*2>+4	Logical name of device table
$PNAME:	$SLOT*2	Installed handlers table
$ENTRY:	<$SLOT*2>+2	Handler address table. Last word contains value –1 and indicates end of table
$STAT:	$SLOT*2	DSTATUS value table
$DVREC:	$SLOT*2	Handler disk block table
$HSIZE:	$SLOT*2	Handler memory size table
$DVSIZ:	$SLOT*2	Device blocks table
$PNAM2:	<$SLOT*2>+2	Optional physical device name table for extended-unit (single letter) device names. Last word contains default device name, if assigned

You can use that table in the following manner. R5 contains the $ENTRY table entry address for this device handler. You could find, for example, the name for this handler in the $PNAME table by subtracting the value for $SLOT*2 from the value contained in R5. Likewise, you could find the DSTATUS value for this handler in the $STAT table by adding the value of <$SLOT*2>+2 to the value contained in R5.

See the *RT–11 System Internals Manual* for more information about the handler tables.

Table 1–5 (Cont.): Handler Service Routine Entry Environment

Register	Contents
R3	Indicates the type of entry. The value in R3 indicates the type of routine that called the handler service routine:

Value	Name	Meaning
0	HRR.FF	Entered from .FETCH
2	HRR.RE	Entered from .RELEASE
4	HRR.LO	Entered from the LOAD command
6	HRR.UN	Entered from the UNLOAD command
10	HRR.AB	Entered from a job abort (RELEASE routine)
12	HRR.SY	Entered from a system bootstrap load (LOAD routine)

Register	Contents
R4	Contains the address of a read routine you can use to perform I/O to the system device, which has been opened as non-file-structured. You must load the following registers with the following contents to use the read routine:

Register	Contents
R0	Block number to read
R1	Number of words to read
R2	Buffer address

You can read into only the low 28K words of memory. To read into high memory, you must first read into low memory and then move the data. The read routine returns with carry clear if there are no errors; carry bit is set if there are errors.

Register	Contents
R5	Contains a pointer to the $ENTRY table entry for this handler.

Handler Service Routine Error Processing

The following list shows how errors in handler service routines should be processed:

- If no errors occur, exit with carry bit clear.
- If errors occur, exit with carry bit set.

The response from RT–11 to handler service routines that exit with the carry bit set varies according to the following:

- If the handler service routine was called by the .FETCH request, RT–11 refuses to fetch the handler.

You should not depend on this response with handlers that should never be fetched; use the .DRPTR FETCH=*NO* parameter instead.

- If the handler service routine was called by the .RELEASE request, RT–11 releases the handler.
- If the handler service routine was called by the LOAD command, RT–11 refuses to load the handler.

  You should not depend on this response with handlers that should never be loaded; use the .DRPTR LOAD=*NO* parameter instead.

- If the handler service routine was called by the UNLOAD command, RT–11 refuses to unload the handler. Further RT–11 response is determined by the contents of R0:
  - If R0 is returned with value zero, RT–11 displays the error message, ?KMON-F-Unable to unload handler.
  - If R0 is returned with value other than zero, RT–11 displays the error message located at the address (in low memory) contained in R0.
- If the handler service routine was called by a job abort, RT–11 ignores the carry bit; the job aborts.
- If the handler service routine was called by a system bootstrap load, the handler can do one of the following:
  - Clear the carry bit and continue.
  - Set the carry bit and return.

    On UNIBUS and Q-bus processors, RT–11 displays the message, ?BOOT-U-Failure to load system handler, and the system halts. On CTI Bus-based processors, RT–11 displays code 000013 and the system halts.
  - Set the carry bit and send an error message to the console terminal.

    The handler sends an error message to the console terminal, using the following code:

    ```
    CODE   =: <200!DEV.xx>
    REPORT =: 672
            JSR     R1,@#REPORT
            .WORD   MSG
            .BYTE   CODE
    MSG:    .ASCIZ  "message"
            .EVEN
    ```

    For UNIBUS and Q-bus processors:

    - RT–11 ignores the contents of the byte CODE.
    - RT–11 adds the prefix "?BOOT-U-" to "message".

      (For the distributed RT–11, "message" is "Failure to load system handler".)
    - The system halts.

For CTI Bus processors:

- RT–11 ignores the contents of "message".
- RT–11 displays the octal value contained in CODE with no prefix. The value in CODE should be 200!DEV.XX, where DEV.XX is the device id for this handler. You can find DEV.XX for this handler in the $DVREC: handler table.

  (For the distributed RT–11, CODE is 000013.)
- The system halts.

1.2.8.2 Special Function Code Support Table (H.SPFx)

H.SPFx supports both the *list* and *extension table* method for describing those special functions used within the handler. Using .DRSPF to create the table is described in the *RT–11 System Macro Library Manual*.

The .DRSPF macro places the table in octal offsets 22 through 30 in the handler's block 0. Offsets 22 through 26 support the list method and each offset has the same structure and is composed of a low and high byte. Offset 30 is a word pointer to a list of other special functions.

The symbol names for the values in H.SPFx are defined in the .DSPDF macro in the distributed file SYSTEM.MLB.

The following is the structure of offsets 22 through 26.

Bit	Symbol	Meaning
Low Byte	DSP.XN	The low byte, consisting of a bit mask that specifies the supported low-order numbers (xxN):

Bit	Symbol	Meaning
001	DSP.X0	xx0 bit mask
002	DSP.X1	xx1 bit mask
004	DSP.X2	xx2 bit mask
010	DSP.X3	xx3 bit mask
020	DSP.X4	xx4 bit mask
040	DSP.X5	xx5 bit mask
100	DSP.X6	xx6 bit mask
200	DSP.X7	xx7 bit mask

Bit	Symbol	Meaning
High Byte	DSP.NX	The high byte, made up of a value to specify the type of special function and the high order numbers (NNx). Specifying a type of special function forces the table entry to a single special function.
001–004	DSP.TY	The type of special function:

Value	Symbol	Meaning
0	DSP.UK	Unknown type
1	DSP.RD	READ type
2	DSP.WR	WRITE type
3	DSP.MV	MOVEMENT type
4	DSP.RW	TRANSFER type
5–7		Reserved

Bit	Symbol	Meaning
010–200	DSP.NN	Value for the special function's high-order two numbers

As an example, assume support for special functions 372, 373, and 377 (no type specified). The contents of the table entry for these would appear in a byte dump as:

```
370 214
```

For an example that includes the TYPE parameter, assume the special function 376 of type WRITE. The contents of the table entry for that would appear in a byte dump as:

```
372 100
```

1.2.8.3 Bad-Block Replacement Geometry Table (H.REPL)

H.REPL stores the geometry of the software (not MSCP) bad-block replacement table. The .DREST macro places a pointer to this table in offset 32_8 in the handler's block 0. The table must be located in block 0.

Of the distributed RT–11 device handlers, H.REPL is found in the RL01/02 and RK06/07 handlers.

The symbol names for the values in H.REPL are defined in the .RGTDF macro in the distributed file SYSTEM.MLB.

The table consists of 1-byte entries and is 6 bytes long.

Offset	Symbol	Contents
0	RGT.FG	A flag in bit 0. If bit 0 is clear, all blocks are replaceable; if set, only some blocks are replaceable. Bits 1–7 are reserved.
1	RGT.PD	A constant for locating the bad sector file. The last addressable block plus this constant is the bad sector file location.
2	RGT.BS	Size in sectors of bad sector file.
3	RGT.TC	Number of tracks per cylinder.
4	RGT.ST	Number of sectors per track.
5	RGT.SB	Half the number of sectors per block, such that two times this number is the sectors per block.
6	RGT.SZ	Size of this table.

1.2.8.4 Bad-Block Replacement Table (HB.BAD)

The bad-block replacement table is stored in the home block of RL01/02 and RK06/07 volumes, beginning at offset 6 (HB.BAD) and ending at offset 200.

The symbol names for the values in HB.BAD are defined in the .BBRDF macro in the distributed file SYSTEM.MLB.

Offset	Name	Meaning
0	BBR.BD	Bad block number.
2	BBR.GD	Replacement block number.
	BBR.SZ	Entry size.

1.2.8.5 Second Handler Status Word (H.STS2)

The following table defines the bits in the second handler status word (H.STS2), which the .DREST macro places in offset 36_8 of block 0.

Bit	Symbol	Meaning
000001	HS2.BI	Handler cannot be installed by the monitor bootstrap.
000002	HS2.KI	Handler cannot be installed by the DCL INSTALL command.
000004	HS2.KL	Handler cannot be loaded by the DCL LOAD command.
000010	HS2.KU	Handler cannot be unloaded by the DCL UNLOAD command.
000020	HS2.MO	Handler supports DCL MOUNT and DISMOUNT commands.

Bit	Symbol	Meaning
000040–100000		Reserved.

1.2.8.6 Handler SYSGEN Options Byte (H.GEN)

The .DREND macro stores the SYSGEN option bits in H.GEN (byte offset 60_8 of block 0).

The value stored in H.GEN is the values for the SYSGEN options OR'd with the value of the .DREND FORCE= parameter.

The symbol names for the values in H.GEN are defined in the .SGNDF macro in the distributed file SYSTEM.MLB. (Note that only symbols in the range 1–200 can be used.)

Bit	Symbol	Meaning
001	ERLG$	Handler supports error logging.
002	MMGT$	Handler supports extended memory.
004	TIMIT$	Handler supports device timeout.
010	RTEM$	Handler is running under RTEM–11.
020–200		Reserved.

1.2.8.7 Handler Internal Data Table and Descriptor Structure (H.TYPE, H.DATA, and H.DLEN)

The structure described in this section is a descriptor table. That is, the structure describes tables located elsewhere in the handler. The contents and location of the structure vary according to the macro that writes it. The structure can be placed in block 0 or an address can be placed in block 0 that points to the structure:

- The .DREST or .DRTAB macro stores the structure in block 0 offsets 70 through 74. The indicated offsets are from location 70.
- The .DRUSE macro stores the structure in the handler file and writes a pointer to the structure in block 0 offset 106. The indicated offsets are from handler file address pointed to by offset 106.

The symbol names for the values in H.TYPE, H.DATA, and H.DLEN are defined in the .DUSDF macro in the distributed file SYSTEM.MLB.

Offset	Symbol	Contents
00	DT.ID	If table generated by .DREST or .DRUSE, contains the RAD50 device type identifier. If table generated by .DRTAB, contains the value –1.
02	DT.PTR	If table generated by .DREST, contains the file address of internal data tables. If table generated by .DRUSE or .DRTAB, contains the file address of the list of data table descriptors.
04	DT.SIZ	Length in bytes of the data table pointed to by this structure.
06	DT.ESZ	When table is generated by .DRUSE only, is the length in bytes of each entry in the table pointed to by this structure.
10	DT.EOL	When table is generated by .DRUSE only, is a null word that signifies the end of the descriptor list.

1.2.8.8 UMR Support and Extended Device-Unit Handlers (H.64UM)

The contents of H.64UM describes the attributes of an extended device-unit handler and the support for UNIBUS Mapping Registers (UMRs).

The .DRDEF macro writes H.64UM in octal location 100 in the handler's block 0.

The symbol names for the values in H.64UM are defined in the .HUMDF macro in the distributed file SYSTEM.MLB.

Bit	Symbol	Meaning
000001–000004	HUM.PU	Required number of permanent UMRs.
000010	HUM.S6	Handler supports other extended device-unit handlers (used in LD handler).
000020	HUM.DM	Handler uses DMA.
000040	HUM.UB	Handler includes .DRDEF macro DMA=str parameter (argument YES or NO).
000100–100000	HUM.64	Field containing RAD50 letter for extended device-unit handler.

If HUM.UB bit is clear, bits HUM.UB, HUM.DM, and HUM.PU are reserved.

IF HUM.PU bits are nonzero, HUM.DM must be set.

1.2.9 Handler Data Structures Related to Block 1

The following sections describe data structures that relate to block 1 of the handler file image. The data structure can reside within block 1 or be pointed to by an address contained there.

1.2.9.1 Handler Flag Word (H1.FLG)

H1.FLG contains flags that provide information about the handler.

The .DRBEG macro writes H1.FLG in octal location 10 of the handler's block 1 (location 1010 of the file image).

The symbol names for the values in H1.FLG are defined in the .HBFDF macro in the distributed file SYSTEM.MLB.

Bit	Symbol	Meaning
000001–004000		Reserved.
010000	DV2.DM	Handler supports DMA and is compatible with RT–11 V5.5 (and subsequent) UMR support.
020000	DV2.NL	Handler cannot be loaded by KMON; can only be loaded by BSTRAP (at bootstrap time).
040000	DV2.V2	The first vector table set up by .DRVTB is followed by a second table. The second table is only for display purposes.
100000	DV2.NF	Handler cannot be fetched but instead must be loaded.

1.2.9.2 Handler Service Routine Entry Point Word (H1.NOP)

H1.NOP describes whether entry points to various handler service routines exist. It also defines the existence of a second handler flag word (H1.FG2). The low 5 bits are significant; the other bits are used to construct a NOP instruction and can be disregarded.

The .DRBEG macro stores the entry point in H1.NOP (offset 12_8 of block 1).

The symbol names for the values in H1.NOP are defined in the .HUMDF macro in the distributed file SYSTEM.MLB.

Bit	Symbol	Meaning
000001	HNP.FE	Handler contains entry point to a FETCH service routine.
000002	HNP.RE	Handler contains entry point to a RELEASE service routine.
000004	HNP.LO	Handler contains entry point to a LOAD service routine.
000010	HNP.UN	Handler contains entry point to an UNLOAD service routine.

Bit	Symbol	Meaning
000020	HNP.F2	Handler contains a second flag word (H1.FG2).
000040	HNP.N1	Part of the NOP instruction (disregard).
000100		Reserved.
000200	HNP.N2	Part of the NOP instruction (disregard).
000400–100000		Reserved.

1.2.9.3 Second Handler Flag Word (H1.FG2)

H1.FG2 contains flags that provide additional information about the handler. If a flag indicates that a location after H1.FG2 is defined, then the preceding locations (to H1.FG2) are also defined.

The .DRBEG macro stores the second handler flag word in H1.FG2 (offset 16_8 in the handler's block 1).

The symbol names for the values in H1.FG2 are defined in the .HF2DF macro in the distributed file SYSTEM.MLB.

Bit	Symbol	Meaning
000001	HF2.SC	Handler code performs special function address checking (therefore H1.SCK exists).
000002	HF2.SD	Handler lists special functions that use DMA (therefore H1.SDF and H1.SCK exist).
000004	HF2.LD	Handler contains pointer to LD translation table (therefore H1.LDT, H1.SDF, and H1.SCK exist).
000010	HF2.ND	Handler contains nonstandard DMA special functions (therefore H1.NDF, H1.LDT, H1.SDF, and H1.SCK exist).
000020–002000		Restricted.
004000	HF2.SR	Handler requires serial satisfaction of I/O requests.
010000	HF2.DM	Handler performs DMA and is compatible with RT–11 V5.5 UMR support.
020000	HF2.S6	Handler supports other extended device-unit handlers (used in LD handler).
040000	HF2.64	Handler supports extended device-unit requests.
100000	HF2.F3	Handler contains a third flag word.

1.2.10 Skeleton Outline of a Device Handler

The skeleton outline in Figure 1–1 provides the structure for a simple device handler. In the figure, *SK* is the device name.

Figure 1–1: Skeleton Device Handler

```
        .Title  SK -- Handler Skeleton
;  SK DEVICE HANDLER
        .IDENT  /V05.05/
.SBTTL  PREAMBLE SECTION
        .MCALL  .DRDEF  ; Get handler definitions
        .MCALL  .ASSUME ; Checking macro
        .MCALL  .EXIT   ; To finish run
        .MACRO  ...     ; Define ellipsis (allow
                        ;  ellipsis to assemble)
        .ENDM
                ; Generate nonexecutable handler information tables
                ;  containing the following information:
                ;  Handler is SK
                ;  Handler ID is 350 (user-written handler)
                ;  Handler accepts neither .READ nor .WRITE
                ;  Handler accepts .SPFUN requests
                ;  Device is 1 block in size
                ;  Device has a CSR at 176544
                ;  Device has a vector at 20
        .DRDEF  SK,350,WONLY$!SPFUN$,1,176544,20
                ; Handler has .Fetch and $LOAD code to be executed:
        .DRPTR  FETCH=Fetch,LOAD=Load
                ; Handler is for a "Null" class device
                ; Handler has a data table called DATABL
                ; Data table is of the SKL format
        .DREST  CLASS=DVC.NL,DATA=DATABL,TYPE=SKL
                ; Handler accepts the following SPFUN codes:
                ;  372,376,377
        .DRSPF  <372>,TYPE=T
        .DRSPF  <376>,TYPE=W
        .DRSPF  <377>,TYPE=R
                ; Handler CSR is not to be checked at install,
                ;  but is to be displayed:
        .DRINS -SK
                ; Here is any installation check code
        ...
        RETURN
        .ASSUME . LE 400,MESSAGE=<;Installation area overflow>
```

Figure 1–1 (continued on next page)

Figure 1–1 (Cont.): Skeleton Device Handler

```
                ; Handler accepts SET SK [NO]BONES command:

        .DRSET  BONES,123456,CORPUS,NO

CORPUS:                 ; SET SK BONES
        COM     R3      ; Flip bits
        NOP             ; Pad code
        .ASSUME . EQ CORPUS+4,MESSAGE=<;No option code in wrong place>
NOCORP:                 ; SET SK NOBONES
        MOV     R3,PICKNT       ; Set value in block 1
        RETURN

        .ASSUME . LE 1000,MESSAGE=<;Set area overflow>

.SBTTL  HEADER SECTION

        .DRBEG  SK      ; Handler Queue Manager Entry point
        BR      START   ; Skip data table
DATABL:
        .RAD50  "SKL"   ; Table ID
WRIST:  .BLKW   1       ; Table contents
ANKLE:  .BLKW   1       ; ...
                ;Set up the Vector table:

SK$VTB: .DRVTB  SK,SK$VEC,SKINT,0
        .DRVTB  ,SK$VEC+4,SKINT,1

PICKNT: .BLKW   1       ; Value controlled by Set command
        .ASSUME .-2 LE SKSTRT+1000,MESSAGE=<;Set object not in block 1>

.SBTTL  I/O INITIATION SECTION

START:                  ; Executable Queue code
        ...
        RETURN
.SBTTL INTERRUPT SERVICE SECTION

        .DRAST  SK,4,ABORT      ; Interrupt entry point
        BCS     INT2            ; Interrupt from second vector
        ...
        RETURN
INT2:                           ; Second interrupt vector code
        ...
        RETURN
.SBTTL  I/O COMPLETION SECTION

ABORT:                          ; Abort entry point
        ...
        .DRFIN  SK              ; Completion return

                                ; End of memory resident part of handler

        .DRBOT  SK,ENTRY        ; Boot code

ENTRY:
        ...                     ; Hard boot code to call read routine
        RETURN
```

Figure 1–1 (continued on next page)

Figure 1–1 (Cont.): Skeleton Device Handler

```
READ:
        ...                     ; Read routine
        RETURN
.SBTTL  HANDLER TERMINATION SECTION
        .DREND  SK              ; End of boot code
        .PSECT  SETOVR          ; Suggested block aligned PSect
FETCH:
        ...                     ; Code executed on FETCH
        RETURN
LOAD:
        ...                     ; Code executed on LOAD
        RETURN
RUN:
        ...                     ; Code executed on RUN
        .EXIT
        .END    RUN
```

1.3 Abort Processing

This section describes the behavior of the resident monitor (RMON) and a device handler when a job abort occurs.

The action taken by RMON in abort processing is determined by three criteria:

- The setting of the ABTIO$ and HNDLR$ bits in the device status word (H.STS).
- The action that caused the abort.
- The presence or absence of a current queue element belonging to the aborting job (or job and channel in the case of .ABTIO aborts).

The first two criteria are described in the following sections. Section 1.3.2 contains a table showing the matrix and order of RMON actions based on combinations of all those criteria.

1.3.1 Handler Status Word Bits ABTIO$ and HNDLR$

The combination of ABTIO$ and HNDLR$, whether set or clear, determines to the following extent how RMON performs abort processing for that handler and other handlers that are loaded in memory:

- If ABTIO$ is set, the handler is entered by RMON during any type of abort; the status of HNDLR$ (set or clear) does not matter.
- If ABTIO$ or HNDLR$ is set (but not both), the handler is entered by RMON when a .ABTIO request is issued by a program to any handler.

When a program invokes the .ABTIO request for a channel associated with any handler, RMON calls the abort entry point of all in-memory handlers having that bit combination (ABTIO$ or HNDLR$ set, but not both). RMON checks each handler for I/O requests that might be internally queued on the channel that is specified in the .ABTIO request. RMON performs abort processing for any outstanding I/O request on the channel being aborted by the .ABTIO request. RMON does not discard the current queue element (*dd*CQE) and whether or not it is satisfied is determined by the handler.

If the hanlder aborts the current queue element, it should clear the queue element's completion routine address (Q.COMP) and issue a .DRFIN to return the queue element to the monitor. All outstanding queue elements that are associated with the aborting job or job and channel are removed from the handler's queue element list.

- If HNDLR$ is set and ABTIO$ is clear, RMON does not keep count (in I.IOCT) of the number of outstanding queue elements for that handler.

  Some handlers, such as the distributed RT–11 MQ and Ethernet handlers, can post a request without necessarily expecting satisfaction of that request. To allow such handlers to be aborted, RMON is inhibited from keeping a count (in I.IOCT) of all outstanding I/O requests. Such handlers can then be aborted when they still contain outstanding queue elements.

  Any user-written internally queued handler that can post an I/O request without requiring satisfaction of that request should be built with HNDLR$ set and ABTIO$ clear.

1.3.2 Types of Aborts and Action Taken by RMON

The resident monitor performs abort processing for any of the following actions:

Abort Type	Description
.CHAIN	I/O for the chaining job is allowed to complete.
.EXIT .SRESET	Job I/O is allowed to complete.
.HRESET ?MON-F- <CTRL/C>	Hard error condition. Job I/O is stopped. *?MON-F-* means an abort caused by a fatal monitor error. *<CTRL/C>* means a double CTRL/C typed at the keyboard.
.ABTIO (Handler used by this channel)	A .ABTIO request is issued for a handler that is associated with the aborting job's channel control block.

Abort Type	Description
.ABTIO (All other handlers)	This handler assembled with device status word bit HNDLR$ set and ABTIO$ clear, and is entered whenever a .ABTIO request is called for any handler on any channel.

Table 1–6 illustrates RMON abort processing. It not only shows the actions performed by RMON, but also the order in which they are performed. Before the table is a legend that defines and explains the symbols used in the table.

The order of certain symbols in the tables is important. The symbols show the order of abort processing for the type of abort. A note defines the symbols that should be read in order.

Symbol Definitions and Explanations for Table 1–6

Symbol	Definition/Explanation
Abort Type	The action that caused the abort.
A$=0	The handler is not built with ABTIO$ (ABTIO$=0).
A$=1	The handler is built with ABTIO$ (ABTIO$=1).
H$=0	The handler is not built with HNDLR$ (HNDLR$=0).
H$=1	The handler is built with HNDLR$ (HNDLR$=1).
ddCQE	The handler contains a current queue element belonging to the aborting job (or job and channel if .ABTIO). The absence of this symbol in a header indicates the handler has no current queue element associated with the aborting job (or job and channel if .ABTIO).

NOTE

The order of the following symbols in the tables is important. The symbols show the order of abort processing for the type of abort. For example, the symbols EJ show that operation E is performed first and operation J is performed next.

Symbol	Definition/Explanation
C	RMON removes all queue elements belonging to the job and channel from the queue and decrements I.IOCT one time for each element removed.
C~	RMON removes all queue elements belonging to the job and channel from the queue but does not decrement I.IOCT.
E	RMON calls the handler's abort entry point.

Symbol	Definition/Explanation
J	RMON removes all queue elements belonging to the job from the queue and decrements I.IOCT one time for each element removed.
J~	RMON removes all queue elements belonging to the job from the queue but does not decrement I.IOCT.
Q	RMON waits for all I/O requests for which it expects satisfaction to be satisfied.
S	RMON waits for all I/O requests for which it expects satisfaction to be satisfied and then issues a .ABTIO for every channel associated with the job.
( )	RMON performs abort processing only if there is outstanding I/O on the channel.
–	RMON does not perform abort processing on this handler.

Table 1–6: RMON Abort Processing

Abort Type	A$=0 H$=0	A$=0 H$=0 ddCQE	A$=0 H$=1	A$=0 H$=1 ddCQE	A$=1 H$=0	A$=1 H$=0 ddCQE	A$=1 H$=1	A$=1 H$=1 ddCQE
.CHAIN	S	S	S	S	S	S	S	S
.EXIT .SRESET	Q	Q	QEJ~	QEJ~	QEJ	QEJ	QEJ	QEJ
.HRESET ?MON-F- <CTRL/C>	J	EJ	EJ~	EJ~	EJ	EJ	EJ	EJ
.ABTIO (Handler used by this channel)	(C)	(EC)	(EC~)	(EC~)	(EC)	(EC)	(EC)	(EC)
.ABTIO (All other handlers)	–	–	(EC~)	(EC~)	(EC)	(EC)	–	–

1.4 Handlers That Queue Internally

A device handler can maintain one or more of its own internal queues of outstanding I/O requests instead of using the usual monitor/handler I/O queue. The purpose of maintaining an internal queue is that it permits several operations to take place on the device simultaneously—that is, the handler can service several requests to access the device at once. Internal queuing might also be useful if a handler needs to perform some type of request ordering based on device-specific criteria.

The distributed RT–11 handlers that control communications, XC, XL, NC, NQ, and NU, use internal queuing to process simultaneous input and output requests. See Figure A–3 for a commented source listing of the XL handler for guidance in implementing internal queuing in your handler.

1.4.1 Implementing Internal Queuing

A handler is entered at its .DRBEG code whenever the queue manager places an I/O request queue element on the handler's empty device queue. The handler checks the queue element for validity. An invalid request returns an immediate hard error.

A handler that implements internal queuing decides how to dispose of the current queue element based on whether processing the request requires post-interrupt activity (another interrupt). If the I/O request does not require post-interrupt activity by the handler, the handler processes the queue element immediately and returns, through .DRFIN, to the monitor. If processing the request cannot be immediately satisfied, the handler removes the request queue element from the device queue and places it on an internal queue. The device queue is then available for another request.

The internally queued handler has sole responsibility for managing internally queued queue elements; for moving them between the internal queue and the device queue. The handler is also responsible for returning appropriate queue elements to the monitor because of an abort on a channel or job.

1.4.2 Interrupt Service for Handlers That Queue Internally

When an operation completes, the handler is normally entered at its interrupt entry point, *dd*INT:. After this, various actions are taken depending on the circumstances. If there is more than one internal queue, the handler determines which request this interrupt involves and, therefore, which internal queue. If the operation is not complete, the handler restarts it or continues it and simply returns to the monitor. If the transfer is complete, the handler returns the request to the monitor by using a fake device queue and modified .DRFIN code.

The handler returns the request to the monitor without exiting in order to process any further outstanding requests. The fake device queue is used to avoid any race condition conflict with the monitor over the use of the device queue. The modified form of .DRFIN code uses a CALL rather than a JMP instruction, so that the handler can regain control after the request is returned to the monitor.

The following example illustrates how an internally queued handler returns a queue element to the monitor. In the example, R4 points to the third word of the queue element to be returned.

```
        .
        .
        .
        MOV     R4,ddFCQE          ; Make queue element first
        MOV     R4,ddFLQE          ;  and last on fake device queue
        CLR     Q$LINK(R4)         ; Make sure it doesn't link anywhere
        MOV     PC,R4              ; R4 -> Fake device queue
        ADD     #ddFCQE-.,R4       ;  ...
        MOV     @#$SYPTR,R5        ; R5 -> $RMON
        CALL    @$QCOMP(R5)        ; Return the queue element
        .
        .
        .
; Check the internal queue and start another operation if necessary
        .
        .
        .
        RETURN

; Fake device queue

        .WORD   0                  ; Required
ddFLQE: .BLKW                      ; Fake LQE
ddFCQE: .BLKW                      ; Fake CQE
```

1.4.3 Abort Procedures for Handlers That Queue Internally

As explained in Section 1.3, the contents of the handler status word, H.DSTS, determines how a handler and RMON process aborts. In particular, it is the ABTIO$/HNDLR$ bit combination in the handler status word. There are some particular considerations with abort processing for a handler that internally queues I/O requests:

- Does the handler expect satisfaction of all outstanding I/O requests?

  Setting bit ABTIO$ and not HNDLR$ stops RMON from maintaining the count (I.IOCT) of outstanding I/O requests for the handler.

- Do other handlers in the system need to be notified if the handler processes an abort? Conversely, does the handler need to be notified if other handlers on the system process an abort?

  All in-memory handlers that are built with either ABTIO$ or HNDLR$ set (but not both set) are entered at their abort entry point by RMON whenever a .ABTIO request is issued by a program. Also, RMON checks for internally queued I/O requests on the specified channel. Abort processing is performed on any handler having outstanding I/O requests on the channel being aborted by a .ABTIO request.

  Whether or not the current I/O request (*dd*CQE) is satisfied is determined by the handler code. All other queue elements associated with the job or the job and channel are removed from the handler's queue element list. That is, *dd*LQE and *dd*CQE are set to the same value.

When the handler is entered at the abort entry point, it checks its internal queue for elements belonging to the aborted job. The job number is passed to the handler in R4. Whether the handler aborts all queue elements belonging to that job or only those for a particular channel is determined by the contents of R5. If R5 contains zero, the handler should abort all queue elements assigned to that job. If R5 is nonzero, it points to the first word of a channel control block (the channel status word), and the handler should abort only the queue elements for that channel.

The handler should purge its internal queue of those elements and use the following procedure to reduce the monitor's count of outstanding I/O requests. R0 through R3 must be saved and restored.

1. Remove any internal queue elements that belong to the aborting job or channel. If there are none, simply issue the RETURN instruction.
2. Otherwise, link the removed elements through the element's link word (Q.LINK); the last element's link word must be 0. Set *dd*CQE to point to the last element of this linked list.
3. Clear each aborting queue element's completion routine address (Q.COMP).
4. Issue the .DRFIN macro.

1.5 Set Options

The keyboard monitor SET command permits you to change certain characteristics of a device handler. The handler must exist as a *dd*.SYS file on the system device (*dd*X.SYS for mapped systems), where *dd* is the two-character device name. For example, the following command changes the column width for a printer:

```
SET LP WIDTH=80         (The default is 132 columns)
```

Another type of SET command can enable or disable a function. The following example shows how a SET command can cause the system to send carriage returns to a printer or to refrain from sending them.

```
SET LP CR               (Sends carriage returns; this is the default)
SET LP NOCR             (Does not send carriage returns)
```

Note that you negate the CR option by adding NO to the start of the option. See the *RT–11 Commands Manual* for more information on the SET options available with existing RT–11 device handlers.

A device handler you write can contain code to implement different options. Follow the format outlined in the following sections to learn how to add SET options to your handler. Adding a SET option affects only the handler file; you need not make any changes to the monitor. Note that SET options are valid for both data and system devices.

1.5.1 How the SET Command Executes

The SET command is driven entirely by a table in block 0 of the handler file and by a set of routines, also in block 0, that modify instructions and data in blocks 0 and 1 of the handler. Remember that block 0 refers to addresses 0 through 776, and that the handler header starts in block 1 at location 1000 in the file.

When you type a SET command at the console terminal, the monitor parses the command line and looks for the handler file on the system device. (The type of handler matches the monitor, such as DU.SYS for unmapped monitors or DUX.SYS for mapped monitors.) The handler need not be installed in the running system. The monitor then reads blocks 0 and 1 of the handler into the USR buffer. It scans the table in block 0 until it finds the table entries for the SET option you specified. From the table entry, it can find the particular routine designed to implement that option and the modifiers permitted by that routine, such as NO or a numeric value. The monitor then executes the routine, which contains instructions that modify code in blocks 0 or 1 of the handler. The code in block 1 is part of the body of the handler and contains the instructions for the default settings of all the SET options. After the code is modified, the monitor writes blocks 0 and 1 back out to the system device. Thus, as a result of the SET command, some instructions or data in the handler file are changed. However, any memory-resident copy of the handler is not affected.

1.5.2 SET Table Format

The table for the SET options consists of a series of four-word entries, with one entry per option. The table begins at location 400 in block 0 of the handler and ends an entry with a word zero. Use the .DRSET macro, described below, to generate the table. Examples of overlaid SET code are located in the example handlers in Appendix A.

The first word of the table is a value to be passed in R3 to the SET routine associated with the option when the monitor processes this option. This word can be a numeric value—such as the default column width for a printer—or it can be an instruction to substitute for another instruction in block 1 of the handler. It must not be 0.

The second and third words of the table are the option name in Radix–50, such as WIDTH or CR. In the table, the characters are left justified and filled with spaces.

The low byte of the fourth word is an offset to the routine that performs the code modification. The high byte indicates the type of SET parameter that is valid. Setting the 100 bit shows that a decimal argument is required. A value of 140 shows that an octal argument is required. Setting the 200 bit means that the NO prefix is valid for this option.

Table 1–7 shows a summary of the SET option table.

Table 1–7: SET Option Table

Offset	Name	Meaning
0	DSE.R3	Value to pass in R3 to the SET routine
2–4	DSE.NA	Radix–50 for option name (two words)
6	DSE.SB	Offset to option routine
7	DSE.PA	Parsing option bits:

Bit	Name	Meaning
0–4		Reserved
5	DSE.8	Set means option has octal value Clear means option has decimal value
6	DSE.NU	Numeric value allowed
7	DSE.NO	NO prefix allowed

Offset	Name	Meaning
	DSE.ES	Entry size

1.5.3 .DRSET Macro

Use the .DRSET macro to set up the option table by calling the macro once for each option so that the macro calls appear one after the other. You must invoke the .DRSET macro after .DRDEF and before the .DRBEG macro.

The format for the .DRSET macro is as follows:

.DRSET option,val,rtn[,mode]

option is the name of the SET option, such as WIDTH or CR. The name can be up to six alphanumeric characters long and cannot contain any embedded spaces or tabs.

val is a parameter that will be passed to the routine in R3. It can be a numeric constant, such as the minimum column width, or an entire instruction enclosed in angle brackets to substitute for an existing instruction in block 0 or 1 of the handler. This parameter must not be 0.

rtn is the name of the routine that modifies the code in block 0 or 1 of the handler. The routine must follow the option table in block 0 and not extend above file address 776. If you need more space for SET code, then this lets you overlay the SET code. See the DL example handler in Appendix A.

mode is an optional argument to indicate the type of SET parameter. Enter *NO* to indicate that a NO prefix is valid for the option. Enter *NUM* if a decimal value is required. Enter *OCT* if an octal value is required. Omitting the *mode* argument indicates that the option takes neither a NO prefix nor a numeric argument. You can combine the NO and numeric arguments as follows. The construction <NO,NUM> indicates that both a NO prefix and a decimal value are valid. The construction <NO,OCT> indicates that both a NO prefix and an octal value are valid. Omitting the *mode* argument forces a 0 into the high byte of the last word of the table entry.

See the sections below for examples of the .DRSET macro.

The first .DRSET macro issues an .ASECT directive and sets the location counter to 400 for the start of the table. The macro also generates a zero word for the end of the table. Because the macro leaves the location counter at the end of the table, you should place the routines to modify code immediately after the .DRSET macro calls in your handler. This makes sure that they are located in block 0 of the handler file.

1.5.4 Routines to Modify the Handler

Your handler needs a routine for each SET option. You need only one routine for an option and the NO version of that option. The purpose of the routine is to modify code in the body of the handler based on the SET command typed on the console terminal. One routine can support several SET options. Typically, the value passed in R3 is used to determine which SET option is being performed.

The routines must immediately follow the option table, described above, and they must be located in block 0, after the table and below address 1000. The code in the body of the handler that the routines modify must be in block 1 of the handler, within the first 256_{10} words.

The name of the routine is its default entry point. This is the entry point for options that take a numeric value, for options that take neither a numeric value nor a NO prefix, and for options that accept a NO prefix but do not currently have it. The entry point for options that allow and have a NO prefix is the default entry point + 4.

On entry to the routine, for all options, the carry bit is clear and registers R0, R1, and R3 contain information for use by the routine and R4 and R5 should be preserved. If numeric values are valid for the option, R0 contains the numeric value from the SET command line. R1 contains the unit number specified as part of the device name; if no unit number was specified, the sign bit is set. R3 contains the *val* word of the SET option table (from .DRSET).

The routine can indicate that a command is illegal by returning with the carry bit set. For example, the printer SET WIDTH option does not allow a width less than 30. If the option routine indicates failure, the monitor prints an error message and does not write out blocks 0 and 1. Thus, the check can be made after the block 1 code is modified.

Once you have added the routines for each option to your handler, you can use the following line of code to make sure you are within the size bounds:

```
.IIF GT,<.-1000>, .ERROR .-1000 ; SET code too big!
```

Then you continue with the rest of the handler code, starting with the .DRBEG macro, which implicitly resets the location counter to 1000 and establishes the handler header.

1.5.5 Examples of SET Options

The following examples taken from a printer handler are implementations of SET options.

The examples were chosen to reflect the SET command examples shown at the beginning of this section. The SET commands were as follows:

```
SET LP WIDTH = 80
```

```
SET LP CR
```

```
SET LP NOCR
```

First, the handler invokes the .DRSET macro to set up the option tables for the two options WIDTH and CR.

The first call indicates that the printer WIDTH option is being established, that 30 decimal is a default value of some kind, that O.WIDTH is the routine to process the option, and that it takes a numeric argument.

```
.DRSET WIDTH,30.,O.WIDTH,NUM
```

The next call indicates that the printer CR option is being established, that NOP is to be passed to the routine, that O.CR is the name of the routine to process the option, and that the CR option can take a NO prefix.

```
.DRSET CR,NOP,O.CR,NO
```

The two macro calls generate the following table:

```
.ASECT
. = 400
.WORD   30.                      ;MINIMUM WIDTH
.RAD50  \WIDTH \                 ;OPTION NAME
.BYTE   <O.WIDTH-400>/2
.BYTE   100

.WORD   NOP                      ;INSTRUCTION TO PASS
.RAD50  \CR    \                 ;OPTION NAME
.BYTE   <O.CR-400>/2
.BYTE   200
.WORD   0                        ;END OF TABLE
```

The routines to process these options immediately follow the end of the table. The following examples show the routines. The body of the code in block 1 of the handler that the routines modify is shown at the end of the section.

```
O.WIDTH:MOV     R0,COLCNT               ;MOVE VALUE FROM USER TO
        MOV     R0,RSTC+2               ;TWO CONSTANTS
        CMP     R0,R3                   ;COMPARE NEW VALUE TO
                                        ;MINIMUM WIDTH, 30.
        RTS     PC                      ;RETURN; C BIT SET ON ERROR
```

Note in the example above that the instructions in the routine O.WIDTH change data in two locations in block 1 of the handler.

```
O.CR:   MOV     (PC)+,R3                ;ENTRY POINT FOR "CR"; MOVE
                                        ;ADDRESS OF NEXT LINE TO R3
        MOV     R3,CROPT                ;ENTRY POINT FOR
                                        ;"NOCR" (O.CR+4);
                                        ;MOVE EITHER "NOP" OR
                                        ;PREVIOUS LINE TO CROPT
        BEQ     RSTC-CROPT+.            ;A NEW INSTRUCTION
        RTS     PC                      ;RETURN
```

NOTE

While executing the routines to process a SET option, R4 and R5 are not available for use.

The routine O.CR has two entry points: for the “CR” option, the routine is entered at O.CR; for the “NOCR” option, the routine is entered at O.CR + 4. Note that (1) the routine substitutes one of two instructions for an instruction located in block 1; (2) a NOP instruction is moved to CROPT if the “NOCR” option is selected; (3) if “CR” is selected, the BEQ RSTC-CROPT+. instruction is moved to CROPT.

The construction of the BEQ instruction is necessary because the branch is being assembled into a location other than the one from which it will be executed. In all the routines, a branch instruction must use the following construction to generate the correct address:

```
        BR      A-B+.
```

A is the destination of the branch instruction.

B is the address of the branch instruction.

. is the current location counter.

Generally, only routines for options that accept NO use these branch instructions.

Finally, look at the code in the interrupt service section of the handler that is modified by the routines you have just seen. Remember that the code to be modified must be located in block 1 of the handler, in the first 256_{10} words.

```
        .
        .
COLCNT: .WORD   COLSIZ                  ;# OF PRINTER COLUMNS LEFT
        .
        .
CHRTST: CMPB    R5,#HT                  ;IS CHAR TAB?
        BEQ     TABSET                  ;YES, RESET TAB
        CMPB    R5,#LF                  ;IS IT LINE FEED?
        BEQ     RSTC                    ;YES, RESTORE COLUMN COUNT
        CMPB    R5,#CR                  ;IS IT CARRIAGE RETURN?
CROPT:  NOP                             ;"NOP" IF "NOCR" OPTION;
                                        ;ELSE IF "CR" OPTION, USE
                                        ;"BEQ  RSTC-CROPT+." FROM
                                        ;SET ROUTINES IN BLOCK 0.
        CMPB    R5,#FF                  ;IS IT FORM FEED?
        BNE     IGNORE                  ;NO, IT IS NON-PRINTING
RSTC:   MOV     #COLSIZ,COLCNT          ;RE-INIT COLUMN COUNTER
```

From the examples in the first part of this section, you can see how the routines in block 0 can modify data and instructions in block 1 of the handler.

1.6 Device I/O Timeout

The device timeout feature lets a handler assign a completion routine to be executed if an interrupt does not occur within a specified time interval. Thus, the handler can perform the equivalent of a mark time operation without the need for a .SYNCH call and its attendant potential delay.

Device timeout is supported by all distributed mapped monitors and is an optional feature on unmapped monitors, available through system generation. (Device timeout support requires monitor timer support, which is included on all distributed monitors except SB.) Device timeout is required by the RT–11 multiterminal monitor and support for it is automatically included when you build that monitor.

Within the handler, you select device timeout by including the system conditional TIM$IT=1. RT–11 provides two macros to help you implement device timeout in your handler. The macros, which are described below, are .TIMIO and .CTIMIO. They are available only to device handlers. If you assemble the handler file with the conditional TIM$IT equal to 1, the .DRDEF macro issues a .MCALL directive for the .TIMIO and .CTIMIO macros.

All code in your handler that applies strictly to device timeout support should be placed inside conditional assembly directives. These directives should include the device timeout code if the symbol TIM$IT is 1, and omit it otherwise. This way, the system parameters select whether or not the device timeout code is included in the handler each time you assemble it.

1.6.1 .TIMIO Macro

Use the .TIMIO macro in the handler I/O initiation section to issue the timeout call. You can issue the request anywhere in the handler except at interrupt level. If you need to issue the request at interrupt level, you must issue a .FORK macro call first.

The .TIMIO request schedules a completion routine to run after the specified time interval has elapsed. The completion routine runs in the context of the job indicated in the timer block. In mapped monitor systems, the completion routine executes with kernel mapping, since it is still a part of the interrupt service routine. (See the *RT–11 System Internals Manual* for more information about interrupt service routines and the mapped monitor environment.) As usual with completion routines, R0 and R1 are available for use. When the completion routine is entered, R0 contains the sequence number of the request that timed out.

Because you must go to fork level (and processor priority 0) to issue a .TIMIO or .CTIMIO request at interrupt level, your handler must disable device interrupts before issuing the .FORK, or must be carefully coded to avoid reentrancy problems. Note that you cannot reuse a timer block until either the timer element expires and the completion routine is entered, or the timer element is canceled successfully.

The format of the macro is as follows:

.TIMIO tbk,hi,lo

tbk is the address of the timer block, a seven-word pseudotimer queue element, described below. Note that you must not use a number sign (#) before *tbk*.

hi is a constant specifying the high-order word of a two-word time interval.

lo is a constant specifying the low-order word of a two-word time interval.

The timer block format is shown in Table 1–8.

Table 1–8: Timer Block Format

Offset	Name	Agent	Contents
0	C.HOT	.TIMIO	High-order time word.
2	C.LOT	.TIMIO	Low-order time word.
4	C.LINK	monitor	Link to next queue element; 0 indicates none.
6	C.JNUM	user	Owner's job number; get this from the queue element.
10	C.SEQ	user	Sequence number of timer request. The valid range for sequence numbers is from 177700 through 177377.
12	C.SYS	monitor	–1
14	C.COMP	user	Address of the completion routine to execute if timeout occurs. The monitor zeroes this word when it calls the completion routine, indicating that the timer block is available for reuse.

Although the .TIMIO macro moves the high- and low-order time words to the timer block for you, you must take care to specify them properly in the macro call. Express the time interval in ticks. There are 60_{10} ticks per second if your system is running with 60-cycle power. If your system is running with 50-cycle power, there are 50_{10}

ticks per second. Professional 300 series processors have 60_{10} ticks per second with either line frequency. Time values for 50-cycle power are shown in square brackets ([ ]) immediately after the 60-cycle figure.

The low-order time word accommodates values of up to 65535_{10} ticks. That is equal to about 1092 [1310] seconds, or about 18.2 [21.8] minutes. If you need to specify a time interval of 18.2 [21.8] minutes or less, place a zero in the *hi* argument, and the number of ticks in the *lo* argument to the .TIMIO macro.

If you need to specify a time interval longer than 18.2 [21.8] minutes, think of the high-order word as a carry word. Each interval of 18.2 [21.8] minutes' duration causes a carry of 1 into the high-order word. So, to specify an interval slightly greater than 18.2 [21.8] minutes, supply a 1 to the *hi* argument, and a 0 to the *lo* argument. To specify 36.4 [43.6] minutes, move 2 to the *hi* argument, 0 to the *lo* argument, and so on. Since the 2-word time permits you to indicate up to 65565 units of 18.2 [21.8] minutes each, the largest time interval you can specify is about 2.3 [2.7] years.

The only words of information you must set up yourself in the timer block are the job number, the sequence number, and the address of the completion routine. You can get the job number from the current queue element, and then move it to the timer block. You assign the sequence number yourself. To ensure a unique number, use a value of 177000+dd$COD, where dd$COD is the device identifier code used in the .DRDEF macro at the beginning of the handler. The job number and sequence number are passed to the completion routine when it is entered. You must move the address of the completion routine to the seventh word of the timer block in a position-independent manner.

The .TIMIO macro expands as follows:

```
.TIMIO  tbk,hi,lo

JSR     R5,@$TIMIT              ;POINTER AT END OF HANDLER
.WORD   tbk - .
.WORD   0                       ;CODE FOR .TIMIO
.WORD   hi                      ;HI ORDER TIME INTERVAL
.WORD   lo                      ;LO ORDER TIME INTERVAL
```

1.6.2 .CTIMIO Macro

When the condition the handler was waiting for occurs, you should issue a cancel timeout call, which disables the completion routine. Use the .CTIMIO macro call in your handler to cancel the timeout request. Execution must be in system state when you issue the call. Be sure to issue a .FORK call first if you use .CTIMIO at interrupt level.

For example, a printer handler could check for an off-line condition. When a program requests an I/O transfer, the handler's I/O initiation section forces an immediate interrupt. The handler's interrupt service section then checks the device error bit. If the bit is set, the printer is not on line and the handler prints a message, sets a 2-minute timer with .TIMIO, and returns to the monitor with a RETURN instruction to wait for another interrupt. The device should not interrupt again until the error condition has been fixed by an operator. If no interrupt occurs within two minutes, the timer completion routine prints another error message, sets another 2-minute timer, and returns again to the monitor with RETURN to wait for an interrupt. (See Figure 1–2 for a printer handler example.)

In this example, when an interrupt finally occurs and the error bit is clear, the handler issues the .CTIMIO call to cancel the timed wait.

As another example, a disk handler could set a timer before it starts up a seek operation. When the interrupt occurs, the seek is complete, and the handler should then cancel the timer.

If the time interval in any application has already elapsed and the device has, therefore, timed out, the .CTIMIO request fails. Because the completion routine has already been placed in the queue, the .CTIMIO call returns with the carry bit set. You can usually ignore this condition.

The format of the .CTIMIO macro call is as follows:

.CTIMIO tbk

tbk is the address of the seven-word timer block described above. Note that this time block you specify in the .CTIMIO call must be the same one already used by the corresponding .TIMIO request.

The .CTIMIO macro expands as follows:

```
        .CTIMIO

        JSR     R5,@$TIMIT              ;POINTER AT END OF HANDLER
        .WORD   tbk - .
        .WORD   1                       ;CODE FOR .CTIMIO
```

Note that if a job aborts and your handler is entered at its abort entry point, you must immediately cancel any outstanding timer requests. However, if a timer completion routine has already been entered, you must wait for it to execute.

1.6.3 Device Timeout Applications

Device timeout support is used by RT–11 in only a few instances. However, there are a number of conditions in which timer requests are appropriate. If you are writing a handler for your own device, consider the following sections to determine whether or not timer requests would be useful to you.

1.6.3.1 Multiterminal Service

The resident multiterminal service in RT–11 that supports DZ11 and DZV11 modems uses device timeout to check the status of remote dial-up lines. The bootstrap starts up a polling routine to check each modem for a change in status. If a change occurs, the terminal service takes the appropriate action: it either recognizes a new line or disconnects a line when carrier is lost. Finally, the polling routine issues a .TIMIO call to start a half-second timer. The timer completion routine restarts the polling routine after a half-second elapses.

1.6.3.2 Typical Timer Procedure for a Disk Handler

A disk handler could implement a timer procedure for any disk operation. The purpose of the timer routine is to cancel or restart any operation that takes too long. If an operation does not complete within a reasonable amount of time, chances are good that a disk error of some sort occurred.

The handler's I/O initiation section sets a timer by using the .TIMIO call. Then the handler starts up the operation that a job requested: a read, write, or seek operation. The handler returns to the monitor with a RETURN instruction and waits for a device interrupt.

If an interrupt occurs before the time limit expires, the handler cancels the timer and performs its normal sequence of error checking on the results of the transfer. In general, the handler either drops to fork level to restart an incorrect operation, or exits to the monitor with .DRFIN to remove the current queue element.

If an interrupt does not occur within the time limit, the timer completion routine begins to execute. Its first action should be to simulate an interrupt. This action duplicates the handler environment after a genuine interrupt and makes sure that the stack has the necessary information. Then the timer completion routine acts as though the device interrupted but the transfer was in error. The timer completion routine simply branches to the correct section of code in the interrupt service section of the device handler to finish the processing.

The timer completion routine should use the following instructions to simulate an interrupt and enter system state:

```
MOV     @SP,-(SP)                ;MAKE ROOM ON THE STACK
CLR     2(SP)                    ;FAKE INTERRUPT PS = 0
.MTPS   #340                     ;GO TO PRIORITY 7
.INTEN  0,PIC                    ;ENTER SYSTEM STATE
```

After the handler enters system state, it takes the appropriate action as a result of the timeout. The handler can try the operation again. To do this, it decrements the retry count, drops to fork level, and branches to the I/O initiation section. The code in the initiation section sets another timer, restarts the transfer, and returns to the monitor with a RETURN instruction to await another interrupt.

If the handler decides that the timeout indicates a serious error, one that should not be retried, this same procedure can be followed for a transfer whose retry count is used up. In this case, the handler sets the hard error bit in the Channel Status Word and then exits to the monitor with the .DRFIN call to remove the current queue element.

NOTE

Before a handler goes through the .DRFIN routine to remove the current queue element, it must cancel any timer request that has not yet expired.

1.6.3.3 Printer Handler Example

The extended example shown in Figure 1–2 consists of excerpts from a version of the RT–11 parallel interface printer handler modified to use timer support to check for the device off-line condition.

When the handler's I/O initiation section starts up a transfer, it forces an immediate interrupt, which causes the handler's interrupt service section to check the error bit in the CSR. If there is an error, control passes to the routine OFFLIN, which issues a .SYNCH call to enter user state, prints an error message on the console terminal, and then sets a 2-minute timer. The handler then returns to the monitor with a RETURN instruction and waits for the device to interrupt.

If the device interrupts, it means that the error condition has been corrected by an operator. The handler cancels the timer and checks the error bit once again to make sure there are no problems. If there is no error, the handler proceeds as usual. If there is an error, the handler loops back to the OFFLIN routine. If an interrupt does not occur within two minutes, the timer completion routine begins to execute. It prints an error message, sets another 2-minute timer, and returns to the monitor with a RETURN instruction to await an interrupt.

Figure 1–2: Printer Handler Example

```
; I/O INITIATION SECTION

        .DRBEG  LP
        MOV     LPCQE,R4                ;R4 POINTS TO CURRENT Q ENTRY
        ASL     6(R4)                   ;WORD COUNT TO BYTE COUNT
        BCC     LPERR                   ;A READ REQUEST IS ILLEGAL
        BEQ     LPDONE                  ;SEEKS COMPLETE IMMEDIATELY
RET:    BIS     #100,@LPS               ;CAUSE AN INTERRUPT, STARTING TRANSFER
        RTS     PC

; INTERRUPT SERVICE SECTION

        .ENABL  LSB

        .DRAST  LP,4,LPDONE
        CLR     @LPS                    ;DISABLE INTERRUPTS
        .FORK   FRKBLK
        TST     TICMPL                  ;IS A TIMER ELEMENT ACTIVE?
        BEQ     1$                      ;NO
        .CTIMIO TIMBLK                  ;YES, CANCEL IT
        BCS     1$                      ;ERROR
        CLR     TICMPL                  ;AND DON'T DO IT AGAIN
1$:     MOV     LPCQE,R4                ;R4 POINTS TO CURRENT QUEUE ELEMENT
        TST     @(PC)+                  ;ERROR CONDITION?
LPS:    .WORD   LP$CSR                  ;LINE PRINTER STATUS REGISTER
ERROPT: BMI     OFFLIN                  ;YES, HANG TILL CORRECTED
        .
        .
        .
; I/O COMPLETION SECTION

LPDONE: CLR     @LPS                    ;TURN OFF INTERRUPT
        .DRFIN  LP
        .
        .
        .
```

Figure 1–2 (continued on next page)

Figure 1–2 (Cont.): Printer Handler Example

```
; PRINTER OFF LINE, PRINT WARNING EVERY 2 MINUTES

OFFLIN: MOV     LPCQE,R5                ;POINT TO QUEUE ELEMENT
        MOVB    Q$JNUM(R5),R5           ;GET JOB NUMBER OF CURRENT JOB
        ASR     R5                      ;SHIFT IT
        ASR     R5                      ; RIGHT
        ASR     R5                      ; 3 BITS
        BIC     #^C<16>,R5              ;ISOLATE JOB NUMBER
        MOV     R5,SYJNUM               ;SAVE IT FOR .SYNCH
        MOV     R5,TIJNUM               ;SAVE IT FOR .TIMIO
        .SYNCH  SYNBLK,PIC              ;GO TO USER STATE
        RTS     PC                      ;SYNCH FAILED, PUNT

1$:     CLR     TICMPL                  ;INDICATE THAT WE GOT HERE
        TST     @LPS                    ;IS THERE STILL AN ERROR?
        BPL     2$                      ;NO, QUIT
        MOV     PC,R0                   ;AS COMPLETION ROUTINE, PRINT MESSAGE
        ADD     #MESSAG-.,R0            ;POINT TO MESSAGE AS PIC
        .PRINT                          ;PRINT IT
        MOV     PC,R0                   ;IN A PIC WAY,
        ADD     #1$-.,R0                ; POINT TO TIMIO COMPLETION ROUTINE
        MOV     R0,TICMPL               ;SAVE IT
        .TIMIO  TIMBLK,0,2*60.*60.      ;SET A 2-MINUTE TIMER
        RTS     PC

2$:     BIS     #100,@LPS               ;ENABLE INTERRUPTS
        RTS     PC                      ;RETURN LATER

TIMBLK: .WORD   0                       ;TIMER BLOCK: HI ORDER TIME
        .WORD   0                       ;LO ORDER TIME
        .WORD   0                       ;LINK
TIJNUM: .WORD   0                       ;JOB NUMBER
        .WORD   177700+LP$COD           ;SEQUENCE NUMBER
        .WORD   0                       ;MONITOR PUTS -1 HERE
TICMPL: .WORD   0                       ;ADDRESS OF COMPLETION ROUTINE
SYNBLK: .WORD   0                       ;SYNCH BLOCK
SYJNUM: .WORD   0                       ;JOB NUMBER
        .WORD   0,0,0,-1,0              ;OTHER
.FRKBLK:.BLKW   4                       ;FORK BLOCK
MESSAG: .ASCIZ  /?LP-W-LP off line - please correct/
        .EVEN
        .DREND  LP
```

1.7 Error Logging

Error logging is an optional feature of RT–11 designed to help you monitor the reliability of your system. Device handlers that include support for error logging call the error logger after each I/O transfer. The error logger creates a historical record of the device's I/O activity that you can use to check its reliability.

You must perform a system generation to select error logging. Error logging is supported in all environments. If your system has the capability to run system jobs, the error logger runs as a system job; on FB systems, as an ordinary foreground job; on single-job systems, as a handler.

The system generation conditionals for error logging are as follows:

ERL$G — If this value = 1, it indicates that error logging is enabled for this system.

ERL$S — This condition defines the number of 256-word blocks to use for the internal logging buffer with single job monitors.

ERL$U This represents the maximum number of individual device units for which the error logger collects statistics. The default value is 10, and the absolute maximum number is 30. Each unit adds seven words to the error logger. One slot is required for each unit. (For example, two slots are required for a system with an RK05 with two units.) Your response to a system generation dialogue question establishes the value of this variable.

You should consider your time and memory requirements before deciding to use error logging because error logging creates a certain minimal amount of overhead for each I/O transfer, and the error logger itself uses almost 2K words of memory. However, the error logger does not have to run constantly, so that the memory it requires can be made available to your programs when necessary, and calls that your handler makes to the error logger return immediately. The most efficient way to use the error logging system is as a check when you suspect device reliability problems, which means using it only when necessary.

The following sections describe how to implement error logging in your device handler and what information you should log. They also show you how to add headings for your device to the error reporting program. See the *RT–11 System Utilities Manual* for more information on the entire error logging system and how to use it.

All code in your handler that applies strictly to error logging should be placed inside conditional assembly directives. These directives should include the error logging code if the symbol ERL$G is 1, and omit it otherwise. This way, the system parameters select whether or not the error logging code is included in the handler each time you assemble it.

1.7.1 When and How to Call the Error Logger

A handler calls the error logger after each I/O transfer, whether the transfer was successful or not. If the transfer was in error, the handler calls the error logger once for each retry of the transfer.

Since calls to the error logger must be serialized, the handler can issue them only during I/O initiation or following a .FORK call.

The handler must set up registers before it issues the call to the error logger. The register assignments for the three kinds of calls are described in the following sections.

1.7.1.1 To Log a Successful Transfer

Set up R4 and R5 as described below before calling the error logger after each successful transfer.

R5 must point to the third word (BLKN) of the current queue element.

R4 contains two bytes of information: the high byte is the device-identifier byte, *dd*$COD; the low byte is –1.

1.7.1.2 To Log a Hard Error

Set up R2 through R5 as described below before calling the error logger after a hard error has occurred. Generally, hard errors are those that are not recoverable. Examples of hard errors are device off line or not powered up, device write-locked, and so forth. Further, a soft error that has exhausted its allotted number of retries is considered a hard error.

R5 must point to the third word (BLKN) of the current queue element.

R4 contains two bytes of information: the high byte is the device identifier byte, *dd*$COD; the low byte is 0.

R3 contains two bytes of information: the high byte contains the total number of retries allotted for this transfer; the low byte contains the number of device registers whose contents should appear in the error report.

R2 is a pointer to a buffer in the handler that contains the device registers to be logged.

1.7.1.3 To Log a Soft Error

Set up R2 through R5 as described below before calling the error logger after a soft error has occurred. Generally, soft errors are those that are recoverable and can possibly be corrected by retrying the transfer. Examples of soft errors include timing errors and hardware read or write errors.

Initialize a counter in your handler with the total number of retries allotted for each transfer. Decrement the count as each retry for a soft error is performed. When the count reaches zero, the error logger considers the error to be a hard error. On soft error, the error report prints a separate entry for each retry of a given transfer.

All retries are printed in the report even if the registers are identical. The report does not distinguish between hard or soft immediate errors. It prints only the contents of the registers at the time of the error and the value of the retry count. An immediate hard error can be recognized in the output since it will appear with a retry count of 0 with no immediately previous errors on that device and unit (with a retry count greater than 0).

R5 must point to the third word (BLKN) of the current queue element.

R4 contains two bytes of information: the high byte is the device identifier byte, *dd*$COD; the low byte is the current value of the retry counter. (This value should decrease with each retry until it reaches 0, at which point the error is considered a hard error.)

R3 contains two bytes of information: the high byte contains the total number of retries allotted for this transfer; the low byte contains the number of device registers whose contents should appear in the error report.

R2 is a pointer to a buffer in the handler that contains the device registers to be logged.

1.7.1.4 Differences Between Hard and Soft Errors

The error logger itself does not differentiate between hard and soft errors and records the same information in both cases. However, by examining the report, you can determine if a hard error occurred, because a transfer that has exhausted all its retries will have records in the report for each of these retries, including one with a retry count of 0. It is therefore up to you to interpret the error.

In some circumstances, user-correctable errors, such as device off line or write-locked, should not call the error logger. Usually disk and tape hardware errors are the only ones reported, since these are the errors that reflect device reliability.

1.7.1.5 To Call the Error Logger

Once the required registers are set up, call the error logger as follows:

```
CALL  @$ELPTR
```

$ELPTR contains a pointer into the Resident Monitor. The .DREND macro allocates space in the handler for this pointer. The pointer is filled in at bootstrap time (for the system device) or at .FETCH or LOAD time (for a data device). If the error logger is not running, the monitor returns immediately to the handler. If the error logger is running, a link word in RMON contains its entry point. The following lines of code from RMON show how the call to the error logger is accomplished.

```
$ERLOG: MOV     (PC)+,-(SP)             ;ENTER HERE FROM HANDLER
                                        ;PUSH NEXT WORD ON STACK
$ELHND::.WORD   0                       ;0 IF ERROR LOGGER NOT RUNNING;
                                        ;ELSE CONTAINS ERROR
                                        ;LOGGER ENTRY POINT
        BNE     1$                      ;BRANCH IF LOADED
        TST     (SP)+                   ;PURGE STACK
1$:     RTS     PC                      ;INVOKES ERROR LOGGER OR
                                        ;RETURNS TO HANDLER
```

On return from the error logger call, R0 through R3 are preserved and R4 and R5 are indeterminate.

1.7.2 Error Logging Examples

See the handler listings in Appendix A for examples of error logging.

1.7.3 How to Add a Device to the Reporting Program

After you implement error logging in your device handler, the next step is to modify the reporting system so that the name of your device will appear in the report headings and the registers will be printed properly. The file ERRTXT.MAC contains the information for report headings for the devices supported by the RT–11 error logging reporting utility ERROUT. To include your device, edit this file, reassemble it, and relink it.

Use the following commands to reassemble and relink ERRTXT.MAC:

```
.MACRO/LIST ERRTXT
.LINK ERROUT,ERRTXT
```

ELBLDR Macro

Use the ELBLDR macro to add a new device to the error log reporting system. Edit the file ERRTXT.MAC to add the ELBLDR macro call for your device. The format of the call is as follows:

ELBLDR xx,<type>,C1,C2,<C3>

xx — is the device-identifier byte, *dd*$COD, that you specified in the .DRDEF macro. It must be a value between 0 and 377 octal.

type — is any ASCII string you want to print on the report as the device type. It can be up to 59 characters long. Remember to enclose it in angle brackets.

C1 — is one of the two strings *DISK* or *TAPE*. It identifies the device general classification.

C2 — is the 2-character device name. You must specify exactly two characters.

<*C3*> — is a list of device register mnemonics (minus the first two characters) representing the registers that the handler logs. Separate the mnemonics with commas. Remember to use the angle brackets (<>).

Assembly errors result if you do not specify the parameters to ELBLDR correctly.

None of the parameters for the ELBLDR call is optional.

For example, the ELBLDR call for the RK handler is as follows:

```
ELBLDR 0,<RK11/RK05>,DISK,RK,<DS,ER,CS,WC,BA,DA,DB>
```

This example shows that the device is the RK11/RK05 disk, its 2-character name is RK, its device-identifier byte is 0, and the registers its handler logs are RKDS, RKER, RKCS, RKWC, RKBA, RKDA, and RKDB.

The default input file name for ERROUT is ERRLOG.DAT. However, you can save previous ERRLOG.DAT files by renaming or copying them. Thus, ERROUT can operate on any file with the same format as ERRLOG.DAT. The name is not important; the format is. The internal format of the data in this file is documented in the *RT–11 Volume and File Formats Manual*.

1.8 Special Functions

Handlers use special functions to perform device-specific actions for which there are no corresponding RT–11 programmed requests. Chapter 2 describes those special functions supported by the distributed RT–11 device handlers.

The .SPFUN programmed request initiates special functions. When a program issues a .SPFUN request, it supplies a special function code as one of the arguments. It is the handler's responsibility to process the special function.

1.8.1 .SPFUN Programmed Request

The format of the .SPFUN programmed request is as follows:

.SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

See the *RT–11 System Macro Library Manual* for a description of the .SPFUN programmed request. See Chapter 2 for many special function examples within distributed handlers.

To use special function calls in your handler, you define the interface between the programmed request and the device handler. Thus, the meanings of the *buf*, *wcnt*, and *blk* parameters depend on the particular special function the request invokes; their meaning is dependent on the handler.

Note, however, the following:

- Although the monitor checks to make sure that *buf* is a valid address within the job area, it does not make sure that *buf* plus *wcnt* is still within the job area. It is therefore your responsibility to specify valid values if you use the .SPFUN request to transfer data.
- When using a mapped monitor and therefore a virtual address for *buf*, the buffer address must be mapped before the request is issued. Once the request is issued and the EMT returns, address translation has been performed and the buffer address can be unmapped. In the case of a read (input) operation, if the buffer address is subsequently unmapped, the address must be remapped before data can be accessed from the buffer.
- As previously mentioned, the *buf*, *blk*, and *wcnt* parameters can have any meaning that is supported by the particular handler. You could, therefore, pass an address as an argument.

  Of those parameters, the RT–11 monitor performs address translation for only *buf*. Therefore, if you pass a mapped address in *blk* or *wcnt*, you must not unmap that address while the request is outstanding or active; that is:

  — For nonwait, noncompletion I/O, until a .WAIT request succeeds on the channel.

  — For wait mode I/O, until the request returns.

  — For completion mode I/O, until the completion routine is entered.

If the special function call is to return a single value, *buf* should be a one-word buffer area. You are free to interpret *wcnt* and *blk* as anything you choose. They can be specification words of some sort, pointers to more buffers, and so on, as long as the handler interprets them according to the special function code. Note that the monitor does not alter these values in any way when it passes them to the handler. For example, it does not change the word count from positive to negative.

1.8.2 How to Support Special Functions in a Device Handler

Do the following to implement support for special function calls in your handler:

- Specify SPFUN$ as one of the bits in the .DRDEF *stat* parameter argument. This indicates that the handler can accept special functions.
- Use the .DRSPF macro to list the supported special functions.
- Define symbols in the handler to represent the types of special functions the handler can perform. For example, the DY diskette handler defines the following special function codes:

```
SIZ$FN  = 373                 ;GET DEVICE SIZE
WDD$FN  = 375                 ;WRITE WITH DELETED DATA MARK
WRT$FN  = 376                 ;WRITE ABSOLUTE SECTOR
RED$FN  = 377                 ;READ ABSOLUTE SECTOR
```

Note that all special function codes must be negative byte values (that is, they must be in the range 200 through 377_8). Consult Chapter 2 for those symbols and codes already defined by RT–11. For the sake of consistency across devices, it is advisable to have each special function code represent the same operation on all devices. So, check first to see if a code for your function already exists and use it if it does. If there is no existing code for your particular function, assign codes starting with 200 and work toward 377 from there. (For extended device-unit handlers, the range is 360–377.) This policy should avoid conflicts with future RT–11 codes.

When the handler is entered for an I/O transfer, it should check the fourth word of the queue element to see if this is a request for a special function. Q.FUNC, which is the low byte of the fourth word of the I/O queue element, contains the special function code. On standard I/O requests for read, write, and seek operations, this byte is 0. For special function calls, this value is the negative special function code. Ignore any special function code that is not valid for your device.

If this is a request for a special function, the handler should initiate that function and return with a RETURN instruction. In the interrupt service section the handler should, as usual, check for errors and determine whether the operation is complete. The handler returns either data or words of status information to the calling program in the user buffer.

Since you are implementing the special functions for a particular device, you can establish the calling convention for that function in the .SPFUN programmed request as well as the return convention from the handler. Be sure the handler treats the arguments appropriately for each different special function call.

For a good example of a handler that implements special functions, see the DX handler in Appendix A.

1.8.3 Variable Size Volumes

A handler can control a device that permits volumes with two or more different sizes to be used. Examples of such handlers are the DM handler—which can service both RK06 and RK07 disks through a single controller—and the DY handler—which can service either a single-density or a double-density diskette in a single device unit. A handler for a device that supports volumes of different sizes should pass the size, in blocks, of the smallest volume in the *size* parameter of the .DRDEF macro. This is the value that is returned to a running program when it issues the .DSTAT programmed request.

If it is important that a running program know the size of the volume that is currently mounted, the program can issue a special function to return the volume size. The handler must be able to respond to the request by returning the actual volume size in a one-word buffer area. The handler must have implemented support for special functions, as described above. The standard special function code for returning the actual volume size is 373.

1.8.4 Bad-Block Replacement

If your handler is to support bad-block replacement (BBR) by using a replacement table in the home block, you must implement the BBR special function codes as they are implemented for the DL and DM handlers. See Chapter 2 for more information.

1.9 Devices with Special Directories

The RT–11 monitor can interface to file-structured devices having nonstandard (that is, non-RT–11) directories. Magtapes are an example of special devices. Their handlers set bit 12 (SPECL$) of the device status word. The USR processes directory operations for RT–11 directory-structured devices; for special devices, the handler must process directory operations such as .CLOSE, .DELETE, .LOOKUP, .ENTER, .RENAME, .PURGE, informational (.GFxxx, .SFxxx, and .FPROT), and .CLOSZ, as well as data transfers. See the *RT–11 System Macro Library Manual* for information on those requests.

The monitor requests a special directory operation by placing a positive, nonzero value in the function code byte (Q.FUNC) of the queue element. The positive function codes are standard for all devices. The symbol names are defined in the distributed file, SYSTEM.MLB, and are as follows:

Code	Name	Function
1	CLOS	Close
2	DELE	Delete
3	LOOK	Lookup
4	ENTR	Enter
5	RENM	Rename
7	INFO	.GFxxx, .SFxxx, and .FPROT operations
10	CLOZ	Close with size operation

In a queue element for a special directory operation, word 5 (Q.BUFF) of the queue element contains a pointer to the file descriptor block containing the device name, file name, and file type in Radix–50.

Software errors (such as file not found, or directory full) occurring in special directory device handlers during directory operations are returned to the monitor, processed, and appear in byte 52 as the standard, documented error codes. Hardware errors are returned in the usual manner by setting bit 0 in the Channel Status Word pointed to by the second word of the queue element.

Programmed requests for directory operations to special directory devices are handled by the standard programmed requests. When a .LOOKUP is issued, for example, the monitor checks the device status word for the special device bit. If the device has a special directory structure, the proper function code is inserted into the queue element and the element is directly queued to the handler, bypassing any processing by the USR. Device independence is maintained, since .LOOKUP, .ENTER, .CLOSE, and .DELETE operations are transparent to the user.

For a special device .LOOKUP, the file length is returned in word 6 of the queue element (Q.WCNT). For a .ENTER, word 6 returns the length of the new file.

1.10 Device Handlers in Mapped Systems

Device handlers for unmapped system environments require a few changes to work properly in mapped systems. Before describing the environment for a handler in a mapped system, the following sections outline the nomenclature conventions. The final sections explain how a handler communicates with a user buffer in extended memory.

1.10.1 Naming Conventions and the System Conditional

When you write a device handler, write a common source file called *dd*.MAC, where *dd* is the 2-character device. That source file is then assembled with the correct monitor conditional file such as XM.MAC and the system generation conditional file, such as SYSGEN.CND. This procedure ensures that the system generation features that the handler supports match those of the monitor.

The system generation conditional that represents extended memory support is MMG$T, which has a value of 0 if extended memory support is not selected and a value of 1 if extended memory support is selected. The system conditional MMG$T is correctly set in the distributed monitor conditional files. This means that the extended memory code is only assembled when the value of the conditional MMG$T is 1. The assembly produces *dd*X.OBJ for mapped systems, or *dd*.OBJ for unmapped systems.

All code in your handler that applies strictly to memory management support should be placed inside conditional assembly directives. These directives should include the memory management code if the symbol MMG$T is 1, and omit it otherwise. This way, the system parameters select whether or not the memory management code is included in the handler each time you assemble it.

1.10.2 Mapped Monitor Environment

In a mapped monitor system, at least the handler's root must reside within the low 28K words of physical memory. Typically the entire handler is written to reside in low memory.

The distributed mapped monitors support the .FETCH request, so usually your handler need not be continually loaded in memory. All Digital-supplied handlers for mapped monitors are fetchable with the exception of those few listed in the handler restrictions section of the *RT–11 System Release Notes*.

When handlers are entered, they run with kernel mapping, which permits access to the lower 28K words of memory plus the device I/O page (see Chapter 3 in the *RT–11 System Internals Manual*). The program that requests the I/O transfer, however, need not have the same mapping as kernel mapping. In fact, the program can fall into one of three valid categories:

- A privileged job whose mapping is identical to kernel mapping.
- A privileged job that maps to physical memory addresses above 28K words.
- A virtual job or completely virtual job with any kind of mapping.

Just as RT–11 supplies macros to ease the writing of parts of a device handler, so too does it provide monitor routines that simplify managing mapped systems. RT–11 distributes subroutines that perform the address conversion for you.

The program requesting an I/O transfer supplies a 16-bit virtual buffer address in the programmed request, although that portion of the user's virtual addressing space may be mapped somewhere else in physical memory. The handler must therefore find the actual 18– or 22-bit physical address of the user data buffer before moving information to it or from it. The monitor verifies that the user buffer area occupies contiguous locations in physical memory.

The fact that in a mapped system, locations in physical memory are expressed as 18– or 22-bit addresses, is important when you need to specify an address within the handler itself as a buffer address. If, for example, the handler contains a string of zeroes that it writes to a device as part of initialization, the handler sets up the device write operation, specifying the address of the string in the handler as the buffer address. Since the handler is located within the lower 28K words of physical memory, its physical address can be expressed as its virtual 16-bit address plus extra mapping bits (bits 16 and 17 of an 18-bit address, or bits 16–21 of a 22-bit address), which must be 0.

The *RT–11 System Internals Manual* describes memory mapping in detail.

The RT–11 monitor provides routines for handlers to use to access the real user data buffer in physical memory. The following sections describe these routines and the situations in which they are useful.

1.10.3 Address Translation

RT–11 provides the following two routines for performing address translation for the address passed in Q.BUFF.

- $MPMEM

  Call $MPMEM to return the physical address to be used for MOV operations.

- $MPPHY

  Call $MPPTR (which in turn points to $MPPHY in RMON) to perform address translation for I/O DMA operations.

1.10.3.1 $MPMEM Routine

The $MPMEM subroutine uses queue element offsets Q.MEM (and Q.BUFF) to perform the PAR1 offset mapping.

$MPMEM is located at an address 22(decimal) bytes below the entry address of monitor routine $P1EXT. $P1EXT is pointed to by RMON fixed offset P1$EXT (432).

Before the call, R5 must point to Q.BUFF, the fifth word of the queue element.

On return from the call:

- The first word of the stack, (SP), contains the low-order 16 bits of the physical buffer address.
- The second word of the stack, 2(SP), contains the high-order bits of the physical buffer address. The bit positions for an 18-bit address are 4 and 5; those for a 22-bit address are 4 through 9.

The following code fragment illustrates using $MPMEM. (In code preceding the fragment, R4 was pointed to Q.BLKN, the third word in the queue element.)

```
MOV     @#$SYPTR,R3      ; Get start of RMON
MOV     P1$EXT(R3),R3    ; R3 --> $P1EXT
MOV     R4,R5            ; Make R5 --> 5th word (Q.BUFF) of
CMP     (R5)+,(R5)+      ;  queue element
CALL    $MPMEM(R3)       ; Map KT-11 virtual to physical
MOV     (SP)+,R2         ; R2 = low 16 bits physical address
MOV     (SP)+,R3         ; R3 = high 2 (or 6) bits physical
                         ;  address
        .
        .
        .
```

See also Sections 1.10.6 and 1.10.7.2.

$MPMEM uses Q.MEM rather than Q.PAR because in the case of UMR on UNIBUS processors, the value stored in Q.PAR can diverge from the value stored in Q.MEM.

1.10.3.2 $MPPHY Routine

Call the $MPPHY routine to find the user buffer in physical memory to perform DMA I/O operations. $MPPHY uses the Q.PAR and Q.BUFF queue element offsets to create the correct 18- or 22-bit address for the user buffer.

The format of the call for the $MPPHY routine is as follows:

CALL @$MPPTR

$MPPTR contains a pointer to the $MPPHY routine in the Resident Monitor. The .DREND macro allocates space for this pointer at the end of the handler. The pointer is filled in at bootstrap time (for the system device) or at LOAD time (for a data device).

Before the call:

R5 must point to Q.BUFF, the fifth word in the queue element.

After the call:

(SP), the first word on the stack, contains the low-order 16 bits of the physical buffer address.

2(SP), the second word on the stack, contains the high-order bits of the physical buffer address in bit positions 4 and 5, if it is an 18-bit address, or in bit positions 4 through 9, if it is a 22-bit address.

R5 points to Q.WCNT, the sixth word in the queue element. The value is not changed.

The following example is from the RK handler.

```
        CMP     (R5)+,(R5)+              ;Advance to bufr addr in queue elt
        CALL    @$MPPTR                  ;Convert user virtual addr to physical
        MOV     (SP)+,-(R4)              ;Put low 16 bits in RKBA,
                                         ; High bits on stack
        MOV     (R5)+,-(R4)              ;Put word count into RKWC
        BEQ     7$                       ;0 Count = SEEK
        BMI     5$                       ;Negative = WRITE, So
                                         ; all set up
        NEG     @R4                      ;Positive = READ,
                                         ;Fix count for controller
        MOV     #CSIE!FNREAD!CSGO,R3     ;Function is READ
5$:     BIS     (SP)+,R3                 ;Merge high order address
                                         ; bits into function
        MOV     R3,-(R4)                 ;Start the operation
6$:     RTS     PC                       ;Await interrupt
```

1.10.4 Character Devices: $GETBYT and $PUTBYT Routines

The handlers for character-oriented devices, such as printers, must transfer the data from the device to the user buffer area themselves. The transfer is usually one byte at a time. The device itself uses registers in the I/O page to store one character at a time. The handler can use two monitor routines—$GETBYT and $PUTBYT—to move data between the I/O page and the user buffer area.

1.10.4.1 $GETBYT Routine

A handler can use the $GETBYT monitor routine to move a byte from the user buffer in physical memory to the stack. The handler can then move the character into the device data buffer register in the I/O page and initiate an I/O transfer.

The format of the call for the $GETBYT routine is as follows:

CALL @$GTBYT

$GTBYT contains a pointer to the $GETBYT routine in the Resident Monitor. The .DREND macro allocates space for this pointer at the end of the handler. The pointer is filled in at bootstrap time (for the system device) or at LOAD time (for a data device).

Before the call:

R4 must point to Q.BLKN, the third word in the queue element.

After the call:

(SP), the first word on the stack, contains the next byte from the user buffer in the low byte. The contents of the high byte are not defined.

R4 is unchanged.

The following example from the XL handler shows how the handler gets a byte from the user buffer and outputs it.

```
GNXTCH: MOV     XOCQE,R4                ; R4->current output queue element
        BEQ     10$                     ; None available...
    .
    .
    .
        TST     Q$WCNT(R4)              ; Any characters left to output?
        BEQ     20$                     ; Nope, this request is complete
        INC     Q$WCNT(R4)              ; Yes, now there is one less to do
        CALL    @$GTBYT                 ; Get the byte to output
        MOV     (SP)+,R5
```

The buffer address (Q.BUFF) in the queue element is updated by 1. If a mapping overflow occurs, the monitor routine subtracts 100 from the value in Q.BUFF and adds 1 to the value in Q.PAR and Q.MEM. Mapping overflow occurs if Q.BUFF is 20100 or more.

1.10.4.2 $PUTBYT Routine

After a successful data transfer, a handler can get a character from the device data buffer register in the I/O page and push it onto the stack. It can then use the $PUTBYT monitor routine to move a byte from the stack to the user buffer in physical memory.

The format of the call for the $PUTBYT routine is as follows:

CALL @$PTBYT

$PTBYT contains a pointer to the $PUTBYT routine in the Resident Monitor. The .DREND macro allocates space for this pointer at the end of the handler. The pointer is filled in at bootstrap time (for the system device) or at LOAD time (for a data device).

Before the call:

R4 must point to Q.BLKN, the third word in the queue element.

The byte to transfer to the user buffer must be on the top of the stack. The character must be in the low byte of the stack's first word. The high byte is unpredictable.

After the call:

The word containing the character to transfer is removed from the stack.

R4 is unchanged.

The buffer address (Q.BUFF) in the queue element is updated by 1. If a mapping overflow occurs, the monitor routine subtracts 100 from the value in Q.BUFF and adds 1 to the value in Q.PAR and Q.MEM. Mapping overflow occurs if Q.BUFF is 20100 or more.

The following example from the XL handler shows how the handler gets a character and moves it to the user buffer.

```
30$:
   .
   .
   .
        MOVB    R5,-(SP)                ; Put character here for PUTBYT
        CALL    @$PTBYT                 ; Call the routine
        DEC     Q$WCNT(R4)              ; Is transfer complete? (z-bit=1 if so)
```

1.10.5 Any Device: $PUTWRD Routine

The monitor routine, $PUTWRD, is similar to $PUTBYT, except that $PUTWRD moves a word to the user buffer in physical memory instead of a byte. This routine is useful when the handler needs to transfer a word of status information to the user buffer, rather than a data character from a device. Handlers for any kind of device can use $PUTWRD.

The format of the call for the $PUTWRD routine is as follows:

CALL @$PTWRD

$PTWRD contains a pointer to the $PUTWRD routine in the Resident Monitor. The .DREND macro allocates space for this pointer at the end of the handler. The pointer is filled in at bootstrap time (for the system device) or at LOAD time (for a data device).

Before the call:

R4 must point to Q.BLKN in the queue element.

The word to transfer to the user buffer must be on the top of the stack.

After the call:

The word to transfer is removed from the stack.

R4 is unchanged.

The buffer address (Q.BUFF) in the queue element is updated by 1. If a mapping overflow occurs, the monitor routine subtracts 100 from the value in Q.BUFF and adds 1 to the value in Q.PAR and Q.MEM. Mapping overflow occurs if Q.BUFF is 20100 or more.

The following example from the DY handler shows the handler responding to a special function call that requests the size of the currently mounted volume. In this case, the larger of two possible diskettes is mounted. The handler uses $PUTWRD to move the size of the volume to the user buffer area.

```
        MOV     #DDNBLK,-(SP)           ;Push size in blocks onto stack
        MOV     DYCQE,R4                ;Point R4 to Q.BLKN
        CALL    @$PTWRD                 ;Call the routine
```

1.10.6 Mapping Directly to the User Buffer

Some situations call for combinations of the procedures described in the previous sections. Others require more effort on the handler's part to accomplish a transfer. Some handlers cannot make good use of monitor routines and must access the user buffer directly.

The DM handler for the RK06 disk, for example, normally uses the $MPPHY monitor routine to convert mapped addresses to physical addresses. However, when a Cyclic Redundancy Check (CRC) error occurs, the handler performs its own mapping to the user buffer and then applies the correction for the error before continuing the transfer. The procedure for a handler to map to the user buffer is as follows.

Devices such as the RX01 diskette transfer data one sector at a time between the disk itself and an internal disk data buffer called a silo. Monitor routines for character-oriented devices available to a silo device are too slow for the RX01. So, the handler for the RX01 diskette maps to the user buffer in physical memory and then performs the I/O operation as though it were a simple transfer between memory and the device. The handler implements this mapping by borrowing kernel PAR1.

The handler does this mapping through kernel PAR1. Handlers map to the user buffer through the monitor routine $P1EXT[1].

$P1EXT copies from the handler to the monitor stack the instructions necessary to transfer the data, thereby removing the instructions from possible PAR1 space. $P1EXT next sets the proper PAR1 value and then executes the instructions copied to the stack. When finished, $P1EXT restores PAR1, clears the monitor stack, and returns to the handler at the word following the instruction list. Upon return, all registers are unchanged except as modified by the instruction list.

Call the routine $P1EXT with a JSR R0 followed by a word containing the number of bytes+2 to copy to the monitor stack, a series of instructions to perform the data transfer, and the PAR1 value (Q.PAR) from the queue element. The following instructions from the DX handler illustrate this technique. R1 is the byte count to transfer, R2 points to the user buffer, R4 points to the RX01 CSR, and R5 points to the RX01 data register. P1$EXT is a monitor fixed offset containing a pointer to the routine $P1EXT.

[1] Because all relevant code is executed outside the PAR1 area, the interrupt service in the PAR1 area is handled in mapped monitors by a vector forwarding technique that is transparent to the handler.

```
        MOV     @#SYSPTR,R4             ;R4 -> monitor base
        MOV     P1$EXT(R4),(PC)+         ;Get addr of externalization routine
$P1EXT: .WORD   P1$EXT                   ;Pointer to externalization routine
        .
        .
        .
;--- Remove two lines below if not memory management
        JSR     R0,@$P1EXT              ;Let monitor execute the following code
        .WORD   PARVAL-.                ;Number of bytes + 2 to copy
;---

2$:     TSTB    @R4                     ;Test transfer ready flag
        BPL     2$                      ;Wait till ready
3$:     MOVB    (R2)+,@R5               ;Move a char from user bufr to RX01
        TSTB    @R4                     ;Set CSR for next time
        DECB    R1                      ;Check transfer count
        BNE     2$                      ;If not 0, more to transfer

;--- If memory management, terminate list with PAR1 value
PARVAL: .WORD   0                       ;Remove if not memory management
;---

;Continue with normal processing from here on.
```

The following restrictions apply to the instruction list passed to $P1EXT:

- No instruction in the list can reference any location in the handler, except for relative-address references within the list itself.
- The instruction list can use the stack for temporary storage, but it cannot remove any previous values from the stack or leave any values on the stack after it is done.
- If used in the instruction list, R0 must be saved and restored.
- Instruction lists of more than 32 words are not recommended because of stack space limitations.

If your handler must access the user buffer directly, it is important that you understand how PAR1 maps to the user area. Figure 1–3 shows a virtual job in a typical mapped system with the user buffer located in physical memory above the 28K-word boundary. The user program is mapped to the buffer through PAR6. The handler calls $P1EXT, which borrows kernel PAR1, puts the Q.PAR value from the queue element there, and then uses the Q.BUFF value from the queue element to access the user buffer.

PAR1 maps to physical memory in units of 32-word decimal blocks and at most can map an area 4K words long. (Note that the page length of PDR1 is always set to map the entire page.) If the user buffer starts at a location in physical memory that is not an even multiple of 32 words, PAR1 maps to the first 32-word boundary below the start of the buffer. The PAR1 mapping area can start at any address in physical memory whose low-order two octal digits are 0. Thus, with a particular PAR1 mapping, as much as 4K words or 4K minus 31 decimal words of the user buffer will be mapped. Figure 1–4 shows how this mapping works.

Figure 1–4 shows a buffer area located at 331724 in physical memory with the application program mapped to the buffer through PAR6. The buffer is 24 octal

Figure 1–3: Device Handler Mapping to User Buffer Area

bytes above 331700, which is a 32-word boundary. $P1EXT puts the Q.PAR value, 3317, into PAR1, replacing the default PAR1 value of 0200. This causes PAR1 to map to a 4K-word area in physical memory starting at address 331700. As a result, when the handler refers to kernel virtual addresses in the range 20000 through 37776, it accesses physical memory locations 331700 through 351676. Since the value in Q.BUFF is 20024, by using that value, the handler can access the start of the user buffer area at location 331724.

If the amount of data to be transferred is large, you may need to advance the buffer pointer and adjust the mapping to account for it. There are two ways to advance the buffer pointer. The easier way is to modify PAR1 as you go. For example, for every 32 words you advance through the buffer, add 1 to the PAR1 value and subtract 64 from the offset. The DX handler example just described transfers 64 words at a time, adding 2 to PAR1 (and subtracting 128 from the offset) after each transfer to avoid mapping overflow.

Another way to advance the buffer pointer is to modify the value of Q.BUFF by modifying the value in the queue element itself. To adjust the mapping, step through the following procedures, thinking in terms of 4K-word units. First, after you modify the value of Q.BUFF, compare the new value to 40000. If the value is greater than or equal to 40000, subtract 20000 from it, and add 200 to Q.PAR. These procedures take care of not only adjusting the mapping, but also avoid mapping overflow.

Figure 1–4: PAR1 Mapping

Finally, here are steps to follow to access any location in the user buffer area, if you are given a byte offset from the beginning of the buffer. Essentially, you must determine the number of 32-word units in the offset by dividing the 16-bit byte offset by 100 octal and adding the quotient to PAR1 and the remainder to Q.BUFF. Then you will be able to access the correct location in the buffer.

For example, suppose you needed to access the byte at offset 12345 from the start of the buffer shown in Figure 1–4. Dividing 12345 by 100 yields a quotient of 123 and a remainder of 45. Adding 123 to the current value of Q.PAR, which is 3317, yields 3442 for the new PAR1 value. Adding 45 to the value of Q.BUFF, which is 020024, gives 020071 as the new buffer address. (Note that this is a byte address.)

1.10.7 Extended Memory Subroutines

This section describes a set of subroutines that allow you to perform the following extended memory operations:

- Move data from one place to another in extended memory.
- Obtain a specified amount of memory from the free memory list maintained by RT–11.

- Find a specified global region.
- Convert a user virtual address into a 22-bit physical address.

The entry points for the subroutines that perform these operations are located directly below P1EXT.

1.10.7.1 Converting a Virtual Address into a Physical Address ($JBREL)

The $JBREL subroutine returns the physical address that corresponds to a virtual address for the job number you supply. Your program must be in Kernel mode when it calls $JBREL. If your program is in User mode, use the .CALLK request to transfer control to Kernel mode.

$JBREL is located at an address 26_{10} bytes below the entry address of monitor routine $P1EXT. $P1EXT is pointed to by RMON fixed offset P1$EXT (432).

You supply a job number and virtual address to $JBREL in the following registers:

Register	Contents
R0	The virtual address to be translated into a physical address.
R1	The job number, addressing mode, and space-type (instruction or data) for which the virtual address applies. You can determine the job's number from the .GTJB request. R1 contains the following information, none of which is validated for accuracy:

Bits	Value	Meaning
0	0	Reserved
1–3	0–7	Job Number
4–7	0	Reserved
8–9		Addressing mode:
	00	User
	01	Supervisor
	10	Reserved
	11	Reserved
10		Address space:
	0	Data space (if enabled)
	1	Instruction space
11–15	0	Reserved

Register	Contents
R3	The size in 32-word chunks.

$JBREL passes the job number and virtual address to the monitor. The monitor performs the address translation and returns to $JBREL. If the specified virtual address is not mapped to a virtual job, the equivalent kernel-mapped address is returned.

On return, if the carry bit is clear, $JBREL provides the following information:

Register	Contents
R1	The PAR1 relocation bias.
R2	The PAR1 displacement.
R3	The amount of contiguously mapped memory that begins at the returned PAR1 bias and displacement, in 32-word chunks. If the value returned is less than that specified in R3 as input to $JBREL. The V-bit (overflow) is set; otherwise it is cleared.

If carry is set on return, R1 and R2 contain random data.

The following example code assumes you are running in User mode and, therefore, require the .CALLK request to transfer control to virtual mapping in Kernel mode:

```
        .MCALL  .CALLK, .PRINT, .EXIT
        .LIBRARY "SYSTEM.MLB"
        .MCALL  .SYCDF, .FIXDF, .P1XDF
        .NLIST  BEX

        MMG$T   =: 1
                                        ; Define system logicals:
        .SYCDF                          ;  $SYPTR - base of fixed area
        .FIXDF                          ;  P1$EXT - offset of $P1EXT
        .P1XDF                          ;  $CJVPT - routine offset from $P1EXT

        VIRTAD  =: 0                    ; Virtual address to be translated
        JOBNUM  =: 16                   ; Job number of virtual address

START:  MOV     #VIRTAD,R0              ; Virtual address to translate
        MOV     #JOBNUM,R1              ; Job number for translation
                                        ;  virtual address is user mode
                                        ;  and data space (if enabled)
        MOV     #5,R3                   ; Check that 5 64-byte chunks
                                        ;  are contiguously mapped
        MOV     @#$SYPTR,R2             ; R2 = RMON Base
        MOV     P1$EXT(R2),-(SP)        ; Stack pointer to $P1EXT routine
        ADD     #$CJVPT,@SP             ; Make it point to $JBREL for .CALLK
        .CALLK                          ; Enter KERNEL mode
                                        ;  Execute $JBREL
                                        ;   Return to USER mode
        BCS     10$                     ; Branch if error occurred
        BVS     20$                     ; Branch if less than 5 64-byte
                                        ;  chunks are contiguously mapped
        MOV     R1,PAR1BS               ; Store returned PAR1 value
        MOV     R2,PAR1OF               ; Store returned PAR1 offset
        BR      DONE                    ; Branch to program exit

10$:    .PRINT  #ERROR1                 ; Report the error
        BR      DONE                    ; Branch to program exit

20$:    .PRINT  #ERROR2                 ; Report the error
DONE:   .EXIT                           ; Done with example

PAR1BS: .WORD   0                       ; Physical address's PAR1 value
PAR1OF: .WORD   0                       ; Physical address's PAR1 offset
ERROR1: .ASCIZ  /Error: Check for invalid job number./
ERROR2: .ASCII  /Error: Not all of requested address block is /
        .ASCIZ  /contiguously mapped./
```

```
        .END    START
```

1.10.7.2 Moving Data Within Extended Memory ($BLKMV)

The $BLKMV subroutine moves the contents of memory from one place in 22-bit physical memory to another. The entry point is $P1EXT-2.

In the following example, R0 contains the address of $P1EXT, and BLKMOV equals –2. $BLKMV moves the data from the specified input buffer to the specified output buffer.

```
MOV     #input_buffer_par1,R1
MOV     #input_buffer_par1offset,R2
MOV     #output_buffer_par1,R3
MOV     #output_buffer_par1offset,R4
MOV     #word_count,R5
CALL    BLKMOV(R0)
```

1.10.7.3 Obtaining Free Memory (XALLOC)

The XALLOC subroutine obtains a specified amount of memory from the free memory list maintained by RT–11. The size argument passed in R2 is in units of 32_{10} words. To allocate 32000_{10} words, specify 1000. as the size passed to R2. The entry point for the subroutine is $P1EXT-6.

In the following example, R0 contains the address of $P1EXT, and XALLOC equals –6.

```
MOV     #required_size,R2
CALL    XALLOC(R0)
```

If the required amount of memory is not available, the carry bit will be set on return. In this event, R2 contains the size of the largest amount available.

If the required amount of memory is available, the carry bit will be reset on return. In this event, the memory has been removed from the free list, and R1 contains the region address divided by 32_{10}.

XALLOC uses R3 and destroys the contents of this register.

1.10.7.4 Returning Memory to the Free List (XDEALC)

The XDEALC subroutine returns a specified section of extended memory to the free memory list maintained by RT–11. The entry point for XDEALC is $P1EXT–18$_{10}$. $P1EXT is pointed to by RMON fixed offset P1$EXT (432).

The address and size of the section of extended memory to be returned are specified in units of 32_{10} words. Load R1 with the starting address divided by 32_{10} and R2 with the size of the region in units of 32_{10} words.

In the following example, R0 contains the address of $P1EXT, and $XDEPT is –$18_{10}$.

```
MOV     #region_address,R1      ; Address in units of 32. words
MOV     #region_size,R2         ; Size in units of 32. words
CALL    $XDEPT(R0)
```

On return from XDEALC, the carry bit is clear if the memory was returned. If the carry bit is set, the memory was not returned because the free memory has become too fragmented.

XDEALC destroys the contents of R1 and R2. If you want to preserve the contents of those registers across the call, you must save them.

1.10.7.5 Finding a Global Region (FINDGR)

The FINDGR subroutine finds a global region that has a specific name. The entry point for this subroutine is $P1EXT–10.

In the following example, R4 contains the address of $P1EXT, and FINDGR equals $–10_{10}$.

```
        MOV     #rad50_name_area,R5
        CALL    FINDGR(R4)
```

where *rad50_name_area* is the address of a 2-word area containing the RAD50 name of the region to search for.

If the specified region is found, the carry bit is clear on return. In this event, R1 points to the size word of the associated global region control block.

If no region by the specified name is found, the carry bit is set on return. In this event, R1 points to the size word of the next available global region control block.

If no more global region control blocks are available, R1 is returned with a zero value.

1.10.7.6 Converting a Virtual Address into a Physical Address ($USRPH)

The $USRPH subroutine converts a user virtual address in the current running job into a 22-bit physical address.

NOTE

> No job number is specified. Ensuring that the current running job is also the job for which the address translation is intended is quite difficult. Therefore, unless you have a very good reason for using this routine, Digital recommends you instead use the $JBREL routine, for which you can specify the job number.

The entry point for this subroutine is $P1EXT–14_{10}$.

In the following example, R5 contains the address of $P1EXT, and CVAPHY equals –14.

```
        MOV    #virtual_address,R0
        CALL   CVAPHY(R5)
```

On return, R1 will contain the high-order address bits, and R2 will contain the low-order address bits.

1.11 System Device Handlers and Bootstraps

In these sections, a description of monitor files precedes an explanation of how to create a system device handler or modify an existing handler to use as a system device. Within the main body of this explanation, details are given on the primary driver and on various bootstrap routines. The final sections provide background information on the DUP procedures for bootstrapping a new system device.

1.11.1 Monitor Files

A monitor file must reside on your system device and can have any name you choose, but its required file type is .SYS. If you create a monitor through the system generation process, its name is RT11*xx*.SYG. You must rename the monitor to .SYS before you use it.

Blocks 1 through 4 of each monitor file contain the secondary bootstrap. The secondary bootstrap loads the system device handler and the monitor into memory. It also modifies the monitor tables to connect the monitor with the device handler and assigns the default DK and SY names.

Each device handler that can be used as a system device handler has a special block of device-specific code in it called the *primary driver* that is used by the secondary bootstrap to read the system device handler file and the monitor file from the system device. The secondary bootstrap has room in its own block 0 to store the primary driver.

1.11.2 Creating a System Device Handler

To create a system device handler, you must add the primary driver to a standard handler for a data device. As described in the following sections, the .DRBOT macro does much of that work for you.

1.11.2.1 .DRBOT Macro

Use the .DRBOT macro to help you set up the primary driver. It also invokes the .DREND macro to mark the end of the handler so that the primary driver will not be loaded into memory during normal operations. In general, the code in the primary driver does not have to be Position-Independent. However, any non-PIC reference must be expressed relative to *dd*BOOT::. Note also that locations 60_8 through 116_8 are not available for your use.

The format for the .DRBOT macro is as follows:

.DRBOT name,entry,read

name is the 2-character device name.

entry is the entry point of the software bootstrap routine.

read is the entry point of the bootstrap read routine.

The .DRBOT macro puts a pointer to the start of the primary driver into location 62 of the handler file. It puts the length, in bytes, of the primary driver into location 64. The primary driver, including the error routine supplied by .DREND, must not exceed 1000_8 bytes. Location 66 contains the offset from the start of the primary driver to the start of the bootstrap read routine.

Issue the .DRBOT macro call before the .DREND macro call. Then put the primary driver code between .DRBOT and .DREND, remembering that the primary driver must be one block or less in size—that is, it must be 1000_8 bytes long or less, including the error routine and the locations from 60_8 through 116_8. The .DREND macro is called twice in a system device handler: once by .DRBOT, and once when you use it at the very end of the primary driver. The first occurrence of .DREND closes out the nonsystem section of the device handler and sets up a table of pointers into the monitor, among other things. The second .DREND call, the one you issue yourself, creates the BIOERR bootstrap error routine, instead of repeating the pointer table.

If you use the BOOT command to bootstrap the new device, DUP passes the system unit number to the primary driver in location 4722 and in R0. If you bootstrap the device with a hardware bootstrap or some non-RT–11 utility program, the primary driver must determine the device unit number that was booted and save it in location 4722 and in R0.

1.11.2.2 Primary Driver

The primary driver you add to a standard handler for a data device consists of four parts:

- Entry routine
- Software bootstrap
- Bootstrap read routine
- Bootstrap error routine

The primary driver works together with the RT–11 bootstrap, BSTRAP, to boot the new system device. The primary driver is contained entirely within the p-sect *dd*BOOT, where *dd* is the 2-character device name. The code is loaded and executes, beginning at location 0 in physical memory.

For examples of the primary driver, see the handler listings in Appendix A.

1.11.2.3 Entry Routine

The entry point for the primary driver is *dd*BOOT::. This location must contain only two instructions, and these must follow the Digital standard bootstrap sequence. These instructions are a NOP and a branch to the start of the software bootstrap. If the start of the software bootstrap is too far away for a branch, you can branch to a JMP instruction that starts the software bootstrap. The entry routine for the RK handler is as follows (BOOT1 is defined in the primary driver):

```
RKBOOT:: NOP
        BR      BOOT1
```

Any hardware bootstrap causes the code in p-sect *dd*BOOT to load into memory at location 0. It also starts execution at *dd*BOOT::.

1.11.2.4 Software Bootstrap

The DUP utility executes the software bootstrap as the result of a jump or branch from the entry routine. Upon entry, all registers are available for use in the software bootstrap. The software bootstrap performs the following functions in the order shown:

1. Sets up the stack at location 10000.
2. Saves the number of the device unit from which the system was just bootstrapped. The method you use to find the unit number varies depending on the device; some unit numbers are passed in R0, and others must be extracted from the CSR. Save the unit number on the stack, and elsewhere in memory, if necessary.
3. Calls the bootstrap read routine to read in the rest of the bootstrap.
4. Puts a pointer in B$READ to the bootstrap read routine.
5. Puts the Radix–50 value for “B$DNAM” in B$DEVN.
6. Stores the device unit number in B$DEVU.
7. Jumps to B$BOOT in RT–11’s bootstrap to continue.

The software bootstrap should be located in the primary driver immediately below location *dd*BOOT + 664. (Locations 664 through 776 contain the error routine created by .DREND.)

1.11.2.5 Bootstrap Read Routine

The purpose of the bootstrap read routine (the primary bootstrap) is to read the volume in the device unit from which the system was just bootstrapped. It is called by both the RT–11 bootstrap (BSTRAP, the secondary bootstrap) and by DUP (the software boostrap), as described in the previous section.

The interface through which the other routines pass information to the bootstrap read routine is as follows:

R0 contains the block number to read.

R1 contains the word count to read.

R2 contains the memory buffer address into which to store the data.

All registers are available for use in the bootstrap read routine, as is the stack.

The bootstrap read routine normally is a noninterrupt routine, used to read the volume according to the parameters passed in R0 through R2. On error, the routine should jump to BIOERR. If there are no errors, it should return with a RETURN instruction, with the carry bit clear.

The bootstrap read routine should be located in your primary driver at location *dd*BOOT + 120. (Location 120 is the lowest address at which the read routine can be located.)

1.11.2.6 Bootstrap Error Routine

The bootstrap error routine starts at location BIOERR::. The code in this routine is supplied completely by the .DREND macro, which you place at the end of the primary driver.

1.11.3 DUP and the Bootstrap Process

This section shows how DUP carries out three commands related to bootstrapping. The commands are as follows:

```
BOOT ddn:filnam
COPY/BOOT xxn:filnam ddm:
BOOT ddn:
```

1.11.3.1 BOOT ddn:filnam

Use the *BOOT ddn:filnam* command to perform a software bootstrap of a specific monitor file on a specific device. In the command line, *dd* represents the 2-character device name; *n* is its unit number. Both the new monitor file and the new device handler must be present on device *dd*.

As soon as this command is issued, DUP first checks that device *dd* is a random-access device. Next, it locates the monitor file *filnam*.SYS on the device. (The .SYS file type is both the default and the required file type.) Then DUP reads blocks 1 through 4 into a memory buffer. These blocks contain the secondary bootstrap for the monitor.

The next-to-last word in block 4 contains the suffix for the handlers associated with this monitor. DUP uses this to build the file name of the device handler, usually *dd*.SYS or *dd*X.SYS. DUP reads block 0 of the device handler file into a memory buffer, using the contents of locations 62 and 64 to locate the primary driver, and reads it into a memory buffer.

Next, DUP copies the primary driver into a buffer at the beginning of the secondary bootstrap, which is also in a memory buffer. It loads the information shown in Table 1–9 for the primary driver and the secondary bootstrap.

Table 1–9: DUP Information

Offset from Start of Memory Buffer	Contents
4722	Booted unit number (B$DEVU)
4724–4726	Booted file name in Radix–50 (B$DNAM)
5000	Date at which booted
5002–5004	Time at which booted

DUP then copies the primary driver and secondary bootstrap from the memory buffer into memory locations 0 through 5004. Then it jumps to location 1000 to start the secondary bootstrap at its DUP entry point so that the secondary bootstrap can load the monitor and the system device handler into memory.

Figure 1–5 illustrates the procedure.

Figure 1–5: BOOT ddn:filnam Procedure

1.11.3.2 COPY/BOOT xxn:filnam ddm:

Use the *COPY/BOOT xxn:filnam ddm:* to copy the secondary bootstrap from the monitor file on device *xx* to blocks 2, 3, 4, and 5 of device *dd*. In the command line, *xx* represents the device on which the monitor file is stored; *n* is its unit number; *dd* represents the 2-character name of the device that is to receive the bootstrap; *m* is its unit number.

As soon as this command is issued, DUP checks that devices *xx* and *dd* are random-access devices. Next, it locates the monitor file *filnam*.SYS on the *xxn:* device. It reads blocks 1 through 4 into a memory buffer. These blocks contain the secondary bootstrap for the monitor.

DUP locates the appropriate handler file on device *dd*. DUP then reads block 0 of the device handler file into a memory buffer, using the contents of locations 62 and 64 to locate the primary driver, and reads it into a memory buffer.

The handler for the system device *dd* must already be located on *dd* before you can copy the bootstrap to the device. DUP loads two words of Radix–50 for *filnam* into locations 4724 and 4726 of the memory buffer. Next, DUP copies the primary driver into block 0 of device *dd*. Finally, DUP writes the secondary bootstrap to blocks 2 through 5 of device *dd*.

Figure 1–6 illustrates the procedure.

Figure 1–6: COPY/BOOT xxn:filnam ddm: Procedure

1.11.3.3 BOOT ddn:

Use the *BOOT ddn:* command to perform a software bootstrap of a specific device that already has a specific monitor secondary bootstrap in blocks 2, 3, 4, and 5 (placed there by the COPY/BOOT command). In the command line, *dd* represents the 2-character name of the device to be booted; *n* is its unit number. Both the new monitor file and the new device handler must be present on device *dd*.

As soon as this command is issued, DUP first checks that device *dd* is a random-access device. Then it reads blocks 2, 3, 4, and 5 into a memory buffer. These blocks contain the secondary bootstrap for the monitor. The primary driver is already in locations 0 through 776.

DUP locates the appropriate handler file on device *dd*. This procedure is a check that the volume has a system device handler stored on it so that it can be validly bootstrapped.

DUP then extracts the file name of the monitor file from locations 724 and 726 of block 4 and locates the monitor file on the device to make sure that it really exists.

Next, DUP loads the information shown in Table 1–10 for the primary driver and the secondary bootstrap.

Table 1–10: DUP Information

Offset from Start of Memory Buffer	Contents
4722	Booted unit number
5000	Date booted
5002–5004	Time booted

DUP then copies the primary driver and secondary bootstrap from the device into memory locations 0 through 4777. Then it jumps to location 1000 to start the secondary bootstrap at its DUP entry point so that the secondary bootstrap can load the monitor and the system device handler into memory.

If the /FOREIGN option is used, DUP reads in block 0 and jumps to location 0.

Figure 1–7 illustrates the procedure.

1.12 Including Support for Multiterminal Handler Hooks

Including handler hooks support in a multiterminal monitor and in your handler lets the handler use any serial line on the system. The distributed LS and XL handler source files contain conditionalized support for multiterminal handler hooks. In this section, the XL handler is used to provide example code. A copy of the XL handler with extended comments is located in Appendix A.

This section provides information on including support for multiterminal handler hooks in your handler. Chapter four in the *RT–11 System Internals Manual* contains a section that describes how the monitor supports such handlers. You should read that section before you read this one, as that section describes the basic monitor /handler protocol. It also describes the monitor data structures that your handler writes and accesses and the interrupt service routines your handler uses to read and write data.

Figure 1–7: BOOT ddn: Procedure

Support for multiterminal handler hooks should be included in at least the following places. Each item is described in detail with example code.

- Installation code following .DRINS, Section 1.12.1
- Set code for the supported SET command conditions at .DRSET, Section 1.12.2
- Establish the monitor hooks at installation or LOAD/FETCH code, Section 1.12.3
- Handler hook interrupt processing during execution of interrupt service code, Section 1.12.4
- Remove handler hooks connection with the monitor at UNLOAD/RELEASE code Section 1.12.5

1.12.1 Installation Support

The handler does the following at installation:

- Determines if the handler should use the handler hooks monitor support.

  If not required, the handler can install for nonmultiterminal support.

  If required but not available, the handler refuses to install.

- Assuming the proper conditions are met, the handler accepts the installation.

The following code is from the installation section of the XL handler source. R0 contains the contents of RMON fixed offset 54, $SYPTR, and $THKPT has been defined as 472:

```
        TSTB    I$MTTY                  ;Are handler hooks needed?
        BEQ     20$                     ;Nope...
        TST     $THKPT(R0)              ;Yes, is the support available?
        BEQ     40$                     ;Nope, reject the installation
        BR      30$                     ;Yes, nothing to do until fetch/load

20$:    .
        .
        .
        .
30$:    TST     (PC)+                   ;Accept the installation (carry=0)
40$:    SEC                             ;Reject the installation (carry=1)
        RETURN
        .
        .
        .
I$MTTY: .BYTE   -1                      ; : Install-time 'hooks required' flag
        .BYTE                           ;reserved
```

1.12.2 SET Command Support

Two SET command conditions should be supported by a handler that has been built with hooks support:

- SET dd LINE=n

  Support for this condition is included so that the handler can change the serial line to which it will attach. The default line number can be established during system generation.

- SET dd [NO]MTTY

  Support for this condition is included when a handler is built to support both a standard interface and the multiterminal monitor hooks. In such a case, by default the handler assumes connection to the standard interface until the command is issued.

  When the MTTY condition is specified, the handler should clear the installation CSR (found in handler file image 176). The handler should also clear the vector information in the handler header (handler file image offset 1002). The original contents of these words can be built into words elsewhere in the handler from which they can be restored when the NOMTTY condition is specified.

The code to support those SET command conditions for the XL handler follows:

The following is in block 0, following the installation code:

```
I$MTTY: .BYTE   -1                      ; : Install-time 'hooks required' flag
        .BYTE                           ;reserved
VECSAV: .WORD   100000+<<XL$VTB-H1.VEC>/2-1> ; : Vector info for SET NOMTTY
CSRSAV: .WORD   XL$CSR                  ; : CSR info for SET NOMTTY
        .
        .
        .
.DRSET  LINE    16.                     O.LINE  NUM     ;LINE=n
.DRSET  MTTY    -1                      O.MTTY  NO      ;[NO]MTTY
        .
        .
        .
; SET XL LINE=line_number

O.LINE: CMPB    R0,R3                   ;Is line number valid?
        BHI     O.ERR                   ;Nope...
        MOVB    R0,O$LINE               ;Yes, set line number to use
        BR      O.NOR

; SET XL [NO]MTTY

O.MTTY: BR      10$                     ;Entry point for MTTY
        NOP                             ;placekeeper
        CLR     R0                      ;Entry point for NOMTTY
        MOV     CSRSAV,INSCSR           ;Nope, restore install-time CSR
        MOV     VECSAV,H1.VEC           ; and vector information
        BR      20$

10$:    CLR     INSCSR                  ;Reset install-time CSR and
        CLR     H1.VEC                  ; vector so handler installs
20$:    MOVB    R0,O$MTTY               ;Set/Reset MTTY hooks use flag
        MOVB    R0,I$MTTY               ; and inform install code of setting
        BR      O.NOR
        .
        .
        .
O.NOR:  TST     (PC)+                   ;Success (carry=0)
O.ERR:  SEC                             ;Failure (carry=1)
        RETURN
        .
        .
        .
```

The following is in the executable portion of the handler (block 1 and beyond):

```
; *** SET ***
O$MTTY: .BYTE   -1                      ;Default to hooks used
.Assume <O$MTTY-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>

; *** SET ***
O$LINE: .BYTE   XL$LUN                  ;Default line to use
.Assume <O$LINE-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>
```

1.12.3 Establish Hooks Connection with Monitor

The handler establishes hooks connection with the monitor at the LOAD/FETCH code. The code should do the following:

- Determine if handler hooks are required and if not, proceed with nonmultiterminal hooks code (connect to standard interface) so long as the CSR and vector do not conflict with any TCB in the multiterminal configuration.

  The handler installation code should determine if support exists in the monitor for handler hooks. Therefore, because the handler is installed, support for handler hooks can be assumed.

- From RMON fixed offset $THKPT, the handler should access the monitor data structure THOOKS and store the addresses of the hooks support routines in the in-memory image of the handler.
- Conduct the following tests:
  1. Determine which serial line is to be used and verify its validity.

     Compare the requested line number with the maximum supported in THOOKS.
  2. Determine if the line is available.

     From THOOKS data, access the TCB for the line and determine if the T.CSR word exists (showing the interface is present on the system) and if the value is correct.
  3. Verify that the line is not the console line.

     Check the CONSL$ bit in the T.STAT word of the TCB.
  4. Verify that the line is not already owned.

     Check that the T.OWNR word of the TCB is zero.
- If the tests above are passed, the handler should determine its physical name and place it in the handler at the word just before the handler hooks routine.
- If the tests above are not passed, the handler should report a LOAD/FETCH error by setting the PSW carry bit and return.
- The handler then performs the following operations in the indicated order (to avoid any race condition):
  1. Store the address of the TCB to which it is attached in memory.
  2. Place the address of its handler hooks entry point in the T.OWNR word of the TCB.

     That address must reside in the low 28K-words of memory in Kernel mode and Instruction address space.
  3. Set the HANMT$ bit in the T.STAT word of the TCB.
  4. If you handler needs to monitor modem control signals, set the HANMC$ bit in the T.STAT word of the TCB. Otherwise, modem control is handled by the multiterminal monitor as described in the remote terminal section of the *RT–11 System Internals Manual*

The following code from the XL handler source illustrates connection between the handler and the monitor.

```
;+
;
; LOAD
;       This routine is entered on FETCH or LOAD of the XL handler
;       and is used 1) to verify use of the handler in the specific
;       configuration and, if needed, 2) to establish the required
;       connections between the handler and the interrupt service of
;       a monitor with support for multiterminal handler hooks.
;
;-

        .ENABL  LSB

FETCH::
LOAD::
        MOV     R5,ENTRY$               ; Save entry point
        MOV     R2,SLOT$                ;  and table size
        MOV     @R5,R5                  ; R5 -> Base of handler (in memory)
        MOV     @#$SYPTR,R0             ; R0 -> Base of RMON
        TSTB    <O$MTTY-XLLQE>(R5)      ; Terminal hooks to be used?
        BEQ     20$                     ; Then use normal DL
        MOV     $THKPT(R0),R1           ; R1 -> Multiterminal handler hooks
                                        ;  data structure in RMON
        BEQ     60$                     ; Monitor doesn't have the support...
        TSTB    (R1)+                   ; Bypass structure size byte
        MOVB    (R1)+,R2                ; R2 = Number of LUNs on system
        MOV     (R1)+,R3                ; R3 -> TCB list
        MOV     (R1)+,<MTOENX-XLLQE>(R5) ; Set pointer to output enable routine
        MOV     (R1)+,<MTYBRX-XLLQE>(R5) ; Set pointer to Break control routine
        MOV     (R1)+,<MTYCTX-XLLQE>(R5) ; Set pointer to Control routine
        MOV     (R1)+,<MTYSTX-XLLQE>(R5) ; Set pointer to Status routine
        MOVB    <O$LINE-XLLQE>(R5),R0    ; R0 = Line to attach to
        BMI     60$                      ; Must be a positive number
        CMPB    R0,R2                    ; Is line in this configuration?
        BGE     60$                      ; Nope, invalid line number
        ASL     R0                       ; Shift for word offset into TCB list
        ADD     R0,R3                    ; R3 -> TCB list entry
        MOV     @R3,R3                   ; R3 -> TCB for LUN
        TST     T.CSR(R3)                ; Is the line present in hardware?
        BEQ     60$                      ; Nope...
        TST     T.STAT(R3)               ; Is the line a console?

        .Assume CONSL$ EQ 100000
        BMI     60$                      ; Yes...
        MOV     R5,R0                    ; R0 -> Handler hook routine
        ADD     #<XLHOOK-XLLQE>,R0       ; ...
        TST     T.OWNR(R3)               ; Is the line already attached?
        BEQ     10$                      ; Nope...
        CMP     R0,T.OWNR(R3)            ; Yes, to this handler?
        BNE     60$                      ; Nope...
10$:    MOV     ENTRY$,R1                ; R1 -> $ENTRY entry
        SUB     SLOT$,R1                 ; R1 -> $PNAME ENTRY
        MOV     @R1,-2(R0)               ; Inform handler of its physical name,
        MOV     R3,<TCBADX-XLLQE>(R5)    ; link the handler to the TCB
        BIS     #<HANMT$!HANMC$>,T.STAT(R3) ; declare line owned by handler
                                         ; and that handler will process modem,
        MOV     R0,T.OWNR(R3)            ; finally link the TCB to the handler
        BR      50$
         .ENDC ;NE XL$MTY

20$:    BIT     #MTTY$,$SYSGE(R0)        ; Is this a multiterminal monitor?
        BEQ     50$                      ; Nope, then there can't be a conflict
        .ADDR   #MTAREA,R0               ; R0 -> .MTSTAT EMT area
        .ADDR   #MTSTAT,R1               ; R1 -> Status block
        .MTSTA  R0,R1                    ; Get info about multiterminal system
        BCS     60$                      ; Errors?
        MOV     @#$SYPTR,R0              ; R0 -> $RMON
        MOV     MTSTAT,R1                ; R1 -> First TCB in system
        ADD     R0,R1                    ; ...
        MOV     MTSTAT+MST.LU,R2         ; R2 = Highest LUN on the system
                                         ; (Number_of_LUNs - 1)
30$:    TST     T.CSR(R1)                ; Is this a configured line?
        BEQ     40$                      ; Nope...
        CMP     <XIS-XLLQE>(R5),T.CSR(R1) ; Will use of the CSR conflict?
        BEQ     60$                      ; Yes, reject the load
        CMP     <XL$VTB-XLLQE>(R5),T.VEC(R1) ; Will use of the VECTOR conflict?
        BEQ     60$                      ; Yes, reject the load
40$:    ADD     MTSTAT+MST.ST,R1         ; On to next TCB
        DEC     R2                       ; More TCB's to check?
```

```
        BGE     30$                     ; Yep...
        .BR     50$                     ; Nope, use of interface won't conflict

50$:    TST     (PC)+                   ; Success return
60$:    SEC                             ; Error return
        RETURN

ENTRY$: .BLKW                          ; : -> $ENTRY table entry
SLOT$:  .BLKW                          ; : Size of a monitor handler table

MTAREA: .BLKW   3                      ; : EMT area for .MTSTAT
MTSTAT: .BLKW   8.                     ; : Status block from .MTSTAT
```

1.12.4 Handler Hook Interrupt Processing

The handler hook interrupt entry point is called by the monitor whenever an interrupt occurs on the line to which the handler is attached.

When an input interrupt occurs, the monitor calls the handler hook entry point with the character in R5 and the TH.PIC function code in R0. The handler processes the character, preserving the registers, and returns.

When an output interrupt occurs, the monitor calls the handler hook entry point with the TH.GOC function code in R0. The handler returns the next output character in R5 and the PS carry bit is clear. If the handler has no character for output, it returns the PS carry bit set. All registers are preserved.

The multiterminal interrupt service controls character output. A handler cannot send output directly to an interface, but must instead indicate it has output by calling the MTOENB routine.

The following code from the XL handler source file illustrates the process.

```
; The following byte indicates whether the handler should make use
; of the multiterminal hooks during FETCH/LOAD and during operation.

; *** SET ***
O$MTTY: .BYTE   -1                     ;Default to hooks used
.Assume <O$MTTY-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>

; *** SET ***
O$LINE: .BYTE   XL$LUN                 ;Default line to use
.Assume <O$LINE-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>

ISPND:  .BYTE   -1                     ; : Input suspend flag
OSPND:  .BYTE   -1                     ; : Output suspend flag
        .EVEN
        .
        .
        .
;+
;
; XLHOOK
;       Entered from multiterminal input or output interrupt service.
;
; Call (TH.GOC):
;       R0 = Function code
;
; Return (TH.GOC):
;       PSW<C> = 0, R5 = character
;       PSW<C> = 1, no character available
;
; Call (TH.PIC):
;       R0 = Function code
;       R5 = character
;
;-

        .ENABL  LSB
```

```
XLPNAM: .Rad50  /XL/                    ; : Rad50 physical name
                                        ; loaded by FETCH/LOAD code

XLHOOK:
        .Assume <XLHOOK-XLPNAM> EQ 2 MESSAGE=<XLPNAM must preceed XLHOOK>

        MOV     R4,-(SP)                ;Save register for awhile

; Function code = 1 = TH.GOC
;       (Get Output Character)

        CMP     R0,#TH.GOC              ;'Get Output Character' request?
        BNE     10$                     ;Nope...
        TSTB    OSPND                   ;Is output suspended?
        BNE     20$                     ;Yep...
        CALL    HOINT                   ;Yes, hook handler output service
        BR      30$

; Function code = 2 = TH.PIC
;       (Put Input Character)

10$:    CMP     R0,#TH.PIC              ;'Put Input Character' request?
        BNE     20$                     ;Nope...
        TSTB    ISPND                   ;Is input suspended?
        BNE     20$                     ;Yep...
        CALL    HIINT                   ;Yes, hook handler input service
        BR      30$

20$:    SEC                             ;Return failure
30$:    MOV     (SP)+,R4                ;*C* Restore previously saved register
        RETURN
```

1.12.5 Remove Handler Hooks Connection to Monitor at UNLOAD/RELEASE

Upon UNLOAD or RELEASE, the handler should perform the following operations in the indicated order (to avoid any race conditions):

1. Clear the HANMT$ and HANMC$ bits in T.STAT in the TCB.
2. Clear the T.OWNR word of the TCB.

The following code from the XL source file illustrates the procedure:

```
UNLOAD::
        MOV     @R5,R5                  ; R5 -> Handler entry point (XLLQE)
        TST     <STATFG-XLLQE>(R5)      ; Is handler in use?
        BNE     10$                     ; Nope, it can be unloaded...
        MOV     <XICQE-XLLQE>(R5),-(SP) ; Check internal queues
        BIS     <XOCQE-XLLQE>(R5),(SP)+ ;  ...
        BEQ     RELEAS                  ; They're empty...
        .ADDR   #NOUNLO,R0              ; R0 -> Error message string
                                        ;  (KMON reports error)
        SEC                             ; Indicate error
        RETURN                          ;  and return to KMON

RELEAS::
        MOV     @R5,R5                  ; R5 -> Handler entry point (XLLQE)
10$:
        TSTB    <O$MTTY-XLLQE>(R5)      ; Terminal hooks in use?
        BEQ     20$                     ; Nope...
        MOV     <TCBADX-XLLQE>(R5),R1   ; R1 -> TCB we're hooked to
        BEQ     30$                     ; We're not...
        CALL    <DISINI-XLLQE>(R5)      ; Disable input
        CALL    <DISOUI-XLLQE>(R5)      ;  and output interrupts
        CLR     R0                      ; Deassert all modem control bits
        CALL    <SETSTT-XLLQE>(R5)      ;  ...
        CLR     T.OWNR(R1)              ; Disconnect TCB from handler
        BIC     #<HANMT$!HANMC$>,T.STAT(R1) ; ...
        BR      30$

20$:                                    ; Perform UNLOAD/RELEASE operations
                                        ;  for a nonhooked handler
        .
        .
        .
30$:    CLC
        RETURN
```

```
NOUNLO: .NLCSI  TYPE=I,PART=PREFIX
        .ASCIZ  "F-Handler may not be unloaded while in use"
```

1.13 Including Extended Device-Unit Support

When modifying a user-written handler to enable extended device-unit support, you should be aware of how an extended-unit handler interacts with two RMON tables ($OWNER and $PNAM2) and the functions specific to an extended-unit handler in the following:

- .DRDEF and .DRBEG macros
- LOAD/FETCH routine
- UNLOAD/RELEASE routine
- Q.FUNC byte of an I/O queue element

See the *RT–11 System Internals Manual* for a description of the $PNAM2 table. See the following sections for a description of the changes to the $OWNER table, the macros, routines, and byte.

1.13.1 .DRDEF and .DRBEG Macros

The .DRDEF and .DRBEG macros generate required code for the preamble of a device handler. Specify the UNIT64=YES parameter to the .DRDEF macro to place the 1-letter extended-unit handler name and ownership table in blocks 0 and 1 of the handler.

The format for calling .DRDEF is:

```
.DRDEF name,code,stat,size,csr,vec,[UNIT64=YES]
```

Note the *name* parameter to the .DRDEF call. The macro .DRDEF uses the first letter of the name parameter as the 1-letter physical device name of an extended-unit device.

The name parameter defines both the dd$NAM constant (the traditional 2-letter physical device name) and the dd$PN2 constant (the 1-letter device name).

The macro .DRDEF places the RAD50 representation of the 1-letter device name followed by two blanks in offset H.64UM (100_8) of block 0 of an extended-unit handler. It also places the location of the extended-ownership table in offset H.UNIT (76_8) of block 0 of the handler and indicates generation of the table in the last 32-bytes of memory-resident code. (If the monitor under which the handler is running does not support extended device units, those last 32 bytes are not loaded into memory.)

.DREND creates the extended-ownership table in the memory resident portion of the handler because UNIT64=YES was specified in the previous call to the .DRDEF macro. The extended-ownership table (dd$U64) is always 16_{10} words (64_{10} nibbles) long.

1.13.2 LOAD/FETCH and UNLOAD/RELEASE Routines

You place the new LOAD/FETCH and UNLOAD/RELEASE routines in the extended-unit handler. You place those routines in the handler SETOVR PSECT and order the PSECTs in the handler source code such that SETOVR is the last. You then link the handler as illustrated in the following example command:

```
.LINK/NOBITMAP/EXE:BIN:SPX.SYG/BOUNDARY:512. OBJ:SPX
Boundary?  SETOVR
.
```

1.13.2.1 LOAD/FETCH Routine

If the running RT–11 monitor has extended device-unit and device ownership support, then the LOAD/FETCH routine:

1. Places a pointer to the handler's extended-ownership table in the second word of the handler's entry in the monitor's $OWNER table.
2. Sets the first word of the handler's entry in the $OWNER table to a value of 2. This value, and any nonzero even value in the $OWNER unit 0 nibble, is a flag (the $XUNIT flag) indicating both that the handler supports up to 64 units and that the second word of the handler's entry in the $OWNER table points to a separate list holding the $OWNER nibbles. The $XUNIT flag is filled in at bootstrap/install time.

   The definition of a nibble (4 contiguous bits) in the $OWNER table for a nonextended-unit device handler is that its value is either 0 or the job number + 1. Therefore, an $OWNER nibble of a nonextended-unit device handler is always 0 or odd, since job numbers are always even.

   Figure 1–8 shows the handler entry in the $OWNER table pointing to the extended-ownership table in the handler.

1.13.2.2 UNLOAD/RELEASE Routine

If extended-unit support is enabled in the running RT–11 monitor, the UNLOAD /RELEASE code of an extended-unit handler clears the second word of the handler's entry in the monitor's $OWNER table, since the extended-ownership table (along with the handler itself) is being removed from memory.

1.13.2.3 Example LOAD/FETCH and UNLOAD/RELEASE Routines

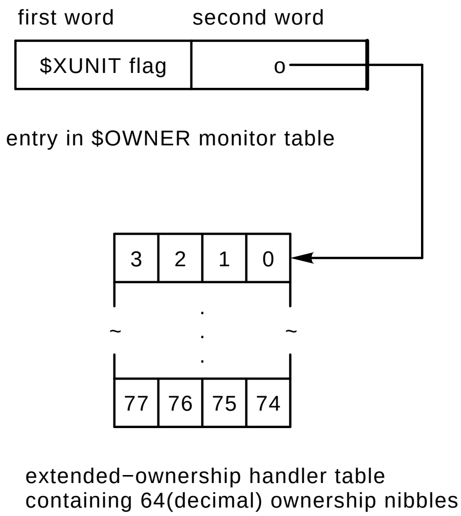
The following example LOAD/FETCH and UNLOAD/RELEASE routines would be appropriate for extended device-unit handers:

```
.IF     NE,dd$N64

        .PSECT  SETOVR

.SBTTL  LOAD  - Load/Fetch code for extended device-unit handler
```

Figure 1–8: Relationship of $OWNER Table to Extended-Ownership Table




```
;+
; Example LOAD/FETCH routine for a extended device-unit Handler.
;
; INPUT
;
;       R0              -> handler routine being called
;       R1              -> GETVEC routine
;       R2              $SLOT*2
;       R3              type code
;                       0 -- .FETCH
;                       2 -- .RELEASE
;                       4 -- $LOAD
;                       6 -- $UNLOAD
;                       10-- Job Abort
;                       12-- BSTRAP
;       R4              -> read routine
;       R5              -> $ENTRY word as above
;
;
; BSTRAP or KMON INSTALL modifies $PNAME, $PNAM2, and $OWNER+0
; for an extended device-unit handler.  You need to insert only the
; address of the extended-ownership table into $OWNER+2 here.
;
;
; OUTPUT
;       Registers need not be saved by the handler code
;       Carry clear, unless an error was detected by
;       $SYS or the handler code.
;       If an I/O error occurred, R0 is cleared and Carry set.
;       If the handler returns with Carry set, R0 is passed,
;       as it was returned by the handler.
;-
        .LIBRARY "SRC:SYSTEM"   ;Indicates SYSTEM.MLB

        .MCALL  .CF3DF          ; CF3.64 definition
        .MCALL  .FIXDF          ; $CNFG3 and $PNPTR definitions
        .MCALL  .SYCDF          ; $SYPTR definition

        .CF3DF
        .FIXDF
        .SYCDF
```

```
LOAD:
FETCH:
        MOV     @#$SYPTR,R0              ; R0 -> Base of RMON
        BIT     #CF3.64,$CNFG3(R0)       ; Extended unit support in monitor?
        BEQ     10$                      ; Branch if not, done.
        BIT     #CF3.OW,$CNFG3(R0)       ; Owner table support in monitor?
        BEQ     10$                      ; Branch if not, done.
20$:    MOV     R2,R3                    ; R3 = $SLOT*2
        ASL     R3                       ; *4
        ASL     R3                       ; *8
        ADD     R2,R3                    ; R3 = $SLOT*10
        CALL    FIXOWN                   ; Insert extended ownership table addr
                                         ;  into $OWNER word #2.
                                         ; R1 -> $OWNER+2
10$:    CLC
        RETURN                           ; Done.

.SbTtl  UNLOAD  - Unload/release code for a extended device-unit handler

;+
; Example UNLOAD/RELEASE routine for a extended device-unit Handler.
;
; INPUT
;
;       R0              -> handler routine being called
;       R1              -> GETVEC routine
;       R2              $SLOT*2
;       R3              type code
;                       0 -- .FETCH
;                       2 -- .RELEASE
;                       4 -- $LOAD
;                       6 -- $UNLOAD
;                       10-- Job Abort
;                       12-- BSTRAP
;       R4              -> read routine
;       R5              -> $ENTRY word as above
;
;
; This routine should zero the $OWNER+2 pointer to the extended ownership
; table of an extended device-unit handler.
;
; OUTPUT
;       Registers need not be saved by the handler code
;       Carry clear, unless an error was detected by $SYS or the handler code
;       If an I/O error occurred, R0 will be cleared and Carry set.
;       If the handler returns with carry set, R0 will be passed
;       as it was returned by the handler.
;-

RELEASE:
UNLOAD:
        MOV     @#SYPTR,R0               ; R0 -> base of RMON
        BIT     #CF3.64,$CNFG3(R0)       ; extended device-unit support
                                         ;  in monitor?
        BEQ     10$                      ; Branch if not
        BIT     #CF3.0W,$CNFG3(R0)       ; Owner table support in monitor?
        BEQ     10$
        MOV     R2,R3                    ; R3 = $SLOT*2
        ASL     R3                       ; *4
        ASL     R3                       ; *8
        ADD     R2,R3                    ; R3 = $SLOT*10.
        CALL    FIXOWN                   ; R1 -> $OWNER+2
        CLR     @R1
10$:    CLC
        RETURN

.SBTTL  FIXOWN  - insert pointer to extended ownership table into $OWNER
```

```
;+
; FIXOWN - insert pointer to extended ownership table into second word
; of $OWNER table (64 UNITS ONLY!!!)
;
; INPUT
;       R2 = $SLOT*2
;       R3 = $SLOT*10.
;       R5 -> $ENTRY entry for this handler
;       dd$X64: extended ownership table
;
; OUTPUT
;       $OWNER+2 points to extended ownership table
;       R1 points to $OWNER+2
;-

FIXOWN: MOV     @#SYPTR,R1              ; R1 -> $RMON
        MOV     $PNPTR(R1),-(SP)
        ADD     R1,@SP                  ; @SP -> beginning of $PNAME
        ADD     R2,@SP                  ; @SP -> beginning of $ENTRY
        MOV     R5,R1                   ; R1 -> $ENTRY entry for this handler
        SUB     (SP)+,R1                ; R1 = byte offset into $ENTRY
        ADD     R5,R1                   ; R1 = $ENTRY + double-word index
        SUB     R3,R1                   ; R1 -> $OWNER of this handler + 8.
        CMP     -(R1),-(R1)             ; R1 -> $OWNER of this handler + 4
        MOV     @R5,-(R1)               ; move addr of ddLQE into $OWNER+2
        ADD     #dd$X64-ddLQE,@R1       ; make $OWNER (pic) to point to
                                        ; extended ownership table
        RETURN

.ENDC ;NE dd$N64
```

1.13.3 Q.FUNC Definition

The Q.FUNC byte of an I/O queue element passed to an extended-unit handler is different from the Q.FUNC byte of an 8-unit handler. However, the Q.FUNC byte passed to an 8-unit handler is unchanged to allow upward compatibility and to allow extended-unit handlers to function properly for units 0-7, when extended-unit support is not included in the running RT–11 monitor.

Q.FUNC is the low byte of the fourth word of the I/O queue element passed to a handler in an I/O request. Q.FUNC contains the special function code and the high 3 bits of the handler unit number.

The following diagram shows the bit layout of the Q.FUNC byte for an extended-unit handler:

T	N	U	M	F	U	N	C

T means the TYPE of I/O request.
NUM means the UNIT NUMBER.
FUNC means the FUNCTION.

The I/O request can be one of two types:

- On standard I/O requests or requests for special directory operations, the T bit is 0. In this case:
  - NUM is the high 3 bits of the handler unit number.
  - FUNC is a value 0000 through 1111. (The value 0000 specifies a read, write, or seek operation; 0001 through 1111 specifies a special directory operation.)

- On special function (SPFUN) I/O requests, the T bit is 1. In this case:
  - NUM is one's complement of the high 3 bits of the handler unit number.
  - FUNC is a value 0000 through 1111 that specifies an SPFUN operation from SPFUN 360 (0000) to SPFUN 377 (1111).

1.13.4 Programmed Requests of Extended-Unit Handlers

You must modify programs that assemble device specifications from physical device names and unit numbers for those programs to support extended-unit handlers. You can do this in conjunction with use of the .CSTAT programmed request, which reports the device on which a file is located.

For an extended-unit handler, .CSTAT returns the 2-letter device name from the $PNAME table if the device unit specified falls in the 0-7 range. If the device unit specified is greater than 7, .CSTAT returns the 1-letter device name found in the new $PNAM2 table.

1.14 How to Assemble, Link, and Install a Device Handler

Assembling, linking, and installing a new device handler are simple procedures described in detail in the following sections.

1.14.1 Assembling a Device Handler

The command you use to assemble your handler can include the following elements:

- Your MACRO–11 source file should be named *dd*.MAC, where *dd* is the 2-character device name.
- You can use the /SHOW:MEB assembler option to print the expansions of macros such as .DRBEG and .DRAST in the assembly listing.
- Each monitor has a corresponding conditional source file, such as XM.MAC, which defines the basic features of that monitor.
- SYSGEN.CND is the default name of the SYSGEN conditional file and is a product of the system generation process. Omit this file if you are assembling a device handler that will run with a distributed RT–11 monitor.

  If your handler is to be used with a monitor that was produced through the system generation process, you must use the SYSGEN conditional file with which you assembled that monitor so that the handler conditionals will match the monitor conditionals and the handler will operate in the correct environment. You can specify the name of the SYSGEN conditional file by requesting an answer file during the SYSGEN process as the .CND file takes the file name of the answer file.
- If you have used symbol names from the distributed system library SYSTEM.MLB, you should assemble your handler with that library. The default device for SYSTEM.MLB is SRC, so you should assign SRC to that device on which SYSTEM.MLB resides and include SRC in the full file specification (SRC:SYSTEM.MLB).

Include the line `.LIBRARY "SRC:SYSTEM.MLB"` early in your program to call that library.

To assemble a handler for an unmapped system, use the following command where *mon* is the distributed monitor conditional source file:

```
.MACRO/CROSSREFERENCE/SHOW:MEB/LIST mon+SYSGEN.CND+SYSTEM.MLB/LIBRARY+dd/OBJECT
```

To assemble a handler for a mapped system, use the following command, where *mon* is the distributed monitor conditional source file:

```
.MACRO/CROSSREFERENCE/SHOW:MEB/LIST mon+SYSGEN.CND+SYSTEM.MLB/LIBRARY+dd/OBJECT:ddX
```

1.14.2 Linking a Device Handler

Once your source file assembles without errors, you are ready to link it. To link a handler for an unmapped system, use the following command:

```
.LINK/NOBITMAP/EXECUTE:dd.SYS  dd
```

To link a handler for a mapped system, use the following command:

```
.LINK/NOBITMAP/EXECUTE:ddX.SYS  ddX
```

If the handler requires block alignment of some code, use the following command where *nnn* is the block alignment boundary for PSECT *psect*:

```
.LINK/NOBITMAP/EXECTURE:ddX.SYS/BOUNDARY:nnn ddX
psect
```

1.14.3 Installing a Device Handler

Before you can use your new handler, you must inform the monitor that the handler is present and you want it installed. Add the monitor information about it to the monitor device tables described in Chapter 2 of the *RT–11 System Internals Manual*. The process of adding a new device is called installation. There are two separate routines in the RT–11 system that can install a device handler: the bootstrap and the monitor INSTALL command. Both routines require a device's hardware to be present on the system before they install the device handler. (Section 1.14.3.6 describes a way to circumvent this restriction if you need to install a handler for a nonexistent device.)

The following sections describe the various ways to install device handlers in an RT–11 system.

1.14.3.1 Using the Bootstrap to Install Handlers Automatically

The bootstrap routine first locates the system device handler on the device from which you booted the system and installs it. Then it scans the rest of the handler files on the system device and tries to install the corresponding handler for each hardware device it finds on the system. If the hardware is not present, the bootstrap does not install the device.

The only difficulty with this procedure occurs when there are more handler files than device slots. A distributed monitor reserves one device slot for each device RT–11 supports. A monitor you create through system generation reserves one slot for each device you request. In addition, it provides the number of empty slots you specify.

A slot is considered to be reserved for a particular device if the $PNAME monitor table has an entry for that device. A slot is empty if $PNAME has a zero word.

The automatic device installation routine in the bootstrap has a set of priorities to determine which handlers to install when there are more handlers than slots. If all slots are empty, the bootstrap installs the system device handler plus the first handlers it encounters on the system device whose device hardware is present. For example, if a system has eight slots, all empty, the bootstrap installs the system device handler and the first seven legitimate handlers it finds on the system device.

If one or more slots are reserved for specific devices (that is, the devices have entries in the $PNAME table), the bootstrap reserves those slots for the corresponding handlers until it can verify the presence of the appropriate hardware. If the hardware exists, the bootstrap installs its device handler. If the hardware is not present, the bootstrap clears its $PNAME entry, thus creating an empty slot.

Figure 1–9 summarizes the algorithm the bootstrap uses to install device handlers.

As you can see, handlers with entries in the $PNAME table have higher priority at boot time. If the handler file is on the system device and the device hardware exists, the bootstrap always installs the handler.

When you write a device handler yourself, you should have no problem installing it in your RT–11 system because you can rely on the bootstrap to install the handler for you if the handler resides on the system device, if its hardware is present, and if there is an empty slot in the monitor tables. If your system has no free slot, you can create one or more by simply storing fewer device handler files on your system device and rebooting the system. You can also use the monitor INSTALL command (described in Section 1.14.3.2) to install a new handler without rebooting the system. (This new handler may be one that the bootstrap could not install due to lack of free slots, or it may be a new handler that you just created or just copied to the system device.) Or, if you created your system through system generation, you can use the DEV macro (described in Section 1.14.3.3) to reserve a slot for a new device handler and give it priority for installation at bootstrap time. Figure 1–10 summarizes the ways you can install a new device handler.

1.14.3.2 Using the INSTALL Command to Install Handlers Manually

Before using the INSTALL command to install a handler manually, use the SHOW command to see if there are any empty device slots on your system. If there are none, use the REMOVE command to remove a device you do not need and make room for your new device, which you then add by using the INSTALL command. The formats of these commands are documented in the *RT–11 Commands Manual*.

If a device slot was already available, your device will install automatically the next time you bootstrap the system. If you used REMOVE and INSTALL to add your new device to the system, you must reissue the commands after each bootstrap. To

Figure 1–9: Bootstrap Algorithm for Installing Device Handlers

Figure 1–10: Installing a New Device Handler

install the new device automatically at each bootstrap, put REMOVE and INSTALL commands in your system's startup indirect file. This saves you the trouble of typing the commands yourself. In addition, it gives the device the appearance of being permanently installed.

1.14.3.3 Using the DEV Macro to Aid Automatic Installation

If you created your system through a system generation, you can edit a system MACRO–11 source file to add a new device to the $PNAME table, thus giving it preference in the automatic handler installation procedure. The file you edit is SYSGEN.TBL, one of the files you assemble to create a monitor file.

Use the DEV macro in the file SYSGEN.TBL to add a new device to the system permanently. The format of the DEV macro is as follows:

DEV name,s

name is the 2-character device name.

s represents the device status word (leave this argument blank).

The following examples are taken from the SYSGEN.TBL file:

```
DEV     RK                ;INSTALLS THE RK DISK
DEV     LP                ;INSTALLS THE LINE PRINTER
DEV     MT                ;INSTALLS MAGTAPE
```

After you edit SYSGEN.TBL to add the DEV macro call for your device, you must reassemble it. Use the following command:

```
.MACRO/OBJECT:TBxx  mon+SYSGEN.CND+SYSGEN.TBL
```

xx represents the monitor type, such as SB, FB, XM, or another of the mapped monitors.

mon represents the monitor conditional source file, such as XM.MAC, which defines the basic features of that monitor. Once the assembly is complete, relink the object files to create your new monitor. Follow the commands in the command file that resulted from your system generation procedure to build the modified system.

1.14.3.4 Installing Devices Whose Hardware Is Present

Both routines in RT–11 that can install a device handler—the bootstrap code and the monitor INSTALL command code—install handlers only for those devices whose hardware is present on the current system configuration. The routines look at location 176 in block 0 of the handler and test the address that 176 contains, which is normally the base CSR for the device. If the hardware for the device is not present on the system, a bus timeout occurs, causing a trap to 4, which the installation routines field. As a result, neither the bootstrap routine nor the INSTALL command will install the device handler. In addition, the INSTALL command prints the *?KMON–F–Invalid device installation* message.

The installation routines think the device's hardware is present if its CSR responds on the bus. However, this simple test is not sufficient to determine, in some cases, which hardware device is present. For example, some devices are assigned the same addresses in the I/O page for one or more of their status registers. If RT–11 just tested a "shared" I/O page address, it still does not know which of two devices is really present and therefore which handler to install. The RX01 and RX02 diskette devices, for example, have the same bus address and the same number of status registers in the I/O page. When RT–11 attempts to install the DX handler, it must be able to determine whether or not hardware is present, and whether or not it is the RX01 device. Clearly, it should not install the DX handler when the hardware is really the RX02 device.

There is almost always some difference between two or more devices that is discernible from their registers in the I/O page. Each handler for one of the hard-to-identify devices can test for this difference and inform the RT–11 installation routine whether or not it should install the device handler it is currently considering.

1.14.3.5 Writing an Installation Verification Routine

RT–11 handlers for devices with shared I/O page addresses all contain an installation verification routine to distinguish which hardware device is actually present and to permit or inhibit installation of the current handler. If you write a device handler yourself, you can include your own installation verification routine.

In general, the installation verification routines distinguish which hardware is present based on one of the three following conditions:

- Of the two devices that share some registers, one device has more registers than the other.
- If two devices share addresses for all their registers, and if they have the same number of registers, sometimes one device has a read/write bit where the other device has a read-only bit.
- Sometimes a device has a unique identification bit or byte.

The installation verification routines, then, determine which device is present based on the results of testing one of the distinguishing conditions. Once this determination has been made, the routine signals to the RT–11 installation routine whether or not to install the current handler and then returns to the monitor with the carry bit set to prevent installation and with the carry bit clear to permit installation.

Note that your installation verification routine can use all registers.

Entry Points of the Installation Verification Routine

An installation verification routine that you write in your own handler starts at location 200 in block 0 of the handler. It must not extend beyond location 360, unless you link your handler with the /NOBITMAP option – in which case location 376 is the limit. Location 200 is the entry point that the bootstrap code uses to install a data device. The INSTALL monitor code always enters here, as well.

Location 202 is the entry point that the bootstrap code uses to install the system device. The INSTALL monitor code never enters here.

If you do not care whether your handler is installed as the system device or as a data device, put a NOP instruction at location 200. If your handler must be installed as the system device handler, use the following instructions to prevent its installation under any other circumstances:

```
        . = 200                 ;NON-SYSTEM ENTRY POINT
        BR      ERROR           ;BRANCH TO ERROR ROUTINE
        .
        .
        .
        ; Code to execute when installed as system device
        .
        .
        .
ERROR: SEC                      ;SET CARRY TO PREVENT INSTALLATION
        RTS     PC              ;AND RETURN
```

The .DRINS macro sets up the installation code area in block 0 of a device handler. .DRINS defines symbols for the installation verification code entry points and for the installation CSR. After .DRINS is called, the location counter is set to 200, the address of the data device installation entry point.

.DRINS Macro—Use the .DRINS macro near the beginning of your device handler, before the header section. The .DRINS macro is described in Section 1.2.1.3.

If the Hardware for This Handler Has an Extra Register

If this handler is for a device that shares an I/O page address with another device, you can identify which device is present if the two devices have a different number of registers. When the device for the current handler has one more register than the other device, use the following instructions to test for the extra register:

```
        MOV     176,R0                  ;GET THE SHARED CSR
        TST     n(R0)                   ;TEST THE EXTRA REGISTER AT OFFSET n
                                        ;THE SHARED CSR
        RTS     PC                      ;RETURN (WITH CARRY SET
                                        ;IF WRONG DEVICE)
```

This routine tests the extra register. If there is no device configured there, the bus times out, causes a trap to 4, and sets the carry bit. The installation verification routine returns to the monitor with the carry bit set, indicating that the correct hardware for the current handler is not present, and that this handler should not be installed.

On the other hand, if the extra register responds to the test, the TST instruction returns with the carry bit clear, which means that the correct hardware for this device handler is present, and that RT–11 should install the handler.

If the Hardware for This Handler Has Fewer Registers

If the hardware for the other device that shares an I/O address with the device for this handler has more registers, this handler can test for the absence of the extra register. If the extra register is not found, RT–11 should install the current handler.

The following instructions take care of this situation:

```
        MOV     176,R0                  ;GET THE SHARED CSR
        TST     n(R0)                   ;TEST THE EXTRA REGISTER AT OFFSET n
                                        ;FROM 176. IS A DEVICE HERE?
        BCC     1$                      ;YES, OTHER DEVICE IS HERE.
        CLC                             ;NO, CLEAR CARRY
        RTS     PC                      ;INSTALL CURRENT HANDLER

1$:     SEC                             ;SET CARRY
        RTS     PC                      ;DO NOT INSTALL CURRENT HANDLER
```

Essentially, this routine checks for the presence of the other device's extra register. If it is not present, the routine instructs RT–11 to install the current handler.

If an Identification Bit or Byte Exists

If the devices that share an I/O page address also share an identification bit or byte, an installation verification routine can check the bit or byte and determine which hardware is present. It can then permit or inhibit the installation of the current handler based on that information.

In RT–11, for example, the RX01 and RX02 devices share the CSR. Bit 11, called CSRX02, is clear if the device is an RX01, and set if the device is an RX02. The following example is from the DY device handler, which should only be installed if RX02 hardware is present.

```
        .ASECT
        . = 200                         ;VERIFICATION ROUTINE GOES HERE
        NOP                             ;SAME CHECK FOR SYSTEM AND NON-SYSTEM
        BIT     #CSRX02,@176            ;IS RX02 BIT ON?
        BEQ     1$                      ;NO, THIS IS AN RX01.
                                        ;DON'T INSTALL THIS
                                        ;DY HANDLER.
        TST     (PC)+                   ;CLEAR CARRY, SKIP SEC INSTRUCTION.
                                        ;WE HAVE AN RX02, INSTALL DY HANDLER
1$:     SEC                             ;SET CARRY, DON'T INSTALL DY HANDLER
        RTS     PC                      ;RETURN TO MONITOR
```

If One Device Has a Read/Write Bit

If one of the devices that share an I/O page address has a read/write bit in the CSR where the other device has a read-only bit, the verification routine can determine which hardware is present by following a general procedure to check the bit and permit or inhibit the installation of the current handler based on the results. The routine should read the bit, toggle it, and write it back to the CSR. Then the routine should read the bit again. If the value of the bit changed, the device with the read/write register is present. If the value remained constant, the device with the read-only register is present. The routine can set the carry bit appropriately and return to the monitor. If carry is set, RT–11 does not install this handler. If carry is clear, RT–11 does install this handler.

1.14.3.6 Overriding the Hardware Restriction

If for any reason you need to install a device handler whose hardware is not present in your current system configuration, you can circumvent the checks in the bootstrap and INSTALL routines by running SIPP and patching the handler. You clear location 176 in the handler file's block 0, then use the INSTALL command or reboot the system to install the device handler.

1.15 How to Test and Debug a Device Handler

Once your new handler is assembled, linked, and installed, you are ready to begin testing it. Remember during debugging that you must remove the old handler and install the new one each time you create a new version of *dd*(X).SYS.

Test the handler in three stages, according to these guidelines:

1. Use the hardware version (SDH.SYS or SDHX.SYS) of DBG–11 to observe the handler as it processes a data transfer.

   If for some reason you would rather use ODT or VDT to observe the handler as it processes a data transfer, see Sections 1.15.2 and 1.15.3. However, debugging is significantly easier when using a symbolic debugger, so look closely at using DBG–11 before choosing ODT or VDT.

2. Test the handler with keyboard monitor commands, with system utility programs, and with FORTRAN, C, or another programming language. Try the COPY command, for example, to copy data to and from the device, or run PIP to do the same thing. Try using the handler with FORTRAN READ or WRITE statements, or with BASIC–PLUS INPUT or PRINT statements. If your handler sets the bit in the device status word that indicates that the handler is for an RT–11 directory-structured device, DUP will operate correctly on the device with no further modifications. That is, you should be able to use DUP to initialize the device (through the INITIALIZE command) and to consolidate free space (through the SQUEEZE command). The RESORC program needs no modification to recognize the new device and will include it in its SHOW DEVICES report.
3. Give the handler an extended workout with an application program that uses wait-mode I/O, asynchronous I/O, and completion routines.

When the handler passes all the tests successfully, you can begin using it as part of your regular RT–11 system.

1.15.1 Using DBG–11 to Test a Handler

Chapter 5 of the *DBG–11 Symbolic Debugger User's Guide* describes using DBG–11 to debug a device handler. If you have not used DBG–11 previously, you should work through the examples in the manual before testing and debugging your handler.

1.15.2 Using ODT to Test a Handler

The easiest way to use ODT to test a handler is to run ODT as the foreground job. If you normally use only a single-job monitor, it is worthwhile to switch to a multi-job monitor just for debugging.

Since you will be doing some careful debugging work, Digital also recommends that you be the sole user during this time. Bring up your system from a hardware bootstrap. Do not start any system jobs or load any handlers.

Link ODT for the foreground with the following command:

```
.LINK/MAP/FOREGROUND  ODT
```

Next, load the device handler you need to debug:

```
.LOAD dd[X]
```

Now, issue a SHOW D command. Note the address given for the device handler that you are debugging. For this example, assume the value is 131634. Subtract 6 (in octal) from this address to get the base address of the handler. In this case,

```
131634
-    6
------
131626
```

Start ODT as the foreground job:

```
.FRUN ODT
```

```
ODT V01.04
*
```

Set relocation register 0 to the value computed from the address given by the SHOW D command:

```
131626;0R
```

You can step through the handler in memory as you follow the instructions in your assembly listing. The first five words are the header; the first executable instruction is the sixth word. Set your first breakpoint at the sixth word:

```
0,12;0B
```

Set other breakpoints at various points in the handler that you want to examine during debugging. Another critical place is the interrupt entry point. You can find its location by checking the handler's MACRO–11 listing. Remember, the interrupt entry point is called *dd*INT:; you should be able to find it easily and set a breakpoint there.

When you have finished setting breakpoints in the handler, exit from ODT:

```
0;G
```

Now try using the handler. You could try using DUP to initialize the device, or PIP to copy data to the device. Or, run a test program that you have designed especially for this purpose. When execution reaches the first breakpoint in the handler, ODT takes control. Use ODT as usual to examine locations and check their values, or to modify instructions. Note that the default priority of ODT is 7; this prevents other interrupts from disturbing your debugging session. Since you are the only user on the system, ODT's high priority should cause no problem. (Note, however, that the system clock will lose time, and that ODT usually cannot debug race conditions.)

When you are satisfied with the handler's performance, remove the breakpoints from it and proceed with the remainder of execution through the handler:

```
;B
;P
```

Be careful not to unload the foreground job (ODT) while there are still breakpoints set in the handler.

1.15.3 Using ODT in a Mapped Environment

By following a few special guidelines, you can use ODT to debug a device handler in the mapped environment.

Carefully select a place for ODT in memory. You can link it with an application program, or link it so it resides somewhere in memory where it will not be destroyed. If a breakpoint is to be taken in kernel mode, ODT must not reside in the PAR1 area (locations 20000 through 37776). The safest place to put ODT is in the foreground partition, as described in Section 1.15.2.

When you are debugging with ODT, the I/O page must always be mapped.

Setting breakpoints also requires care. As soon as you enter ODT, look at the breakpoint trap vector (BPT) at locations 14 and 16 in low memory. When you set a breakpoint, you must manually set the current mode bits, bits 14 and 15, of the PS at location 16. Set them to the mode you expect at the time the breakpoint occurs. The values are 11 for User Mode, 01 for Supervisor, and 00 for Kernel. (RT–11 utility programs such as PIP and DUP run in User Mode and expect the mode bits to be set to 11.)

After setting breakpoints, type 0;G to exit from ODT. This causes an .EXIT request to be performed, which destroys the BPT vector. So, after you exit from ODT, you must manually reconstruct the contents of the vector by using the Deposit command, as follows:

```
D 14=(correct contents of 14),(correct contents of 16)
```

Make sure no other jobs are running when you do this, since context switching causes this technique to fail.

1.16 Contents of .SYS Image of a Device Handler

Table 1–11 shows the layout of the .SYS image of a handler after assembly and linking. Tables 1–3 and 1–4 contain more information about blocks 0 and 1 of the device handler file image. Locations not otherwise identified are reserved for future use by Digital.

Table 1–11: Device Handler .SYS Image

Location	Contents
000000	Handler identifier in RAD50
000002	Pointer to a FETCH service routine (file address)
000004	Pointer to a RELEASE service routine (file address)
000006	Pointer to a LOAD service routine (file address)
000010	Pointer to an UNLOAD service routine (file address)
000012-000016	Reserved
000020	Device classification
000021	Device classification modifier
000022	First special function (index method) list
000024	Second special function (index method) list
000026	Third special function (index method) list
000030	Pointer to further special functions (extension table method)
000032	Pointer to bad-block replacement table
000034	Reserved

Table 1–11 (Cont.): Device Handler .SYS Image

Location	Contents
000036	Second status word
000040-000050	Reserved
000052	Handler size (*dd*END–*dd*STRT)
000054	Device size (*dd*DSIZ)
000056	Device status word (*dd*STS)
000060	Result of FORCE= parameter; byte contains device SYSGEN options for SET *dd* SYSGEN
000061	Reserved
000062	Pointer to the primary bootstrap (file address)
000064	Bootstrap size in bytes
000066	Pointer to the bootstrap read routine (file address)
000070	Varies with the handler; see Table 1–3
000072	Varies with the handler; see Table 1–3
000074	Size in bytes of total list of handler data table descriptors
000076	Pointer to extended device-unit ownership table (file address)
000100	Letter name of extended device-unit handler and device characteristics for UMR support
000102	Reserved
000104	Pointer to further block 0 type information not included in block 0 (file address)
000106	Pointer to the handler data descriptor table (file address)
000110-000173	Information written by .MODULE and .AUDIT.
	(The CSR table begins at 176 and expands downward to a zero word.)
000174-000xxx	'Display' CSRs (DISCSR) read by RESORC. If more than one, each written into previous location
000176	'Installation' CSR (INSCSR); beginning of CSR table
000200	Data device installation entry point (INSDAT)
000202	System device installation entry point (INSSYS)
000204-000377	Installation code; must link with /NOBITMAP option or else range is 204-357
000400-000777	SET code/tables
001000	Either the device vector or an offset to the table of vectors (*dd*STRT)

Table 1–11 (Cont.): Device Handler .SYS Image

Location	Contents
001002	Offset to the interrupt service entry point
001004	Priority (340)
001006	Pointer to the last queue element (*dd*LQE)
001010	Pointer to the current queue element (*dd*CQE) in the handler memory image
001010	Flag word (in handler file image)
001012	NOP instruction OR'd with flags
001014	Branch instruction (optional)
001016	Second flag word (optional)
001020	Pointer to SPFUN address check routine (optional)
001022	pointer to DMA SPFUN table (optional)
001024	Pointer to LD translation table (optional)
1012	Handler entry point
n	Abort entry point (from .DRAST; may be above 1777)
n+2	Interrupt entry point (from .DRAST; may be above 1777)
1776	High limit of area modifiable by SET code
dd$END	$RLPTR: (from .DREND)
	$MPPTR: (from .DREND)
	$GTBYT: (from .DREND)
	$PTBYT: (from .DREND)
	$PTWRD: (from .DREND)
	$ELPTR: (from .DREND)
	$TIMIT: (from .DREND)
	$INPTR: (from .DREND)
	$FKPTR: (from .DREND)
	*dd*END=. (from .DREND)
*dd*END:	
*dd*BOOT:	NOP; Start of primary bootstrap (from .DRBOT)

Table 1–11 (Cont.): Device Handler .SYS Image

Location	Contents
	BR *entry*; Label *entry* from .DRBOT
entry–14	020; (from .DRBOT) This byte identifies the type of CPU. A value of 20 indicates a PDP–11.
entry–12	Controller types; (from .DRBOT) This byte indicates the type of controllers that the operating sytem supports for this device. Its value in RT–11 V5 can be the OR'd result of the following codes: 101 Non-MSCP UNIBUS controller 102 Non-MSCP LSI–11 buscontroller 110 MSCP UNIBUS controller 120 MSCP LSI–11 bus controller
entry–10	020; (from .DRBOT) This byte identifies the type of file structure on the disk. A value of 20 indicates RT–11 file structure.
entry–6	checksum; (from .DRBOT)The checksum byte is a checksum of the previous three bytes. It is computed as the complement of the sum of the bytes.
entry–4	0; (from .DRBOT)
entry–2	diskette type; (from .DRBOT) This byte contains a bootstrap identification number in bits 0–6 and a flag to indicate single- or double-sided diskettes in bit 7. The values can be: Bit 7 = 0, Single-sided diskette Bit 7 = 1, Double-sided diskette
entry:	BR .+2 or BMI .+2 (from .DRBOT) Digital suggests that *entry* be located above location 120 in the bootstrap block. This will avoid conflict with vectors and the monitor SYSCOM area as the monitor is bootstrapped.
	Start of primary bootstrap read routine
662	High limit of primary bootstrap
664	Start of bootstrap error code
776	End of bootstrap error code

Chapter 2

Programming for Specific Devices

This chapter provides information on device handlers that have special device-dependent characteristics. Read this chapter if you need to program specifically for one of the following devices:

- DL (RL01/RL02 disk handler)
- DM (RK06/RK07 disk handler)
- DU (MSCP disk handler)
- DW (CTI Bus-based disk handler)
- DX and DY (RX01/RX02 diskette handlers)
- DZ (Diskette handler)
- LD (Logical disk handler)
- MM, MS, and MT (Magtape handlers)
- MU (TMSCP magtape handler)
- NL (Null handler)
- NC, NQ, and NU (Ethernet handlers)
- UB (UNIBUS mapping register handler)
- VM (Virtual Memory handler)
- XC and XL (Communications Port handlers)

Much of the information in this chapter is based on other information in the RT–11 documentation set. You should be familiar with pertinent information found elsewhere rather than relying only on the information in this chapter. For example, much of the description of special functions as they apply to particular device handlers in this chapter assumes you know and understand the description of special functions (the .SPFUN request) in the *RT–11 System Macro Library Manual*.

You should look at the on-line index utility, INDEX, or in the printed *RT–11 Master Index* for other information in the RT–11 documentation set that pertains to a particular device handler or to various RT–11 features as they apply to that device handler.

Device handler operations are often controlled by various special functions. In this manual, you will be presented with both a code number and name for a special function. You can use the code in the particular special function call (.SPFUN) as documented. You can use the name (rather than the code) if you include in your program a macro call for the appropriate macro in the file, SYSTEM.MLB.

The following macros in SYSTEM.MLB define the names for the indicated type of special functions:

Name	**Device type**
.SFDDF	Disk device handlers
.SFMDF	Magtape device handlers
.SFNDF	Ethernet device handers
.SFXDF	VTCOM device handlers
.SFODF	'Other' device handlers, such as PI

You could, for example, include the following code in your program or handler to define the names of all the disk type special functions in this manual. Then you could use the special function name, rather than the more cryptic function code. (Be sure that the volume that contains SYSTEM.MLB is also assigned the logical name SRC.)

```
.LIBRARY "SRC:SYSTEM.MLB"
.MCALL .SFDDF
.SFDDF
```

2.1 DL (RL01/RL02 Disk Handler)

This section provides specific programming information for RL01 and RL02 disks.

2.1.1 Support for Special Functions

The RL01/RL02 disk handler supports the following special functions. The device-specific parameter arguments are the same as for DM; see Section 2.2 for information.

Code	Name	Action
377	SF.ARD	Read operation without doing bad-block replacement; returns definitive error data.
376	SF.AWR	Write operation without doing bad-block replacement; returns definitive error data.
374	SF.BBR	Re-read the bad-block replacement table in the handler (the program changed it).
373	SF.SIZ	Determine the size, in 256_{10}-word blocks, of a particular volume.

2.1.2 Support for Bad-Block Replacement

Bad-block replacement for the RL01 and RL02 is similar to the bad-block support for the RK06/RK07 (DM). However, the RL01 and RL02 generate neither the bad sector error (BSE) nor the header validity error (HVRC). Therefore, the handler must check the bad-block replacement table for each I/O transfer. Since the table is always in memory as part of the DL handler, the I/O delay is not significant.

The last track of the RL01 and RL02 disks contains a table of the bad sectors that were discovered during manufacture of the disk. The 10_{10} blocks preceding this table (the last 10_{10} blocks in the second-to-last track) are set aside for bad-block replacements. The maximum number of bad blocks (10_{10}) is defined in the handler.

As with the RK06 and RK07, you determine at initialization time whether to cover bad blocks with .BAD files or create a replacement table for them and substitute good blocks during I/O transfers. The advantage of using bad-block replacement is that it makes a disk with some bad blocks appear to have none. On the other hand, covering bad blocks with .BAD files fragments the disk. Because RT–11 files must be stored in contiguous blocks, this fragmentation limits the size of the largest file that can be stored.

The monitor file cannot reside on a block that contains a replaced block if you are using bad-block replacement. If this condition occurs, a boot error results when you bootstrap the system. In this case, move the monitor so that it does not reside on a block with an error.

If you specify the /REPLACE option during initialization of an RL01 or RL02 disk, DUP scans the disk for bad blocks. It merges the scan information with the manufacturing bad sector table, allocates a replacement for each bad block, and writes a table of the bad blocks and their replacements in the first 20 words of block 1 of the disk. Block 1 is a table of two-word entries. The first word is the block number of a bad block; the second word is its allocated replacement. The last entry in the table is 0. The entries in the table are in order by ascending bad block number. A sample table is shown in Figure 2–1.

Figure 2–1: Bad-Block Replacement Table

Bad block	12	Entry 1
Its replacement	10210	
Bad block	37	Entry 2
Its replacement	10211	
Bad block	553	Entry 3
Its replacement	10212	
End of list	0	Entry 4

The handler contains space to hold a resident copy of the bad block table for each unit. The amount of space allocated is defined by the SYSGEN conditional DL$UN, which represents the number of RL01/RL02 units to be supported. The value defaults to 2 if it is not defined. The handler reads the disk copy of the table into its resident area under the following three conditions:

- If a request is passed to the handler and the table for that unit has not been read since the handler was loaded into memory.
- If a request is passed to the handler and the handler detects Volume Check drive status. This status indicates that the drive spun down and spun up again, which means that the disk was probably changed.
- If an SF.BBR request is passed to the handler. This special function is used by DUP when it initializes the disk table to ensure that the handler has a valid resident copy.

2.2 DM (RK06/RK07 Disk Handler)

This section provides specific programming information for RK06 and RK07 disks.

2.2.1 Support for Special Functions

The RK06/RK07 disk handler supports the following special functions:

Code	Name	Action
377	SF.ARD	Read operation without doing bad-block replacement; returns definitive error data.
376	SF.AWR	Write operation without doing bad-block replacement; returns definitive error data.
374	SF.BBR	Reread the bad-block replacement table in the handler (the program changed it). SF.BBR uses no parameters.
373	SF.SIZ	Determine the size, in 256_{10}-word blocks, of a particular volume.

The special function (.SPFUN) request has the following general form, with the *area* and *chan* parameters and the optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*, and the other parameter arguments as described below:

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

func is the code for the function to be performed or the name of the function if the program has been assembled with the distributed module SYSTEM.MLB.

buf For SF.ARD and SF.AWR, the buffer size must be one word larger than required for the data. The first word of the buffer contains any returned error information. The remaining words in the buffer contain the data transferred. The error codes and information are as follows:

Code	Name	Meaning
100000	ES.SUC	The I/O operation is successful.
100001	ES.ECC	An ECC error is corrected.
100002	ES.RTY	An error was recovered on a retry.
100004	ES.UFF	An error was recovered through an offset retry.
100010	ES.RCL	An error was recovered after recalibration.
100200	ES.BBR	A bad block is detected (BSE error).
1774xx	ES.ERR	An error was not recovered.

For SF.BBR, *buf* should be 0.

For SF.SIZ, *buf* is a 1-word buffer where the .SPFUN request returns the size of the volume in 256_{10}-word blocks.

wcnt For SF.BBR, *wcnt* should be 0.

For SF.SIZ, *wcnt* should be 1.

blk For SF.BBR, *blk* should be 0.

For SF.SIZ, *blk* should be 0.

2.2.2 Support for Bad-Block Replacement

The last cylinder of the RK06 and RK07 disks is used for bad-block replacement and error information. RT–11 supports a maximum of 32_{10} replaceable bad blocks on these disks. The bad-block information is stored in block 1 on track 0, cylinder 0, of the disk. The replacement blocks are stored on tracks 0 and 1 of the last cylinder. A bad-block replacement table is created in block 1 of the disk by the DUP utility program when the disk is initialized. When a bad block is encountered and the table is not present in the handler from the same volume, the DM handler reads a replacement table from block 1 of the disk and stores it in the handler.

When a bad sector error (BSE) or header validity error (HVRC) is detected during a read or write, the DM handler replaces the bad block with a corresponding good block from the replacement tracks. The bad-block replacement feature of RT–11 requires blocks 0 through 5 and tracks 0 and 1 of the last cylinder to be good. This procedure causes an I/O delay since the read/write heads must move from their present position on the disk to the replacement area, and back again.

If this I/O delay cannot be tolerated, the disk can be initialized without bad-block replacement. In this case, bad blocks are covered by .BAD files. Neither the bad blocks nor the replacement tracks will be accessed.

You determine at volume initialization time whether to cover bad blocks with .BAD files or to create a replacement table for them and substitute good blocks during I/O transfers. The advantage of using bad-block replacement is that it makes a disk with some bad blocks appear to have none. On the other hand, covering bad blocks with .BAD files fragments the disk. Because RT–11 files must be stored in contiguous blocks, this fragmentation limits the size of the largest file that can be stored.

Only BSE and HVRC errors trigger the DM handler's bad block replacement mechanism. If a bad block develops that is not a BSE or HVRC error, the disk must be reformatted to have this new block included in the replacement mechanism. Reformatting should detect the new bad block. Mark it so that it generates a BSE or HVRC error and add the block number to the bad-block information on the disk. The disk should then be initialized to add the bad block to the replacement table.

The monitor file cannot reside on a block that contains a BSE error if you are using bad-block replacement. If this condition occurs, a boot error results when you bootstrap the system. In this case, move the monitor so that it does not reside on a block with a BSE error. Further, the monitor file (and any handler files) must reside in physically contiguous blocks—none of the blocks can be in the replacement table.

2.3 DU (MSCP Disk Handler)

This section provides specific programming information for MSCP disk devices.

The DU handler for RT–11 supports any disk system using the Mass Storage Communications Protocol (MSCP) interface. All disks using MSCP appear the same to the host computer. Thus, a single RT–11 DU handler can access any kind of MSCP disk.

2.3.1 Support for Special Functions

The DU handler supports the following special functions:

Code	Name	Section	Action
377	SF.ARD	2.3.5.2	Read operation without doing bad-block replacement; returns definitive error data.
376	SF.AWR	2.3.5.2	Write operation without doing bad-block replacement; returns definitive error data.
373	SF.SIZ	2.3.2	Determine the size, in 256_{10}-word blocks, of a particular volume.
	SF.S16		*blk* argument for SF.SIZ to indicate 16-bit starting block.
	SF.S32		*blk* argument for SF.SIZ to indicate 32-bit starting block.
372	SF.TAB	2.3.6	Returns the MSCP translation table.
371	SF.OBY		Obsolete; replaced by SF.BYP (360).
367	SF.R32	2.3.5.3	Read with 32-bit block number.
366	SF.W32	2.3.5.3	Write with 32-bit block number.
360	SF.BYP	2.3.7	Provides direct MSCP access.

2.3.2 Determining Volume Size (SF.SIZ), Code 373

Special function SF.SIZ returns the volume size in the word pointed to by the *buf* parameter argument. For DU, this special function is enhanced over that provided in the DL, DM, DY, and LD handlers. SF.SIZ for DU can return a 32-bit value for the device volume size and is, therefore, appropriate for use with device volumes that contain more than 65K blocks.

The volume size returned by the enhanced SF.SIZ is determined by any partition mapping. If a partition is mapped to the unit to which the channel is opened, the returned volume size is calculated from the base of the mapped partition to the usable end of the volume. If, for example, you have mapped unit DU1 to partition 1, an SF.SIZ for DU1 returns a volume size from the base of partition 1 to the usable end of the volume. If you reference the first partition on the volume, SF.SIZ returns the usable size of the entire volume.

The following description of parameters lists any differences between those for returning a 16-bit volume size and those for the 32-bit volume size.

Macro Call: .SPFUN area,chan,#SF.SIZ,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block

chan is the channel opened on the unit for which you want the volume size

SF.SIZ is code 373 or the name SF.SIZ if the program has been assembled with the distributed module SYSTEM.MLB

buf For 16-bit value, is the address of a 1-word buffer in which volume size is returned

For 32-bit value, is the address of a 4-word buffer that on return contains a 32-bit value for the volume size followed by a 32-bit value for the MSCP logical block number from which the volume size was calculated.

The low-order base bits contain the value 0 and the high-order base bits contain a value indicating the partition to which this unit is currently mapped. If the unit does not exist, SF.SIZ returns a hard error and the contents of *buf* are undefined

wcnt For 16-bit volume size, is 1. For 32-bit volume size, is 4

blk For 16-bit volume size, is 0, indicating subcode SF.S16. For 32-bit volume size, is 1, indicating subcode SF.S32

2.3.3 Obtaining the DU Device Status (STATU$)

DU has a status word containing information about the last operation performed by the handler. The status word is called STATU$ and is located at an offset from the base of DU. See Table 2–1. The offset is stored in the handler as an entry in the table set up by the .DRTAB macro. The first word of the 2-word table entry is the RAD50 characters *UMS*, followed by the value of STATU$. Using .DRTAB is described in the *RT–11 System Macro Library Manual*. The low 5 bits of STATU$ contain the status information. All other bits are reserved.

Table 2–1: STATU$ Status Information

Octal Value	Meaning
00	Success
01	Invalid command
02	Command aborted
03	Unit off line
04	Unit available
05	Medium format error
06	Write-protected medium

Table 2–1 (Cont.): STATU$ Status Information

Octal Value	Meaning
07	Compare error
10	Data error
11	Host buffer access error
12	Controller error
13	Drive error

Use DBG–11, ODT/VDT, Console ODT, or the E keyboard command to examine the contents of STATU$. You will need to perform customization patch 2.7.32 located in the *RT–11 Installation Guide* to use the E command. Use the SHOW MEMORY command display to find the base of the DU handler and add the offset to that base.

You can obtain the information returned in STATU$ from within a program by calling the sytem subroutine, IGTDUS, as described in the *RT–11 System Subroutine Library Manual*.

2.3.4 Support for Bad-Block Replacement

All MSCP (DU) hard-disk systems support bad-block replacement (BBR), performed either by the disk controller or as a feature of the DU handler. For those MSCP hard disks for which BBR is provided by the controller, no support is required by the DU handler; bad-block replacement is transparent to RT–11.

In MSCP systems that use an RQDX1, RQDX2, or RQDX3 controller, BBR is performed by the controller. In those systems, BBR is done automatically by the hardware and does not require bad-block support in the DU handler.

In MSCP systems that use a KDA50, UDA50, KLESI–QA, or KLESI–UA controller, BBR can be performed by the DU handler.

Table 2–2 lists the MSCP controllers and drives supported by RT–11 and indicates whether bad-block replacement (BBR) is performed by the controller or the DU handler. (There is no BBR support for RX50 devices or write-only media.)

Table 2–2: MSCP Bad-Block Replacement (BBR)

MSCP Controller	Bad Block Replaced by:	MSCP Drive
RQDX1	controller	Supported RD-type drives
RQDX2	controller	Supported RD-type drives
RQDX3	controller	Supported RD-type drives
KLESI–QA	handler	Supported RC-type drives

Table 2–2 (Cont.): MSCP Bad-Block Replacement (BBR)

MSCP Controller	Bad Block Replaced by:	MSCP Drive
KLESI–UA	handler	Supported RC-type drives
UDA50	handler	Supported RA-type drives
KDA50	handler	Supported RA-type drives

The distributed DU for mapped monitors (DUX.SYS) supports handler BBR. If you are going to use an unmapped monitor with MSCP disks that require handler BBR, you should perform a system generation for that monitor and request support for DU handler bad-block replacement. Once you have generated such support, you can change monitors and continue DU handler bad-block replacement.

The following is general information on BBR as performed by DU:

- Bad-block replacement is a technique in which substitute blocks are provided for blocks that have caused a read or write error. The replacement blocks appear to occupy the disk positions of the original blocks, and the disk appears to contain only good blocks. You can force bad-block replacement on a device by performing a read and verify operation on all blocks. You perform such a read/verify operation by issuing a FORMAT/VERIFY:ONLY command for the device.
- Whether bad-block replacement is performed by the controller or the handler, it has the effect of making a disk appear to be error free. In certain cases, however, an I/O operation, a verification procedure, or a bad-block search may report the presence of bad blocks on a disk with replaced blocks. In such cases, any block identified as a bad block should be considered to be a good block with bad data. This means that the controller or handler provided a replacement block for a defective block but was unable to recover the data it contained.
- You can force MSCP class devices to clear bad blocks that contain soft errors by coupling the DUP /H option with the /B or /K option. The /H option is not available as a KMON command. You should use only the DUP /H/B or /H/K command options with blank media or a volume you have just backed up.
- If the DU handler is unable to replace a block on a device, DU displays the following error message:

  ```
  ?DU-E-Replace command failure or inconsistent RCT.
  ?DU-E-Software write protecting volume.
  ```

  If you receive that message, you should immediately back up that volume. Then check any file you had open for lost data. You cannot write to that volume again without first taking it off line and then placing it on line.

2.3.5 Non-File-Structured Read and Write Operations

DU supports three methods for performing non-file-structured read and write operations.

2.3.5.1 JREAD and JWRITE

You can perform absolute (non-file-structured access) reads and writes to any MSCP device, using the JREAD and JWRITE system subroutines. JREAD and JWRITE use a 32-bit starting block number, which lets you read and write to any block on any DU device. See the *RT–11 System Subroutine Library Manual* for details on JREAD and JWRITE.

2.3.5.2 Special Functions SF.ARD and SF.AWR

DU supports special functions SF.AWR (code 376) and SF.ARD (code 377). SF.AWR and SF.ARD are appropriate for devices that contain no more than 65K blocks. If the DU device contains more than 65K blocks, see Section 2.3.5.3. For DU, SF.AWR performs a write to the specified sector, and SF.ARD performs a read from the specified sector. Those writes and reads are not absolute; bad-block replacement and block vectoring remain in force.

Special functions SF.AWR and SF.ARD are especially useful because they return status information in the first word of the return buffer. Status information includes any occurrence of a bad-block error, forced error, or drive error. No discrimination for such errors is returned by a .WRITE or .READ request.

DU support for SF.AWR and SF.ARD is the same as DM with the following exceptions:

- DU supports an additional error code:

Code	Name	Meaning
140000	ES.FRC	A forced error occurred. If the device is a disk drive that supports BBR, the device controller or DU handler discovered bad data on a good (replaced) block. (Bad-block replacement was performed but no data was recovered.) If the device does not support BBR, this is an unexpected condition.

- For DU, bad-block replacement and block vectoring remain in force.

2.3.5.3 Special Functions SF.R32 and SF.W32

DU supports two special functions that perform non-file-structured block reads (SF.R32, code 367) and writes (SF.W32, code 366) on devices that contain more than 65K blocks. Because these special functions perform non-file-structured operations, they should generally not be used to perform operations on any device partition that contains a file structure.

Special functions SF.W32 and SF.R32 perform the same operations as the JWRITE and JREAD functions; JWRITE and JREAD use special functions SF.W32 and SF.R32. JWRITE and JREAD are described in the *RT–11 System Subroutine Library Manual*.

CAUTION

SF.W32 can write data to the reserved blocks on your DU device, which can render your DU device useless, because those blocks contain the replacement control table (RCT). You should, therefore, always issue a special function SF.SIZ (373) to a DU device to determine the volume size, because SF.SIZ returns the size at the boundary between the usable logical blocks and the RCT. Writing data only up to the volume size returned by SF.SIZ ensures you will not write data into the RCT.

The format for these special functions is:

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number for I/O in the range 0 to 376_8.

func is the symbol or numeric code value for the function to be performed:

Code	Name	Meaning
366	SF.W32	32-bit non-file-structured block write
367	SF.R32	32-bit non-file-structured block read

buf is the buffer address.

wcnt is the number of words to transfer. Valid values are 0 through 077777_8.

blk is the address of a 4-word argument block:

blk+0 is a 2-word (32-bit) starting block number for this request. The first word contains the low-order bits. The second word contains the high-order bits.

The correspondence between the starting block number and a particular block on a device is determined by any partitioning and unit mapping of the device:

If the device has not been partitioned, starting block 0 specifies physical (and logical) block 0 — the start of the device. Any starting block number is offset from physical block 0.

If the device has been partitioned, logical block 0 of partition 0 continues to contain physical block 0. However, the starting block 0 of this request, because of device partitioning, corresponds to logical block 0 of the unit opened on this channel. Any starting block number is offset from logical block 0 of the partition mapped to the unit. For example, if the channel is opened for a non-file-structured operation to unit DU1 and DU1 is mapped to partition 1 (block 200000_8, starting block 0 corresponds to physical block 200000_8 of this device).

If, for example, your device contains an RT–11 file structure in partition 0, which is mapped to DU0, you could ensure the integrity of that file structure by always performing non-file-structured operations above partition 0 on the device.

blk+4 on return, contains the number of words actually transferred

blk+6 is reserved

2.3.6 DU Translation Table (SF.TAB), Code 372

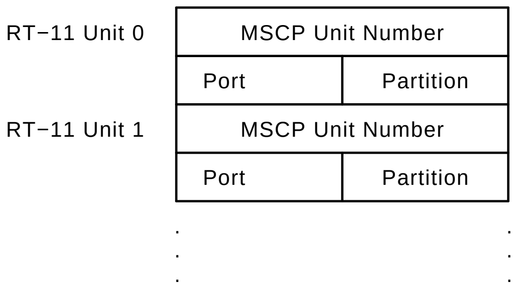
The DU translation table defines the correspondence between RT—11 unit numbers and MSCP unit numbers, ports, and partitions. The format of the table is given in Figure 2–2.

Special function SF.TAB (code 372) interacts with the translation table from an address contained in the *buf* argument of the SF.TAB call. You can read the contents of the translation table to the buffer or write the contents of the buffer to the table. Whether the SF.TAB request is a read or write operation is determined by the *wcnt* parameter argument. This procedure is explained in this section.

For RT–11 V5.4, changes were made in the structure of the DU handler translation table. The names of the offsets in the table and the size of the table was changed. All programs you write to access the information contained in the table should use the following offsets. All programs you have written should be changed to use the following offset names.

Beginning with RT–11 V5.5, you can build a DU handler that supports more than eight units. That affects the size of the translation table.

Figure 2–2: DU Handler Translation Table



Whenever an I/O request is passed to the DU handler, DU uses the RT–11 unit number as an index into this table, extracts the MSCP unit number, port, and partition that have been assigned to that RT–11 unit, and uses the information to access the proper disk.

Size of the Translation Table

The size of the DU translation table in the DU handler is related to the number of device units supported by DU. The DU handler can support up to 64_{10} units. Therefore, the translation table can contain up to 64 table entries.

Structure of the Translation Table

The DU unit translation table consists of a table header followed by table entries. Previously, the DU unit translation table had no header. Now, the DU unit translation table has a header starting at offset DU.ID, which is a word containing the Radix–50 value for the characters DU.

DU.ID is followed by DU.NUM. The low byte of DU.NUM contains the number of entries in the table. The high byte of DU.NUM is reserved.

The structure of the rest of the table remains as before. However, the offset names you should use to specify elements of the table have changed. The following is the structure of the table with the changed offset names:

Table 2–3: MSCP (DU) Translation Table Header

Offset	Name	Meaning
0	DU.ID	Radix–50 value for characters DU
2	DU.NUM	Byte containing number of entries in table
3		Reserved
4	DU.ENT	The offset of the first table entry

Each table entry consists of 4 bytes. Digital recommends you use the symbol DU.ESZ to represent the 4-byte size of each entry.

Table 2–4: MSCP (DU) Translation Table Entry

Offset	Name	Meaning
0	DU.UNI	Physical MSCP unit number. The symbol DU$Uxx=nnnnnn is the initial value for the translation table when the handler is assembled. In the symbol, *xx* is the octal RT–11 DU unit number (0-7 or 0-77) and *nnnnnn* is the MSCP unit number. The SET Dxx UNIT=nnnnnn command can subsequently change the value.
2	DU.PAR	Byte containing partition number. The symbol DU$Axx=nnn is the initial value for the translation table when the handler is assembled. In the symbol, *xx* is the octal RT–11 DU unit number (0-7 or 0-77) and *nnn* is the partition number. The SET DU PART=nnn command can subsequently change the value.
3	DU.POR	Byte containing MSCP port (controller) number. The symbol DU$Oxx=nnn is the initial value for the translation table when the handler is assembled. In the symbol, *xx* is the octal RT–11 DU unit number (0-7 or 0-77) and *nnn* is the MSCP port number. The SET DU PORT=nnn command can subsequently change the value.

Accessing the Translation Table

Before Version 5.5, the translation table access special function code SF.TAB (372) supported only eight units. The *wcnt* parameter for SF.TAB accepted two arguments, SF.TRD (1) to indicate a read of the table and SF.TWR (–1) to indicate a write to the table. The size of the table was fixed at eight entries. If the DU handler on your system continues to support only eight DU devices, you continue to read and write to the translation table as before.

However, if the DU handler on your system supports more than eight units, the SF.TAB special function accepts other values for the *wcnt* parameter to support the extended device units. For DU handlers that implement the extended device-unit feature, you indicate both a read or write operation and the size of the table you are reading and writing by specifying a positive or negative numeric argument for the *wcnt* parameter. A positive numeric argument indicates a read operation of the specified number of words from the DU translation table to the buffer. A negative number indicates a write operation of the specified number of words from the buffer to the DU translation table.

You can use the following procedure to read the translation table from a DU handler that supports extended device units into a buffer and write the translation table from a buffer to DU. The procedure assumes you want to verify or do not currently know the number of entries in the table.

1. A translation table entry is created for each supported unit. You can determine the number of entries by doing a read SF.TAB to return the table entry DU.NUM. DU.NUM is the low byte of the second word in the table and contains the octal number of table entries. Therefore, for the *wcnt* parameter, supply the argument +2, and for the *buf* parameter, point to a 2-word buffer.

2. The translation table header and each entry continue to contain two words. Therefore, you can then read the entire DU handler extended device-unit translation table by supplying the value HEADER+(2*DU.NUM) for the *wcnt* parameter. For example, if DU.NUM indicated 16 entries, the value to specify for *wcnt* to read the entire table would be +(2+(2*16)). The *buf* parameter would point to a buffer of the same size.
3. You could write the contents of the buffer to the DU handler by specifying the value -(2+(2*16)) for the *wcnt* parameter.

You can avoid the calculation process by specifying a buffer of 130_{10} words, which can hold the largest translation table.

2.3.7 Special Function Bypass (SF.BYP), Code 360

Special function SF.BYP bypassess all unit number translations and allows direct access to the MSCP port. For DU, SF.BYP (direct MSCP assess) serves the same purpose as the MU handler's SF.BYP (direct TMSCP access).

The request syntax and parameter argument definitions for SF.BYP are as follows:

Macro Call: .SPFUN area,chan,#SF.BYP,buf,wcnt,blk

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

SF.BYP is code 360 or the name SF.BYP if the program has been assembled with the distributed module SYSTEM.MLB.

buf is the address of the 52_{10}-word TMSCP area.

wcnt when nonzero, is the virtual address of a data buffer to send to the handler. That virtual address is translated to a physical address and placed in the buffer of the TMSCP area.

when zero, the buffer address in the TMSCP area is not altered

blk indicates whether the handler should perform retries:

1 = specifies retries

0 = specifies no retries

The buffer address in special function SF.BYP must point to a 52-word area in the user's job. The first 26 words are used to hold:

- A response packet length in bytes
- A virtual circuit identifier
- An end packet when the command is complete

The second 26 words are set up by the caller and contain:

- A length word (length of command)
- A virtual circuit identifier (must have octal 1 (001) in high byte)

- A valid MSCP command (48-byte command buffer)

Except for port initialization, the user program must do all command packet sequencing, error handling, and reinitialization when the bypass operations are complete. The format of the control block is shown below:

Word	Contents
0	Response Packet Length
1	Virtual Circuit ID (from UDA or QDA controller)
2	MSCP Response Buffer (24 words)
26	Command Packet Length (48 bytes)
27	Virtual Circuit ID (from host)
28	MSCP Command (24 words)
51	Last Word of MSCP Command Packet

2.3.8 Addressing an MSCP Disk

You identify an MSCP disk to the DU handler by specifying:

- The MSCP unit number, in the range 0 through 253
- The controller port number, in the range 0 through 3
- The disk partition number, in the range 0 through 255

As DU is distributed, you address a disk—DU0 through DU7, as desired—and the DU handler references the disks that have been assigned to those RT–11 unit numbers. You can perform a system generation and request extended device-unit support for DU, which lets you address up to 64_{10} disks. See the *RT–11 System Generation Guide* for information.

The default port number is 0, the default partition number is 0, and the default unit numbers correspond to the RT–11 unit numbers. Thus, if no modifications or SET commands are made to the DU handler, an MSCP disk will be referenced exactly like any other RT–11 disk; DU0 will refer to disk unit 0, DU1 will refer to disk unit 1, and so on. However, the names DU0 through DU7 can be reassigned to the MSCP disks of your choice by specifying MSCP unit, port, and partition numbers. Each of these parameters is described below.

2.3.8.1 MSCP Unit Numbers

Traditionally, there has always been a one-to-one correspondence between a physical disk drive unit number and an RT–11 disk unit number. This one-to-one correspondence does not necessarily apply to disks using the MSCP interface. Neither is an MSCP disk controller limited to eight units, nor are the unit identifying numbers limited to the range 0 through 7. The MSCP unit number of a disk is defined by the unit number plug of the disk drive. Although MSCP disks on most RT–11 systems may never have a unit number plug greater than 7, MSCP unit numbers can be in the range 0 through 253. The DU handler supports a 16-bit MSCP unit number, if required by the system configuration.

The relationship between an RT–11 unit number and an MSCP disk unit number is defined within the DU handler. Typically, any necessary assignments are made at system installation time by using a SET command in the following form:

```
SET DUn UNIT=x
```

For example, you might issue the SET command

```
SET DU7 UNIT=21
```

Any references to DU7 would then go to MSCP unit number 21.

2.3.8.2 Controller Port Numbers

The controller port number provides a way of logically identifying the vector/CSR pair of a particular MSCP controller when your system has more than one.

You can access a second MSCP controller through the DU handler in one of two ways. One way is to create a second copy of the handler, as described in Section 2.3.9. You can then use the original DU handler to access disks connected through the first controller port, and the new copy of the handler to access disks connected through the second controller port. Although this procedure requires two copies of the handler, it allows totally independent operation of the two ports, giving maximum I/O throughput.

The second way is to configure the DU handler for multiple ports by defining the conditional assembly parameter DU$PORTS=n. If memory space is at a premium, this may be your best choice. However, the ports will not operate independently and I/O throughput may be slower. If a request is pending for a disk interfaced through port 0, any requests for a disk interfaced through port 1 must wait for the port 0 I/O to complete. The DU handler supports up to four ports, numbered 0 through 3. CSR and vector values for each port can be assigned with SET commands in the following form:

```
SET DU VECTOR=nnnnnn
SET DU VECx=nnnnnn

SET DU CSR=mmmmmm
SET DU CSRx=mmmmmm
```

The value for *x* can be 2, 3, or 4.

If you configure the DU handler for multiple ports, you must specify the port number when you assign an RT–11 unit number to a disk interfaced through a port other than 0. You can do this with a SET command in the following form:

```
SET DUn PORT=x
```

For example, you might issue the SET command:

```
SET DU7 PORT=1
```

This command might be combined with an MSCP unit number assignment:

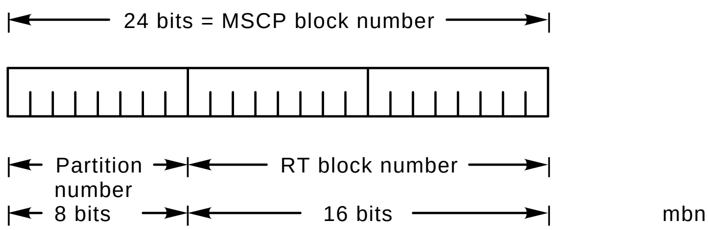
```
SET DU7 UNIT=21,PORT=1
```

You can perform a system generation and request support for multiport booting, as described in Section 2.3.10.

2.3.8.3 Disk Partition Numbers

Disk partition numbers allow RT–11 to use disks having more than 65,535 blocks. The disk partition number can be thought of as a high-order block number, as shown in Figure 2–3.

Figure 2–3: MSCP Disk Block Number



If a disk has more than 65,535 blocks, the DU handler divides the disk into logical partitions of 65,535[1] blocks each. The DU handler supports up to 256_{10} disk partitions. Therefore, the largest disk DU can access has 256*65,535 blocks. To an RT–11 user, such a disk would appear to be 256 separate 65,535-block disks, each disk having its own directory.

Because the DU handler stores the partition numbers as bytes, DU supports an MSCP block number of no more than 24 bits, even though full MSCP supports block numbers of up to 32 bits. However, the partition number entries in the DU handler's translation table could be expanded to word entries if desired and 32-bit block numbers supported with no particular difficulty. Refer to Section 2.3.1 for details of the format of the DU handler's translation table.

Partition numbers are assigned with a SET command in the following form:

```
SET DUn PART=x
```

For example, you might issue the SET command

```
SET DU3 PART=1
```

This command could be combined with unit and port assignments as well:

```
SET DU3 UNIT=2, PORT=0, PART=1
```

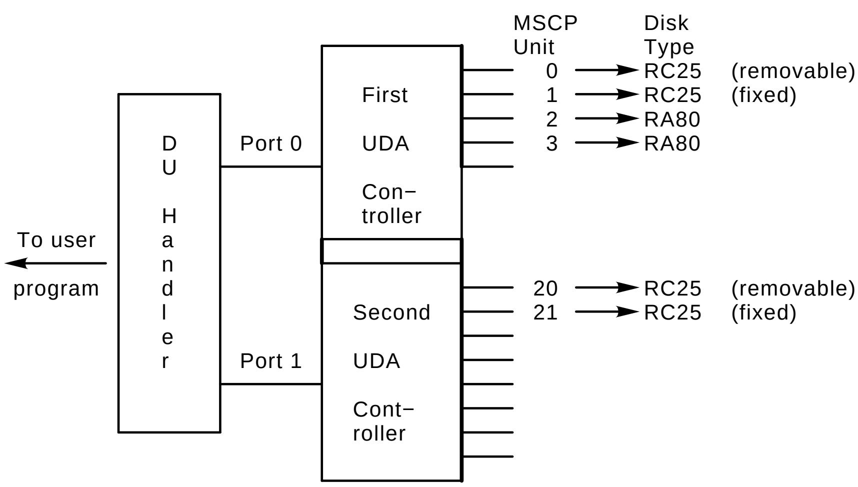
[1] Although RT–11 block numbers can be 0 through 177777_8, or a total of $65{,}536_{10}$ blocks (200000_8 , or 000000 in 16 bits since the 17th bit is lost), the size of a partition is defined as $65{,}535_{10}$ blocks (177777_8), with RT–11 block numbers 0 through 177776. This avoids the problem of 16-bit overflow when dealing with the partition size. Because the partition number is added onto the left of the RT–11 block number to give the MSCP block number, one block between each partition is unused. Refer to the list below for the block numbers of the first three partitions:

Partition	Block Numbers
0	000000–177776, block 177777 unused
1	200000–377776, block 377777 unused
2	400000–577776, block 577777 unused

The mnemonic DU3 will then refer to the MSCP disk with unit plug 2 interfaced through port 0, beginning at block 65,536 of the disk (partition 1).

An example using several disks may help to clarify these concepts. Consider the example of a system with two UNIBUS Disk Adaptor (UDA) controllers interfaced to six disks, shown in Figure 2–4.

Figure 2–4: Two-Port DU Handler



The user of the system illustrated issues the following SET commands:

```
SET DU0 UNIT=0,PORT=0,PART=0
SET DU1 UNIT=1,PORT=0,PART=0
SET DU2 UNIT=2,PORT=0,PART=0
SET DU3 UNIT=2,PORT=0,PART=1
SET DU4 UNIT=3,PORT=0,PART=0
SET DU5 UNIT=3,PORT=0,PART=1
SET DU6 UNIT=20,PORT=1,PART=0
SET DU7 UNIT=21,PORT=1,PART=0
```

These commands assign DU0 to the first (removable) disk of the RC25 with MSCP unit number 0, and DU1 to the fixed disk of the RC25, identified as MSCP unit number 1. The disk unit with MSCP unit number 2 is an RA80, which has more than 65,535 blocks. Therefore, the next commands assign DU2 and DU3 to partition 0 and partition 1 of this disk, respectively. DU4 and DU5 are assigned in similar fashion to partitions 0 and 1 of the RA80 with MSCP unit number 3. Another RC25, interfaced to the second port of the UDA controller, is identified by MSCP units 20 and 21. The last two SET commands assign DU6 and DU7 to the two disks of this RC25 disk system. See Table 2–3 for information on setting up the default settings.

2.3.9 Creating a Second DU Handler

You can create a second DU handler under all monitors. The procedure is different for unmapped or mapped monitors.

2.3.9.1 Under Unmapped Monitors

You cannot run multiple DU handlers through the same MSCP controller; each handler must have a separate controller. Copy the handler to another file name and then modify the new file. Use the handler SET commands to change the vector and CSR of the copy to the values for the second port. For example, you could copy DU.SYS to DA.SYS and use the following SET commands to change the CSR and vector of the DA file:

```
SET DA VEC=nnnnnn
SET DA CSR=mmmmmm
```

The variables *nnnnnn* and *mmmmmm* are the vector and CSR addresses of the second port.

2.3.9.2 Under Mapped Monitors

You cannot run multiple DU handlers through the same MSCP controller; each handler must have a separate controller. You can use the following procedure to create a second DU handler that can be used together with the distributed DU handler under all mapped monitors:

1. If you intend to perform a system generation to build a DU handler with support for extended device units or for any other reason, you must do that before creating a second DU handler. You must also preserve the system generation work files.

2. The second DU handler must be assigned a name that does not conflict with any distributed handler. If the second DU handler will be assembled for extended device-unit support, the first letter of the second DU handler cannot be D or L. For the purpose of this procedure, the second DU handler is named BU. Therefore, copy the DU source file to BU:

   ```
   .COPY DU.MAC BU.MAC [RET]
   ```

3. Unprotect BU.MAC and open it with the editor.

4. Perform a search operation for the symbol DU$NAM and on the other side of the equal sign, change the string <^RDU > to <^RBU >, so that the entire line of code resembles the following:

   ```
   .IIF NDF DU$NAM,        DU$NAM = <^RBU >
   ```

5. Exit from the editor.

6. If you used the system generation procedure to build the DU handler, use the following procedure to assemble and link BU.MAC. If you did not build the DU handler and are using the distributed DU, proceed to step 7.

   a. Copy the device-build (.DEV) command file that was created during the system generation to a file named BU.DEV.

b. Open the file BU.DEV on the editor.

c. Perform a search operation for the string +*DU*. The search places the cursor near the end of the first of three command lines that pertain to DU. The three command lines begin with MACRO, LINK, and SETOVR.

   By placing an exclamation mark (!) character at the beginning of each line, comment out all command lines except the initial commands that assign device logical names and the three command lines that apply to DU.

d. On the command lines that assemble and link DU, change all references from DU to BU, by replacing the *D* with *B*.

e. Exit from the editor.

f. Issue the following command to run BU.DEV as a command file:

```
.$@BU.DEV RET
```

   BU.DEV builds the file BUX.SYG.

g. When BU.DEV has completed, copy the file BUX.SYG to BUX.SYS.

h. Determine the current CSR and vector addresses for DU, using the following command:

```
.SHOW DEV:DU RET
```

   The MSCP port characteristics, such as CSR and vector addresses, for DU and BU cannot overlap. Specify addresses for BU that do not conflict with DU by using appropriate SET commands.

7. If you did not build DU by using the system generation process, issue the following commands to assemble and link BU. In the commands, *ddn* represents that device on which the distributed system conditional file (such as XM.MAC), the created file, BU.MAC, and the system library SYSTEM.MLB reside:

```
.ASSIGN ddn: SRC RET
.MACRO/OBJ:BUX ddn:(XM+BU) RET
.LINK/NOBITMAP/EXE:BUX.SYS/BOUNDARY:512. DK:BUX RET
Boundary?  SETOVR RET
```

8. Determine the current CSR and vector addresses for DU, using the following command:

```
.SHOW DEV:DU RET
```

   Specify addresses for BU that do not conflict with DU by using appropriate SET commands.

2.3.10 Multiport Booting

During system generation, you can select an option for the DU handler that will let you boot RT–11 from any DU port. If you do not specify DU multiport booting during SYSGEN, you can boot RT–11 from DU port 0 only. Use the following procedure to enable multiport DU booting:

1. Use the SET DUn commands to map the particular DU device to the MSCP unit, port, and partition numbers. For example:

   ```
   .SET DU3 UNIT=0, PORT=1 RET
   .SET DU4 UNIT=1, PORT=1 RET
   .SET DU5 UNIT=2, PORT=1 RET
   ```

   For the SET commands to take effect, you must UNLOAD and then LOAD the handler if it is a data device or reboot it if it is a system device.

2. Copy the resulting DU handler to the port on the DU devices you want to be able to boot. For example:

   ```
   .COPY DUX.SYS DU3: RET
   ```

3. To hard-boot the DU unit on a new port, use the COPY/BOOT command to copy the bootstrap to the volume on the desired port. The DU unit on that port will also support the soft-boot BOOT DUn: command.

2.4 DW (CTI Bus-based Disk Handler)

This section provides specific programming information for the hard disks on CTI Bus-based computers.

2.4.1 Support for Special Functions

The DW handler supports the following special functions:

Code	Name	Action
377	SF.ARD	Read
376	SF.AWR	Write
373	SF.SIZ	Return device size

The special function (.SPFUN) request has the following general form, with the *area* and *chan* parameters and the optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual* and the other parameter arguments as described below:

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

func is the special function code or the name if the program is assembled with the distributed file SYSTEM.MLB.

buf For SF.ARD and SF.AWR, is the address of a 256_{10}-word buffer.

For SF.SIZ, is the address of a one-word buffer in which the size of the volume is returned.

wcnt For SF.ARD and SF.AWR, is the track to read or write.

blk For SF.ARD and SF.AWR, is the logical block (rather than physical block) to be read or written. Because the physical block number for DW is one less than the logical block number, address physical block 0 as logical block –1.

For SF.SIZ, should be set to 0.

2.5 DX and DY (Diskette Handlers)

This section provides specific programming information for RX01 and RX02 diskettes.

As distributed, DX and DY support one controller that supports two drives. Each DX and DY handler can support two controllers (and therefore four drives). For example, if the RX01 handler is created through system generation to support two controllers, it will support four devices: DX0, DX1, DX2, and DX3. DX0 and DX1 are drives 0 and 1 of the standard diskette at CSR 177170 and vector 264. DX2 and DX3 are drives 0 and 1 of the other controller (standard alternate address CSR 177150 and vector 270). Note that only one I/O process can be active at one time, even though there are two controllers. Overlapped I/O to the handler is not permitted.

Data is stored on DX and DY diskettes in sectors. Double-density diskette sectors are 128 words long. RT–11 normally reads and writes them in groups of two sectors. Single-density diskette sectors are 64 words long. RT–11 reads and writes them in groups of four sectors. However, special function requests for absolute reads and writes can access sectors individually.

2.5.1 Support for Special Functions

The DX and DY handlers support the following special functions:

Code	Name	Action
377	SF.ARD	Read absolute sector
376	SF.AWR	Write absolute sector
375	SF.WDD	Write absolute sector with deleted data mark
373	SF.SIZ	Return device size, in 256_{10}-word blocks (DY only)

A request to write absolute blocks should not write anything in track 0 if you want to use DUP or the COPY/DEVICE command to back up the volume. DUP does not copy data in track 0. Also, be sure you specify a valid buffer address and word count. The monitor checks that the *buf* parameter argument is in the job area, but it does not check the validity of *buf+(2*wcnt)-1*.

The special function (.SPFUN) request has the following general form, with the *area* and *chan* parameters and the optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual* and the other parameter arguments as described below:

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

func is the code for the function to be performed, or the name of the function if the program has been assembled with the distributed module SYSTEM.MLB.

buf For SF.ARD, SF.AWR, and SF.WDD, is the location of a 129-word buffer (for double-density diskettes) or a 65-word buffer (for single-density diskettes). The first word of the buffer, the flag word, is normally set to 0.

The flag word set to 1 indicates a read on a physical sector containing a deleted data mark. The data area of the buffer extends from the second word to the end of the buffer.

buf for SF.SIZ is the location of a one-word buffer in which 494 is returned by single-density diskettes and 988 is returned by double-density diskettes.

wcnt For SF.ARD, SF.AWR, and SF.WDD, is the absolute track number, 0 through 76, to be read or written.

wcnt for SF.SIZ is reserved and should be set to 1

blk For SF.ARD, SF.AWR, and SF.WDD, is the absolute sector number, 1 through 26, to be read or written.

blk for SF.SIZ is reserved and should be set to 0.

The diskette should be opened with a non-file-structured .LOOKUP. The following example performs a synchronous sector read from track 0, sector 7, into a 65-word area called BUFF.

```
.SPFUN  #RDLIST,#SF.ARD,#BUFF,#0,#7,#0
```

2.6 DZ (Diskette Handler)

This section provides specific programming information for diskettes on CTI Bus-based computers.

2.6.1 Support for Special Functions

The DZ handler supports the following special functions:

Code	Name	Action
377	SF.ARD	Read absolute sector
376	SF.AWR	Write absolute sector

The special function (.SPFUN) request has the following general form, with the *area* and *chan* parameters and the optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual* and the other parameter arguments as described below:

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

func is the code for the function or the name of the function if the program is assembled with the distributed file SYSTEM.MLB.

wcnt is the track to be written.

blk is the sector.

buf is the address of a 256_{10}-word buffer.

The .SPFUN requests do not interleave sectors. RX50 diskettes have 80 tracks, and the .SPFUN requests wrap to track 0 after track 79.

2.7 LD (Logical Disk Handler)

This section provides specific programming information for logical disks.

The Logical Disk handler implements logical disk support. The LD handler accepts I/O requests just like any other disk handler. By means of embedded translation tables, the LD handler determines which physical disk and which starting block offset should be used for each LD I/O request. When the proper physical disk and block number are determined, the LD handler updates the block number and unit number in the I/O queue element so that they correspond to the values for the assigned physical disk. The LD handler then places the queue element on the I/O queue for the physical disk so that the actual I/O can take place.

In addition to operating as outlined above, the LD handler can also be run as a program. When run, the LD handler accepts CSI command lines and switches to initialize, assign, verify, write-enable, or write-lock logical disk units.

2.7.1 Support for Special Functions

The logical disk handler supports the following special functions:

Code	Name	Action
372	SF.TAB	Access the translation tables
373	SF.SIZ	Return unit size. The parameter arguments for SF.SIZ for LD are the same as for DM. See Section 2.2 for information

2.7.2 LD Translation Tables (SF.TAB), Code 372

Special function SF.TAB (code 372) interacts with the translation tables from an address contained in the *buf* parameter argument of the SF.TAB call. You can read the contents of the translation tables to the buffer or write the contents of the buffer to the tables. Whether the SF.TAB request is a read or write operation is determined by the *wcnt* parameter argument. This procedure is explained in this section.

For RT–11 V5.4, changes were made to the structure of the LD translation tables. All programs you write to access the information contained in those tables should reflect the changes. All programs you have written to access LD translation tables should be changed to reflect the changes.

The tables start at a header; previously they started at a label. Following the 2-word header are four LD translation tables. That is unchanged. However, the names of offsets you use to reference the tables have changed. Some table contents have also changed.

Further, you can now build support for up to 64 logical disk units, which affects how you use the tables.

Size of the Translation Tables

The size of the LD translation tables in the LD handler is related to the number of logical disk units supported by LD. Beginning with Version 5.5, you can use the system generation procedure (SYSGEN) to build an LD handler that supports extended device units. By default, SYSGEN builds support for 16_{10} logical disk units when you request extended device-unit support. You can request up to 64_{10} units. Of those 64 units, 32 can be mounted and 32 are reserved to Digital.

Structure of the Translation Tables

The LD translation tables consist of a 2-word header followed by four LD translation tables. The LD translation tables start at header LD.ID. Header LD.ID is a 1-word table identifier and contains the Radix–50 value for the characters LD. Header LD.ID is followed by LD.NUM, a 1-byte count of the number of entries in the table. As LD is distributed, the value in LD.NUM is 10_8, indicating eight table entries. If LD is built for extended device-unit support, the value in LD.NUM can contain a value up to 100_8, indicating support for up to 64 logical disk units. LD.NUM is the low-order byte of the word LD.ID+2. The high-order byte of LD.ID+2 is reserved.

The four LD translation table offset names, location, and contents are:

LD.FLG (LD.ID+4) The table beginning at offset LD.FLG is the table previously at the label HANDLR. LD.FLG contains one word for each LD unit number. The count of LD unit numbers is stored in LD.NUM. The bits in each word of LD.FLG have the following meaning:

Bits	Name	Meaning
0–5	LD.NDX	An index to the handler tables in RMON for the physical device corresponding to the LD unit number.
6	LD.UNX	A flag that signals the index entry (bits 0–5) may be inaccurate and should be updated. LD sets LD.UNX for all units if, upon entry, the LDREL$ bit in RMON fixed offset CONFG2 is set.
7	LD.UOF	A flag that signals the entry in the LD.OFS table for that LD unit may be inaccurate. LD.UOF is set whenever a volume is squeezed. LD checks LD.UOF each time it uses an LD unit; if set, LD verifies that unit's LD.OFS table entry before proceeding.
8–13	LD.UNT	Contain the unit number of the physical disk assigned to the logical disk unit.
14	LD.RDO	Is the write-lock bit. If LD.RDO set, the LD unit is read only.
15	LD.ACT	Is the allocation bit. If LD.ACT set, the LD unit is assigned. If LD.ACT clear, the LD unit is not assigned.

LD.OFS (LD.FLG+<2*Contents of LD.NUM>) The second translation table starts at the offset LD.OFS and contains one word for each LD unit number. The count of LD unit numbers is stored in LD.NUM. Each word in LD.OFS contains the offset in blocks from the beginning of the assigned physical disk to the start of the area on that physical disk assigned to that LD unit number.

LD.SIZ (LD.FLG+<4*Contents of LD.NUM>) The third translation table starts at offset LD.SIZ and contains one word for each LD unit number. The count of LD unit numbers is stored in LD.NUM. Each word in LD.SIZ contains the size in blocks of the area on the physical disk assigned to that logical disk unit.

LD.NAM (LD.FLG+<6*Contents of LD.NUM>) The fourth translation table starts at the label LD.NAM and contains four words for each LD unit number. The count of LD unit numbers is stored in LD.NUM.

The first word of each 4-word entry contains the Radix–50 2-character name of the physical disk that is assigned to that logical disk unit. That Radix–50 word must be the physical (not logical) device name without any unit number. DL is a valid physical device name; DK and DL1 are not valid.

The second, third, and fourth words of each entry contain the Radix–50 file name and file type assigned as the logical disk.

Accessing the Translation Tables

Before Version 5.5, the translation table access special function code SF.TAB (372) supported only eight units. The *wcnt* parameter for SF.TAB accepted two arguments, +1 to indicate a read of the table and -1 to indicate a write to the table. The size of each LD translation table was fixed at eight entries. Beginning with Version 5.5, if the LD handler on your system continues to support only eight logical disk units, you continue to read and write to the translation tables as before.

However, if the LD handler on your system supports more than eight units, the SF.TAB special function provides additional values for the *wcnt* parameter to support the extended device units. For LD handlers that implement the extended device-unit feature, you indicate both a read or write operation and the size of the table you are reading and writing by specifying a positive or negative numeric argument for the *wcnt* parameter. A positive numeric argument indicates a read operation of the specified number of words from the LD translation tables to the buffer. A negative number indicates a write operation of the specified number of words from the buffer to the translation tables. For example, a *wcnt* parameter argument of +16 reads 16 words, and an argument of –16 writes 16 words.

You can use the following procedure to read the translation tables from an LD handler that supports extended device units into a buffer and write the translation table from a buffer to LD. The procedure assumes you do not currently know (or want to verify) the number of entries in the table.

1. Entries are reserved in each translation table for the total number of logical disk units supported by the handler. The offset at which each table starts is determined by the number of supported units. Therefore, to determine the starting offset for each table within the four translation tables, you first determine how many logical disk units are supported by the handler.
2. You can determine the number of entries by doing a read SF.TAB to return the table entry LD.NUM. LD.NUM is the low byte of the second word in the table and contains the number of table entries. Therefore, for the *wcnt* parameter, supply the argument +2, and for the *buf* parameter, point to a 2-word buffer.

3. Once you have determined the number of supported logical disk units, you can use that value to perform read/write operations for the tables.
4. You can read the LD translation tables into memory by performing a single SF.TAB read operation. The number of words in the LD translation tables is two for the header (LD.ID plus LD.NUM), the value in LD.NUM for each of the first three tables and four times the value in LD.NUM for the fourth table:

   2+7*(LD.NUM)

   For example, if LD.NUM indicated 100_8 entries, the value to specify for *wcnt* to read the entire table would be $+450_{10}$. The *buf* parameter would point to a buffer of the same size.

   You could write the contents of the buffer to the LD handler by specifying the value -450_{10} for the *wcnt* parameter.

2.7.3 Other Bits Used by the LD Handler

The LD handler uses bit 4 (LDREL$) in CONFG2, monitor fixed offset 370. This bit is set whenever a handler is unloaded or released. The LD handler checks this bit to see if a handler assigned to an LD unit has been removed from memory since it was last used. If the bit is set, the LD handler sets bit 7 in all the entries in the HANDLR table, then clears the LDREL$ bit. When the LD handler begins to process an I/O request, the LD handler checks bit 7 for the requested LD unit. If bit 7 is set, the LD handler verifies that the handler for the disk assigned to that LD unit number is in memory, then clears the bit. The LD handler checks and clears bit 7 for a unit only when an I/O request is sent to that unit. Checking only when absolutely necessary ensures that the LD handler will not waste time verifying units that may never be used by a particular user program.

2.8 MM, MS, and MT (Magtape Handlers)

This section provides specific programming information for reel-type magnetic tape devices.

Magnetic tape (magtape) has a sequential (not random-access) file structure. There is no directory at the beginning of each tape. RT–11 magtape handlers support a file structure that is compatible with ANSI tape labels and format, giving you full access to the tape controller without concern for the specifics of the device. See *RT–11 Volume and File Formats Manual* for more information on the format of magtapes and tape labels.

NOTE

Support for RT–11 magtape file structure is compatible only among systems that support DEC and ANSI standards for tape labels and file formats. DOS-formatted tapes cannot be read or written.

See the *RT–11 Commands Manual* for SET command conditions for each of the magtape handlers. Those conditions can set the number of tracks, the density, the parity of the tape drive, and the CSR and vector addresses.

See also the *RT–11 Master Index* under *Magtape* and the individual magtape handlers for more information.

2.8.1 File Structure Module (FSM)

The File Structure Module (FSM) creates the file structure on magtapes written by the distributed magtape handlers. The FSM is a discrete module (FSM.MAC) that is assembled with the magtape hardware handlers when handlers are built; it is included in the distributed magtape handlers. The FSM uses a protocol that is understood by RT–11 utilities and described in the *RT–11 Volume and File Formats Manual*.

When you issue a call for a file-oriented operation, the monitor (and perhaps the USR) builds a queue element and passes it to the FSM. The FSM processes the operation by manipulating the magtape drive.

Through the system generation procedure, you can build each of the magtape handlers without the FSM; a hardware-only version of each handler. A hardware magtape handler is smaller and requires less memory, but does not contain any routines that define a file structure. It does contain routines that manipulate the magtape drive. See Section 2.8.7.

Further, unless you write your own file structure module that duplicates the functionality of the FSM, RT–11 utilities do not understand whatever protocol you use to manage the magtape.

Therefore, Digital recommends that you use the distributed magtape handlers (unless special circumstances indicate that a handler without the FSM is appropriate), since only the handlers that contain the FSM can communicate with the RT–11 system utility programs.

This section uses some magtape-specific abbreviations:

BOT beginning-of-tape

EOF end-of-file

EOT physical end-of-tape

LEOT logical end-of-tape
LEOT consists of an EOF1 label (which includes one tape mark) followed by two tape marks.

2.8.2 Compatibility of Magtape Operations with the FSM

As briefly explained above, the distributed magtape handlers contain the basic magtape hardware handler, which is assembled with a file structure module (FSM). As shown in the following tables, some magtape operations are intercepted by the FSM and some operations bypass the FSM and are processed directly by the basic magtape hardware handler.

Although the distributed magtape handlers can process all the magtape operations described in this section, performing hardware-oriented operations that are incompatible with the FSM disrupts the magtape's file structure and can make the magtape unsuitable for further file-oriented operations. In other words, to preserve the file-oriented nature of a magtape volume, perform only file-oriented operations on that volume or other operations that are compatible with the FSM.

The operations you can perform on a magtape can be divided into three classes:

- Operations that use the FSM. These are file-structured operations that require the distributed handlers.
- Operations that bypass the FSM but are compatible with the FSM. These are non-file-structured operations that the FSM understands.
- Operations that bypass the FSM and produce a magtape that is incompatible with the FSM. You can perform these operations with the distributed handlers but the resulting magtape is not compatible with the FSM or any RT–11 utilities.

The following tables list magtape operations and their compatibility with the FSM. The tables list where more information can be found for each operation.

Table 2–5: Magtape Operations That Use the FSM

Operation	Section	Description
FSM Search by Sequence Number	2.8.4.1	Search for a file on a magtape based on file's sequence number.
FSM Search by File Name	2.8.4.2	Search for a file on a magtape based on the file name.
.ENTER	2.8.4.3	Open a file.

Table 2–5 (Cont.): Magtape Operations That Use the FSM

Operation	Section	Description
.LOOKUP	2.8.4.4	Find a file.
.READx	2.8.4.5	Read from a file.
.WRITx	2.8.4.6	Write to a file.
.CLOSE	2.8.4.8	Close a file.
.PURGE	2.8.4.9	Delete entry and close channel.

Table 2–6: Magtape Operations That Are Compatible with the FSM

Operation	Code	Section	Description
NFS .LOOKUP	N/A	2.8.5.1	Open a channel to a device (non-file-structured .LOOKUP operation). Required before any special function.
SF.USR	354	2.8.5.2	After NFS .LOOKUP, can be used in the following ways: Perform asynchronous directory operations that do not require the USR. Emulate a file-structured .LOOKUP or .ENTER to gain access to a file for further special function operations.
SF.MRD	370	2.8.5.3	After initial NFS .LOOKUP and SF.USR, perform read operations of variable length blocks.
SF.MWR	371	2.8.5.4	After initial NFS .LOOKUP and SF.USR, perform write operations of variable length blocks.
SF.MST	367	2.8.5.7	After initial NFS .LOOKUP, stream TS05 (MS only).
.CLOSE	N/A	2.8.5.6	Close channel and make device available.

Table 2–7: Magtape Operations That Are Not Compatible with the FSM

Operation	Code	Section	Description
SF.MOR	372	2.8.6.1	Rewind and place drive off line.
SF.MRE	373	2.8.6.2	Rewind.
SF.MWE	374	2.8.6.3	Write with extended gap.
SF.MBS	375	2.8.6.4	Backspace.
SF.MFS	376	2.8.6.5	Forward space.
SF.MTM	377	2.8.6.6	Write tapemark.

Table 2–7 (Cont.): Magtape Operations That Are Not Compatible with the FSM

Operation	Code	Section	Description
NFS .READx	N/A		Obsolete. Non-file-structured read operation (use SF.MRD).
NFS .WRITx	N/A		Obsolete. Non-file-structured write operation (use SF.MWR).

2.8.3 Spacing Error Recovery

Any errors detected during spacing operations abort the recovery attempt, and generate a hard (position) error.

Magtape handlers both with or without the FSM perform the following operations if a read parity error is detected.

1. Backspaces over the block and rereads. When unsuccessful, the procedure is repeated until five read commands have failed.
2. Backspaces five blocks, spaces forward four blocks, then reads the record.
3. Repeats steps 1 and 2 eight times or until the block is read successfully.

The handler performs the following operations upon detection of a read after write (RAW) parity error.

1. Backspaces over one block.
2. Erases 3 inches of tape and rewrites the block. In no case is an attempt made to rewrite the block over the bad spot, since, even if the attempt succeeds, the block could be unreliable and cause problems later.
3. Repeats steps 1 and 2 if the read after write still fails. When 25 feet of erased tape have been written, a hard error is given.

2.8.4 Magtape Operations That Use the FSM

The following magtape operations, listed in Table 2–5, use the FSM. The distributed magtape handlers support these operations.

2.8.4.1 FSM Searching by Sequence Number

The FSM can search for files on tape based on their sequence number. It uses the relationship between the current tape position and the desired new position to find the desired file according to the following algorithm:

1. When the file sequence number for the desired file is greater than the number of the current position, the handler moves the tape forward.

   For example, if the tape is currently positioned at file sequence number 1, and the desired file is number 2, the tape moves forward from its position at the tape mark after file number 1 to the tape mark at the start of file number 2.

2. When the file sequence number for the desired file is less than the number of the current position, the handler optimizes its seek time by moving the tape backward or forward, depending on the location of the file. In practice, the handler almost always rewinds the tape and then searches forward.

   For example, assume the number of the current position is 2 and the desired file has sequence number 1. The tape leaves its position at the tape mark for file 2 and rewinds to the beginning of the volume. It then moves forward to the tape mark at the start of file 1. As another example, assume the current position is 9 and the desired file has sequence number 6. The tape rewinds to the beginning of the volume and the search proceeds in the forward direction.

If you release the handler through the UNLOAD command or the .RELEASE programmed request, the file position is lost. In this situation the tape moves backward until the handler locates BOT or a label from which it can determine the tape's position.

2.8.4.2 FSM Searching by File Name

The FSM can search for files on tape based on their file names. The routine to match file names uses an algorithm that enables the handler to recognize file names and file types used by other Digital operating systems. The FSM uses the file identifier field, translating the contents to a recognizable file name. This file name is matched to a file name stored in Radix–50 format. The format is as follows:

filnam.typ

filnam is a valid RT–11 file name left-justified in a six-character field. Unused character positions are not padded.

typ is a file type left-justified in a 3-character field.

The algorithm the handler uses is backward compatible across all versions of the operating system. RT–11 tapes can be detected by the presence of *RT11* in character positions 64 through 67 of the HDR1 label. The algorithm is as follows:

1. Clear the character count (CC).
2. Check the next character in the file name. If it is a dot, do the following:
   a. Mark a dot found.
   b. When CC < 6, insert spaces and increment the CC until it equals 6.
   c. When CC > 6, delete characters and decrement the CC until it equals 6.
3. If CC = 6 and if *RT11* is found in character positions 64 through 67 of the system code field, insert a dot in the translated name, mark the dot found, and increment CC.
4. Move the character into the translated file name and point to the next character.
5. Increment the CC.
6. When CC < 10_{10} go back to step 2.

7. Check the dot-found indicator. If no dot was found, back up four characters and insert .DAT for the file type.
8. Perform a character-by-character comparison between the desired file name and the file name that was just translated from the file identifier field in the HDR1 label. When they match exactly, consider the file found.

2.8.4.3 .ENTER Programmed Request

The .ENTER programmed request opens a file on a magtape by writing a HDR1 label and tape mark on the tape and leaving the tape positioned after the tape mark. The request initializes some internal tables and makes entries for the last block written and current block number. (The last block or file on tape is always the most recent one written.) Table 2–8 shows the sequence number values for .ENTER requests.

The .ENTER programmed request has the following format, with the *area, chan,* and *dblk* parameters as described in the *RT–11 System Macro Library Manual*. The *seqnum* parameter is described below.

Macro Call: .ENTER area,chan,dblk,,seqnum

Table 2–8: Sequence Number Values for .ENTER Requests

Seqnum Argument	File Name	Action Taken	Tape Position
>0	not null	Position at file sequence number and perform a .ENTER.	Found: ready to write. Not found: at LEOT; LEOT is an EOF1 label followed by two tape marks. LEOT is different from the physical end-of-tape.
0	not null	Rewind tape and search tape for file name. If found then give error. If not found then enter the file.	Found: before file. Not found: ready to write.
–1	not null	Position tape at LEOT and enter file.	Ready to write.
–2	not null	Rewind tape and search tape for file name. Enter file at found file or LEOT, whichever comes first.	Ready to write.
0	null	Perform a non-file-structured .LOOKUP.	Tape is rewound.

The .ENTER request returns the errors shown in Table 2–9.

Table 2–9: .ENTER Errors

Byte 52 Code	Meaning
0	Channel in use.

Table 2–9 (Cont.): .ENTER Errors

Byte 52 Code	Meaning
1	Device full. EOT was detected while writing HDR1. Tape is positioned after the first tape mark following the last EOF1 label on the tape. No such job exists (system job support only).
2	Device already in use. Magtape already has a file open on that unit.
3	File exists, cannot be deleted.
4	File sequence number not found. Tape is positioned the same as for device full.
5	Invalid argument error. A *seqnum* argument in the range –3 through –32767 was detected. A null file name was passed to .ENTER.

The .ENTER request issues a directory hard error if errors occur while entering the file.

2.8.4.4 File-Structured .LOOKUP Programmed Request

A file-structured .LOOKUP request finds a file by searching for a specific HDR1 label. Upon finding and reading the HDR1 label, the tape is positioned before the first data block of the file.

The .LOOKUP request has the following format, with the *area, chan,* and *dblk* parameters as described in the *RT–11 System Macro Library Manual*. The *seqnum* parameter argument values are shown in Table 2–10:

Macro Call: .LOOKUP area,chan,dblk,seqnum

Table 2–10: Sequence Number Values for File-Structured .LOOKUP Requests

Seqnum Argument	File Name	Action Taken	Tape Position
>0	null	Perform a file-structured .LOOKUP on the file sequence number.	Found: ready to read first data block. Not found: at LEOT.
0	not null	Rewind to the beginning of tape, then use file name to perform a file-structured .LOOKUP.	Found: ready to read first data block. Not found: at LEOT.
–1	not null	Do not rewind; perform a file-structured .LOOKUP for a file name.	Found: ready to read first data block from the current position. Not found: at LEOT.
>0	not null	Position at file sequence number and perform a file-structured .LOOKUP. If file name does not match file name given, return error.	Found: ready to read first data block. Not found: at the beginning of the file specified by the sequence number.

The file-structured .LOOKUP returns the errors shown in Table 2–11.

Table 2–11: .LOOKUP Errors

Byte 52 Code	Meaning
0	Channel in use.
1	File not found. Tape is positioned after the first tape mark following the last EOF1 on the tape.
2	Device in use. Magtape already has a file open.
5	Invalid argument error. A *seqnum* argument in the range –2 through –32767 was detected. A .LOOKUP request must have a positive sequence number.
6	Invalid unit number.

The .LOOKUP request issues a directory hard error if errors occur while entering the file.

2.8.4.5 .READx Programmed Requests

In this section, the term .READx refers to the .READ, .READC, and .READW group of programmed requests. Further, .READx requests are described for files that have been opened with the .ENTER and file-structured .LOOKUP requests.

The .READx requests read data from magtape in blocks of 512 bytes each. If a request is issued for fewer than 512 bytes, the handler reads the correct number of bytes. If the request is for more than 512 bytes, the handler performs the request with multiple 512-byte transfers (the last request may be for fewer than 512 bytes).

The .READx requests are valid in a file opened with a .LOOKUP request. They are also valid in a file opened with an .ENTER request, provided the block number requested does not exceed the last block written. (Exceeding the last block written returns code 0.)

If a tape mark is read, the routine repositions the tape so that another request causes the tape mark to be read again. When a .CLOSE is issued to a file opened by an .ENTER request, the tape position is left unchanged. Because magtape is sequentially accessed, a reposition in a file (a backup) without subsequently positioning to the end of the file (before a .CLOSE) causes data loss.

The guidelines for block numbers are as follows:

1. When a .LOOKUP is used (to search the file) with this request, the handler tries to position the tape at the indicated block number. When it cannot, a 0 (EOF code) is issued, and the tape is positioned after the last block on the file.
2. On an entered file, .READx checks to determine if the block requested is past the last block in the file. If it is, the tape is not moved and the 0 error code is issued.

The .READx request has the following format, with the *area, chan, buf, wcnt, blk* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .READx area,chan,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

Table 2–12 shows the errors the .READx requests return.

Table 2–12: .READx Errors

Byte 52 Code	Meaning
0	Attempt to read past a tape mark; also generated by block that is too large.
1	Hard error occurred on channel.
2	Channel not open.

2.8.4.6 .WRITx Programmed Requests

In this section, the term .WRITx refers to the .WRITE, .WRITC, and .WRITW group of programmed requests. Further, .WRITx requests are described for files that have been opened with the .ENTER and file-structured .LOOKUP requests.

The .WRITx requests write data to magtape in blocks of 512 bytes. If a request is issued for fewer than 512 bytes, the handler forces the writing of 512 bytes from the buffer address. If a request is issued for more than 512 bytes, the handler performs multiple 512-byte transfers.

The .WRITx requests are valid in a file opened with an .ENTER. Once a file is opened, .WRITx determines if the requested block is past the last block in the file. If it is, the tape is not moved and the 0 error code is issued.

The .WRITx request has the following format, with the *area, chan, buf, wcnt, blk* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .WRITx area,chan,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

Table 2–13 shows the errors the .WRITx requests return.

Table 2–13: .WRITx Errors

Byte 52 Code	Meaning
0	End-of-tape. The data for the last write was not written, but the previous block is valid. Also issued if the block number is too large.
1	Hard error occurred on channel.
2	Channel not open.

After a write operation, the rest of the tape is undefined (see Figure 2–5).

Figure 2–5: Operations Performed After the Last Block Written on Magtape

In example 1 in Figure 2–5, blocks A, B, and C are written on the tape with the head positioned in the gap immediately following block C. Any forward operation of the tape drive except by write commands (that is, write, erase gap and write, or write tape mark) yields undefined results due to hardware restrictions.

In example 2 in Figure 2–5, the head is shown positioned at BOT after a rewind operation so that successive read operations can read blocks A, B, and C. The head is left positioned as shown in example 3. Note that this is the same condition as shown in example 1, and all restrictions indicated in example 1 are applicable.

2.8.4.7 .CLOSZ, .DELETE, .GFxxx, .RENAME, and .SFxxx Programmed Requests

These requests are invalid operations on magtape, and any attempt to execute them returns an invalid operation code (code 2) in byte 52.

2.8.4.8 .CLOSE Programmed Request

The action of the .CLOSE request depends on how the file was opened.

- When a file is opened with an .ENTER request, the file is closed by writing a tape mark, an EOF1 label, and three more tape marks. In this operation, the tape is left positioned just before the second tape mark at LEOT. Note that the rest of the tape is no longer readable.

- When a file is opened with a file-structured .LOOKUP, the tape is positioned after the tape mark following the EOF1 label for that file.

The .CLOSE request has the following format, with the *chan* parameter as described in the *RT–11 System Macro Library Manual*:

Macro Call: .CLOSE chan

This request issues a directory hard error if a malfunction is detected. The error can be recovered with the .SERR request.

2.8.4.9 .PURGE Programmed Request

The action performed by a .PURGE request is determined by the following:

- If the magtape channel has been opened by a .ENTER request, a .PURGE request deletes the current entry by a series of BACKUP and WRITE-TAPE-MARK operations, leaving the magtape positioned just before the second tape mark at LEOT.
- If the magtape channel has been opened with a file-structured or non-file-structured .LOOKUP, the .PURGE request frees the unit table entry for the handler, closes the channel, and makes the handler available for other operations.

The .PURGE request has the following format, with the *chan* parameter as described in the *RT–11 System Macro Library Manual*:

Macro Call: .PURGE chan

2.8.5 Magtape Operations That Are Compatible with the FSM

The following magtape operations (as listed in Table 2–6), bypass the FSM but are compatible with the FSM. The distributed magtape handlers support these operations and a magtape that is manipulated by these functions is supported by RT–11 utilities.

2.8.5.1 Non-File-Structured .LOOKUP Programmed Request

You must issue a non-file-structured .LOOKUP request to open a channel to the device before starting any I/O operations. The non-file-structured .LOOKUP request causes the handler's hardware level to mark the drive busy so that no other channel can be opened to that drive until a .CLOSE is issued.

The .LOOKUP request has the following format, with the *area, chan,* and *dblk* parameters as described in the *RT–11 System Macro Library Manual*. The values for the *seqnum* parameter argument are described in Table 2–14:

Macro Call: .LOOKUP area,chan,dblk,seqnum

Table 2–14: Sequence Number Values for Non-File-Structured .LOOKUP Requests

Seqnum Argument	File Name	Action Taken	Tape Position
0	null	Perform a non-file-structured .LOOKUP.	Rewound.
–1	null	Perform a non-file-structured .LOOKUP.	Not moved.

Table 2–15 shows the errors that can be returned by the non-file-structured .LOOKUP request.

Table 2–15: Non-File-Structured .LOOKUP Errors

Byte 52 Code	Meaning
0	Channel in use; channel already open.
1	File not found; no such job.
2	Device in use. The drive being accessed is already attached to another channel.
5	Argument is invalid; for example, magtape file sequence number.
6	Invalid unit number.

2.8.5.2 Asynchronous Directory Operations (SF.USR), Code 354

SF.USR must be preceded by a non-file-structured .LOOKUP and can be used to perform two operations:

- SF.USR can perform asynchronous directory operations without the USR, which makes it useful for long tape searches. It is particularly useful in multi-job environments, because the search operation locks the USR during directly issued .ENTER and .LOOKUP requests.
- SF.USR allows an emulation of the .ENTER and file-structured .LOOKUP requests to be issued after a non-file-structured .LOOKUP assigns a channel to the magtape handler.

The special function SF.USR has the following format, with the *area* and *chan* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.USR,buf,,blk

SF.USR is the code 354 or the name SF.USR if the program has been assembled with the distributed file SYSTEM.MLB.

buf is the address of a 7-word block with the following format:

Word	Meaning
0–2	Radix–50 representation of the file name.
3	One of the following codes: 3 for .LOOKUP 4 for .ENTER
4	Sequence number value. See the corresponding sections for .LOOKUP or .ENTER for complete information on the interpretation of this value.
5,6	Reserved.

blk is the address of a 4-word error and status block used for returning .LOOKUP and .ENTER errors that are normally reported in byte 52. See Section 2.8.5.5. Only the first word of *blk* is used by this request. The other three words are reserved for future use and must be zero. If the value of *blk* is 0, no error information is returned. Figure 2–6 shows a programming example.

Figure 2–6: Asynchronous Directory Operation Example

```
.TITLE Asynchronous Directory Operation Example
        .ENABLE LC          ; Print lower case
        .NLIST  BEX         ; Don't list text storage
        .MCALL  .LOOKUP, .SPFUN, .CLOSE, .PRINT, .EXIT

        ; Definitions

        SF.USR = -20.       ; Asynchronous request
        LOOKUP =   3        ; Lookup code for async request
        ENTER  =   4        ; Enter code for async request
        CHAN   =   0        ; Use channel 0
        FNF    =   1        ; 1 = File not found error
        FSN    =   0        ; Use 0 as file sequence number

;Example assumes that magtape handler is loaded.
```

Figure 2–6 (continued on next page)

Figure 2–6 (Cont.): Asynchronous Directory Operation Example

```
 START:  .LOOKUP #AREA,#CHAN,#NFSBLK,#0     ; Open a channel
                              ; for the next request
        BCS     LOOKER        ; Branch if error occurred
        .SPFUN #AREA,#CHAN,#SF.USR,#COMBLK,#ERRBLK
                              ; Do a lookup
        BCC     FILFND        ; Branch if file found
        CMP     #FNF,ERRBLK ; File not found error?
        BEQ     NOTFND        ; Branch if yes
        MOV     #ASYERR,R0  ; No, some other error
        BR      CLOSE

LOOKER: MOV     #LOOERR,R0  ; NFS Lookup error
        BR      CLOSE

FILFND: MOV     #OK,R0        ; Report success
        BR      CLOSE

NOTFND: MOV     #FNFERR,R0     ; Report file not found
CLOSE:  .PRINT                ; Print error pointed to
                              ;  by R0
        .CLOSE #CHAN          ; Clean up...
        .EXIT                 ;  and return to monitor

;Data area

AREA:   .BLKW   5             ; EMT argument block
NFSBLK: .RAD50  /MT /         ; Use this to open
        .WORD  0              ;  magtape in non-file-
        .WORD  0              ;  structured mode
        .WORD  0

COMBLK: .RAD50  /FILNAMTYP/     ; This is the file name
                              ; we're looking for
        .WORD  LOOKUP         ; This is the asynch op
                              ;  code for lookup
        .WORD  FSN            ; This is file sequence
                              ;  number for the lookup
        .WORD  0,0            ; Reserved (must be 0)
ERRBLK: .WORD   1             ; Set first word non-0
        .WORD  0,0,0          ;  so errors return here

;Messages

LOOERR: .ASCIZ  /Non-file-structured lookup failed/
OK:     .ASCIZ  /File found, lookup successful/
FNFERR: .ASCIZ  /File not found/
ASYERR: .ASCIZ  /Error in asynchronous request/
        .EVEN
        .END   START
```

2.8.5.3 Read Physical Blocks (SF.MRD), Code 370

After an NFS .LOOKUP request (and optionally after an SF.USR), the SF.MRD request reads blocks of any size.

The special function SF.MRD has the following format, with the *area, chan, buf, wcnt,* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MRD,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MRD is the code 370 or the name SF.MRD if the program is assembled with the distributed file SYSTEM.MLB.

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the errors shown in Table 2–16. Additional qualifying information for these errors is returned in the first two words of the *blk* parameter argument status block. See Section 2.8.5.5.

Table 2–16: SF.MRD (Code 370) Errors

Byte 52 Code	First Word Code	Qualifying Information
EOF (Value = 0)	1	Tape before EOF only (tape mark detected).
	2	Tape before EOT only (no tape mark detected).
	3	Tape before EOF and EOT (tape mark detected).
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions).
	1	Tape drive not available.
	2	The controller lost the tape position.
	3	Nonexistent memory accessed.
	4	Tape is write locked.
	5	The last block read had more information. The MM handler returns (in the second status word) the number of words not read.
	6	A short block was read. The second status word contains the difference between the number of words requested and the number read.

2.8.5.4 Write Physical Blocks (SF.MWR), Code 371

After an NFS .LOOKUP request and optionally after an SF.USR, the SF.MWR request writes blocks of any size.

The special function SF.MWR has the following format, with the *area, chan, buf, wcnt* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MWR,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MWR is the code 371 or the name SF.MWR if the program is assembled with the distributed file SYSTEM.MLB.

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the errors shown in Table 2–17.

Table 2–17: SF.MWR (Code 371) Errors

Byte 52 Code	First Word Code	Qualifying Information
EOF (Value = 0)	1	Tape before EOF only (tape mark detected).
	2	Tape before EOT only (no tape mark detected).
	3	Tape before EOF and EOT (tape mark detected).
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions.)
	1	Tape drive not available.
	2	The controller lost the tape position.
	3	Nonexistent memory accessed.
	4	Tape is write locked.

NOTE

The TJU16 tape drive can return a hard error if a write request with a word count less than 7 is attempted.

2.8.5.5 Exception (Error and Status) Reporting

Special function requests report end-of-file and hard error conditions through byte 52 in the system communication area. You can also receive additional information about those two error conditions. You can specify an address in the special function's *blk* parameter that points to a 4-word error and status block which returns that information.

Specify #0 for *blk* if you do not want exception reporting.

Although all four words in the error and status block must be initialized to 0 before the first special function is called, only words 1 and 2 of the status block return information. Words 3 and 4 are reserved and not written and therefore need only be initialized once (remain as set to 0).

The meaning of the error and status block contents is tied to the contents of byte 52 in the system communications area. The program should therefore check the state of the carry bit and byte 52 before attaching importance to the contents of the error and status block.

End-of-File Condition Exception Reporting

Besides an actual EOF, the magtape handler's hardware level returns an end-of-file condition when the handler encounters an EOT, tape mark, or BOT. An end-of-file condition produces the following:

- Sets the carry bit and byte 52 is zero.
- The first word of the error and status block is shown in Table 2–18.
- The second word contains the number of blocks not spaced when a tape mark is detected during a spacing operation.

Table 2–18: End-of-File Qualifying Information

First Word	Meaning
1	Tape before EOF only (tape mark detected).
2	Tape before EOT only (no tape mark detected).
3	Tape before EOT and EOF (tape mark detected).
4	Tape before BOT (no tape mark detected).

Hard Error Condition Exception Reporting

A hard error condition:

- Sets the carry bit and byte 52 is 1.
- Returns in the first word the qualifying information shown in Table 2–19.

Table 2–19: Hard Error Qualifying Information

First Word	Meaning
0	No additional information (includes parity error and all others not listed below. Consult documentation for your particular tape drive for all possible error conditions.)
1	Tape drive not available.

Table 2–19 (Cont.): Hard Error Qualifying Information

First Word	Meaning
2	The controller lost the tape position. When this error occurs, rewind or backspace the tape to a known position.
3	Nonexistent memory was accessed.
4	Tape is write locked.
5	The last block read had more information. The MM handler returns (in the second status word) the number of words not read.
6	A short block was read. The second status word contains the difference between the number of words requested and the number of words read.

2.8.5.6 .CLOSE Programmed Request

The magtape handler at the hardware level accepts the .CLOSE request and causes the handler to mark the drive as available; the channel becomes free.

The .CLOSE request has the following format, with the *chan* parameter as described in the *RT–11 System Macro Library Manual*:

Macro Call: .CLOSE chan

2.8.5.7 Enabling 100ips Streaming on a TS05/TSU05/TSV05 (SF.MST), Code 367

The SF.MST special function places the TS05 drive in 100ips streaming mode.

The special function SF.MST has the following format, with the *area* and *chan* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MST,buf,,blk

SF.MST is the code 367 or the name SF.MST if the program is assembled with the distributed file SYSTEM.MLB.

buf is a word which enables or disables streaming.

If *buf* contains a 1, streaming is enabled.

If *buf* contains a 0, streaming is disabled.

blk is a pointer to a 4-word error block. (See Section 2.8.5.5.)

Streaming is automatically turned off when a .CLOSE is issued on a channel open on magtape, when an abort occurs, or if there is a magtape I/O error.

This special function is valid only for a TS05 using the MS handler. An SF.MST call is ignored if it is used with any other magtape handler or if it is used with the MS handler running a TS11 magtape.

If you want to run a TS05 in streaming mode, you must also use double-buffered I/O so that there is always a request pending in the magtape I/O queue. If there is not, there will be too much delay between I/O requests and the streaming will not work properly.

2.8.6 Magtape Operations That Are Not Compatible with the FSM

The magtape operations listed in Table 2–7 and described below bypass the FSM and are incompatible with the file structure produced by the FSM. The operations are direct hardware calls to the magtape handler. The distributed magtape handlers accept these operations, but a magtape that is manipulated by these functions is no longer ANSI-compatible or supported by RT–11 utilities.

When any of the following operations is called, the stored file sequence number and block number information are erased and are not reinitialized until a .CLOSE and another file-opening command have been performed. Note that the .CLOSE moves and, in the case of the file opened with .ENTER, writes the tape regardless of any commands that have been issued since the file was opened. When the file is closed, the magtape handler cannot write the size of the file because the file size is lost to the handler. It writes a zero in its place. The file sequence number field will be correct.

You initiate operations and use these special functions in the same manner as those that are compatible with the FSM:

1. Open a channel to the device by issuing a non-file-structured .LOOKUP.
2. You can optionally open a file on the magtape volume by issuing an SF.USR.
3. Issue the special functions to read, write, or position the magtape.
4. Close the channel.

If you are going to be using the operations in this section consistently, you should investigate performing a system generation and building a magtape handler that does not contain the FSM; a hardware-level-only handler. Such a handler is appropriate for the operations in this section and has a much smaller memory image. See Section 2.8.7 and the *RT–11 System Generation Guide* for information.

2.8.6.1 Rewinding and Going Off Line (SF.MOR), Code 372

This request is the same as rewind, except that it takes the tape drive off line and then rewinds to BOT. The handler is free to accept commands after the rewind is initiated.

The special function SF.MOR has the following format, with the *area, chan,* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MOR,,,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MOR is the code 372 or the name SF.MOR if the program is assembled with the distributed file SYSTEM.MLB.

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the same error code and qualifying information as the rewind request.

2.8.6.2 Rewinding (SF.MRE), Code 373

The SF.MRE request rewinds the tape to BOT. The MT and MM handlers cannot accept other requests until the rewind operation is complete; the MS handler can.

The special function SF.MRE has the following format, with the *area, chan,* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MRE,,,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MRE is the code 373 or the name SF.MRE if the program is assembled with the distributed file SYSTEM.MLB.

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the error shown in Table 2–20.

Table 2–20: SF.MRE (Code 373) Errors

Byte 52 Code	First Word Code	Qualifying Information
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions).
	1	Tape drive not available.

2.8.6.3 Writing with Extended Gap (SF.MWE), Code 374

This request permits you to write on tapes that have bad spots. The call syntax is identical to the SF.MWR request except for its function code, which is 374. The errors are explained in Table 2–21.

Table 2–21: SF.MWE (code 374) Errors

Byte 52 Code	Meaning
0	The EOT marker has been detected.
1	Hard error occurred on channel.
2	Channel not open.

Additional qualifying information for these errors is returned in the first two words of the status block. See Section 2.8.5.5.

2.8.6.4 Spacing Backward (SF.MBS), Code 375

The SF.MBS request spaces the magtape backward block-by-block or until a tape mark is detected.

You should note that because magtape is sequentially accessed, an SF.MBS operation in a file without a subsequent positioning to the end of the file (before a .CLOSE) causes data loss.

The special function SF.MBS has the following format, with the *area, chan, wcnt* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MBS,,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MBS is the code 375 or the name SF.MBS if the program is assembled with the distributed file SYSTEM.MLB.

wcnt is the number of blocks to space past (must not exceed 65534_{10}).

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the errors shown in Table 2–22.

Table 2–22: SF.MBS (Code 375) Errors

Byte 52 Code	First Word Code	Qualifying Information
EOF (Value = 0)	1	Tape before EOF only (tape mark detected).
	2	Tape before EOT only (no tape mark detected).
	3	Tape before EOF and EOT (tape mark detected).
	4	Tape before BOT (no tape mark detected). The second word in the status block contains the number of blocks requested to be spaced *wcnt*, minus the number of blocks spaced if a tape mark or BOT is detected. Otherwise, its value is not defined.
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions).
	1	Tape drive not available.
	2	The controller lost the tape position.

2.8.6.5 Spacing Forward (SF.MFS), Code 376

The SF.MFS request spaces the magtape forward block-by-block or until a tape mark is detected. When a tape mark is detected, the handler reports it along with the number of blocks not skipped. These commands can be used to issue a space-to-tape-mark command by passing a number greater than the maximum number of blocks on a tape. The tape is left positioned after the tape mark or the last block passed. The two spacing requests have the following forms.

The special function SF.MFS has the following format, with the *area, chan* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MFS,,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MFS is the code 376 or the name SF.MFS if the program is assembled with the distributed file SYSTEM.MLB.

wcnt is the number of blocks to space past (must not exceed 65534_{10}).

blk is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the errors shown in Table 2–23.

Table 2–23: SF.MFS (Code 376) Errors

Byte 52 Code	First Word Code	Qualifying Information
EOF (Value = 0)	1	Tape at EOF only (tape mark detected).
	2	Tape at EOT only (no tape mark detected).
	3	Tape at EOF and EOT (tape mark detected). The second word in the status block contains the number of blocks requested to be spaced (*wcnt*), minus the number of blocks spaced if a tape mark or BOT is detected. (A tape mark is counted as a block.) Otherwise, its value is not defined. The tape will be positioned after the tape mark on forward spacing and before the tape mark on backward spacing.
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions).
	1	Tape drive not available.
	2	The controller lost the tape position.

NOTE

Due to hardware restrictions, Digital recommends that no forward space commands be issued if the reel is positioned past the EOT marker.

2.8.6.6 Writing a Tape Mark (SF.MTM), Code 377

The SF.MTM request writes a tape mark.

The special function SF.MTM has the following format, with the *area, chan* and optional *crtn, BMODE=str*, and *CMODE=str* parameters as described in the *RT–11 System Macro Library Manual*:

Macro Call: .SPFUN area,chan,#SF.MTM,,,blk[,crtn][,BMODE=str][,CMODE=str]

SF.MTM is the code 377 or the name SF.MTM if the program is assembled with the distributed file SYSTEM.MLB.

blk	is the address of a 4-word error and status block used for returning the exception conditions. See Section 2.8.5.5.

This request returns the errors shown in Table 2–24. Additional qualifying information for these errors is returned in the first two words of the *blk* argument status block. See Section 2.8.5.5.

Table 2–24: SF.MTM (Code 377) Errors

Byte 52 Code	First Word Code	Qualifying Information
EOF (Value = 0)	1	Tape before EOF only (tape mark detected).
Hard error (Value = 1)	0	No additional information (consult documentation for your particular tape drive for all possible error conditions).
	1	Tape drive not available.
	2	The controller lost the tape position.
	4	Tape is write locked.

2.8.7 Hardware Magtape Handler

The hardware magtape handlers are identical to the distributed handlers except they are not built with the FSM. Therefore, the hardware magtape handlers accept only hardware requests. These are applicable in I/O operations where no file structure exists. Any file structure request you make to the hardware handler results in a monitor directory I/O error. The hardware handler is a subset of the file structure magtape handler. It can perform I/O operations on physical blocks, position the tape, and recover from errors.

Any file-structured request causes the hardware handler to issue a hard error. The hardware handler accepts only the non-file-structured .LOOKUP, .CLOSE, or special function requests.

If you do not need the file structure support, use the hardware handlers. You must perform a SYSGEN (see the *RT–11 System Generation Guide*) to get the hardware magtape handlers, then you must rename them in order to use them. Use a series of monitor commands similar to the following, which replace the file structure MS handler with the hardware MS handler.

1. Remove the distributed handler:

   ```
   .REMOVE MS  RET
   ```

2. Save the distributed handler:

   ```
   .RENAME/SYS MS[X].SYS MS[X]FS.SYS  RET
   ```

3. Replace the distributed handler with the hardware handler you built during SYSGEN:

   ```
   .RENAME/SYS MS[X]HD.SYG MS[X].SYS  RET
   ```

4. Install the hardware handler:

```
.INSTALL MS RET
```

2.8.8 Transporting Tapes to RT–11

RT–11 can read files written on other computer systems that support the ANSI standard labels. The following sections give a few examples of how to write ANSI tapes on some common Digital PDP–11 operating systems. Keep in mind that there are other factors involved in addition to the label and format compatibility, including density, parity, and number of tracks. Consult the appropriate system documentation for complete information on using magtapes under the different operating systems. (See the *RT–11 Volume and File Formats Manual* and the *RT–11 System Utilities Manual* for information on transporting tapes from RT–11 to other systems.)

2.8.8.1 From RSTS/E

RSTS/E supports two types of magtape format, DOS–11 and ANSI. In the following examples, *dd* represents the magtape handler name. To ensure that an ANSI file structure is written, issue the following commands:

Examples

1. `ASSIGN ddn: .ANSI`

   Allocates the device to the job and ensures that an ANSI file structure is used.

2. ```
   RUN $PIP
   ddn:xxxxxx/ZE
   ```

   PIP initializes the tape; *xxxxxx* is the volume ID.

3. `Really zero ddn:? YES`

   PIP prompts before initializing the tape.

4. `PIP ddn:=TEST1.MAC,TEST2.MAC`

   PIP copies files to the tape.

5. `DEASSIGN ddn:`

   Deallocates the device.

2.8.8.2 From RSX–11M

RSX–11M needs the following commands to access a magtape:

Examples

1. `ALL ddn:`

   Allocates a drive.
   ```

2. `INI ddn:RT11`

   Initializes the tape and gives the name *RT11* as the volume identification.

3. `MOU ddn:RT11`

   Mounts the tape volume.

4. `PIP ddn:=[13,14]TEST1.MAC,TEST2.MAC`

   Copies files to the tape.

5. `DMO ddn:RT11`

   Dismounts the tape volume.

6. `DEA ddn:`

   Deassigns the drive.

2.8.8.3 From RSX–11D and IAS

Use the following commands to write an ANSI tape on RSX–11D or IAS:

Examples

1. `INI ddn:RT11`

   Initializes the tape and gives the name *RT11* as the volume identification.

2. `MOU ddn:RT11`

   Mounts the tape volume.

For RSX–11D, use PIP to write files to the tape; for IAS, use the COPY command.

Examples

1. `DMO ddn:RT11`

   Dismounts the tape volume.

The contents of files written under the RSX–11D, RSX–11M, and IAS systems do not necessarily correspond to those types of data files under RT–11. For example, under RT–11, text files consist of stream ASCII data (carriage return and line feed characters are embedded in the text); the other operating systems use a different type of character storage. Be sure to pay attention to the contents of the files you need to transfer.

When you write files to be read under RT–11, the only valid block size the utility programs use is 512 characters per block. However, the DIR program will list the directory of any ANSI compatible tape.

2.8.8.4 From VMS

Creating a magtape on a VAX processor running the VMS operating system for subsequent transfer to a PDP–11 running RT–11 is described in the *RT–11 Volume and File Formats Manual*. Look there for the procedure.

2.8.9 Seven-Track Magnetic Tape

Seven-track tapes contain six data tracks and one parity track, so a maximum of six data bits can be contained in one tape character. With seven-track tapes, the MT handler operates in either six-bit mode or core dump mode.

Six-bit mode is not compatible with the data normally created by PDP–11 systems; it is provided for transferring data to or from other systems. In addition, file structure operations cannot be performed in this mode. With the density set at 200 or 556 bpi, the magtape always operates in six-bit mode. When reading in six-bit mode, the handler places each six-bit tape character right-justified in a PDP–11 byte; the high-order two bits of the byte are set to 0. When writing in six-bit mode, the handler writes the low-order six bits of a PDP–11 byte as the six data bits of a tape character; the high-order two bits of the PDP–11 byte are not transferred or affected.

Core dump mode is compatible with PDP–11 systems. At 800 bpi, seven-track tape transfers can occur in either six-bit mode (SET MT: DENSE=807) or core dump mode (the default). Figure 2–7 illustrates the differences between six-bit mode and core dump mode.

In core dump mode, each PDP–11 byte is split into two tape characters. In writing to the tape, the handler writes the low-order four bits of a PDP–11 byte as the low-order four bits of the first tape character and the high-order four bits of the PDP–11 byte as the low-order four bits of the next tape character. The high-order two bits of each tape character are set to 0.

In reading from the tape, the reverse process occurs. The low-order four bits of the first tape character become the low-order four bits of the PDP–11 byte; the low-order four bits of the next tape character become the high-order four bits of the PDP–11 byte.

The high-order two bits of each tape character are not involved in the transfer, although they are included in the parity calculation. Thus, in core dump mode, the actual number of tape characters read or written is twice the number of PDP–11 bytes requested to be transferred; this conversion is performed by the magtape controller.

Figure 2–7: Seven-Track Tape

2.9 MU (TMSCP Magtape Handler)

This section provides specific programming information for TMSCP magtapes.

The MU handler supports magtape systems that use the tape mass storage communication protocol (TMSCP).

NOTE

The MU handler contains the same basic structure and provides the same support for programmed requests and special functions as described in Section 2.8 except as explicitly stated in this section. Therefore, this section describes only how the MU handler is different from the MM, MS, and MT handlers.

2.9.1 Support for Special Functions

The following special functions are either not supported by the reel-type magtape handlers or are supported in a different manner.

The SF.MTB and SF.BYP special functions are not affected by the presence (or absence) of the File Stucture Module (FSM), as they are not concerned with operations on magtape volumes. Rather, they are conerned with data structures within the handler itself or the handler's controller.

Code	Name	Function
352	SF.MTB	Magtape data table access
	SF.TRD	*wcnt* argument for a read from the table; specified with a +1
	SF.TWR	*wcnt* argument for a write to the table; specified with a –1
360	SF.BYP	Direct TMSCP access; special function bypass
374	SF.MWE	Not Supported; writes with extended file gap executes as a write (SF.MWR) operation

2.9.1.1 TMSCP Translation Tables (SF.MTB), Code 352

Whenever an I/O request is passed to the MU handler, MU uses the RT–11 unit number as an index into the translation tables. MU then extracts the TMSCP unit number and port that have been assigned to that RT–11 unit, and uses the information to access the proper magtape drive.

You can read or write (modify) the memory-resident contents of the translation tables by using SF.MTB.

Size of the Translation Tables

The size of the translation tables is determined by the number of device units supported by DU. The distributed MU supports one unit; you can build an MU that supports up to four units. You can determine the number of supported units for a particular handler by reading the MU.NUM field, as explained further.

Structure of the Translation Tables

As shown in Tables 2–25 and 2–26, the MU unit translation tables consist of a table header followed by table entries. The header starts at offset MU.ID, which is a word containing the Radix–50 value for the characters MU.

The MU.ID offset is followed by MU.NUM. The low byte of MU.NUM contains the number of entries in the table (and therefore the number of supported units). The high byte of MU.NUM is reserved.

The next offset is MU.ENT, which contains a pointer to the first table entry.

Table 2–25: TMSCP (MU) Translation Table Header

Offset	Name	Meaning
0	MU.ID	Radix–50 value for characters MU
2	MU.NUM	Byte containing number of entries in table
3		Reserved
4	MU.ENT	The offset of the first table entry

Each table entry is 4 bytes, and Digital recommends you use the symbol MU.ESZ to represent the 4-byte size of each entry.

Table 2–26: TMSCP (MU) Translation Table Entry

Offset	Name	Meaning
0	MU.UNI	Physical TMSCP unit number. The symbol MU$Ux=nnnnnn is the initial value for the translation table when the handler is assembled. In the symbol, *x* is the octal RT–11 MU unit number (0–3) and *nnnnnn* is the TMSCP unit number. The SET MUx UNIT=nnnnnn command can subsequently change the value.
2	MU.JOB	Byte containing the number of the job connected to this TMSCP unit.
3	MU.POR	Byte containing the TMSCP port (controller) number. The symbol MU$Ox=nnn is the initial value for the translation table when the handler is assembled. In the symbol, *x* is the octal RT–11 MU unit number (0–3) and *nnn* is the TMSCP port number. The SET MU PORT=nnn command can subsequently change the value.
4	MU.ESZ	Size of an entry (4 bytes)

Accessing the Translation Tables

Special function SF.MTB can read or write the TMSCP translation tables. Whether a read or write operation is performed is determined by the *wcnt* argument. Specify +1 (SF.TRD) for *wcnt* to read the tables; –1 (SF.TWR) to write the tables.

The translation tables are read from or written to a buffer, which is pointed to by the *buf* parameter.

2.9.1.2 Special Function Bypass (SF.BYP), Code 360

Special function SF.BYP bypasses all unit number translation and allows direct access to the TMSCP port. For MU, SF.BYP (direct TMSCP access) serves the same purpose as the DU handler's SF.BYP (direct MSCP access).

The request syntax and parameter argument definitions for SF.BYP are as follows:

Macro Call: .SPFUN area,chan,#SF.BYP,buf,wcnt,blk

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

SF.BYP is code 360 or the name SF.BYP if the program has been assembled with the distributed module SYSTEM.MLB.

buf is the address of the 52_{10}-word TMSCP area.

wcnt when nonzero, is the virtual address of a data buffer to send to the handler. That virtual address is translated to a physical address and placed in the buffer of the TMSCP area.

when zero, the buffer address in the TMSCP area is not altered.

blk indicates whether the handler should perform retries:

1 = specifies retries

0 = specifies no retries

The buffer address in special function SF.BYP must point to a 52-word area in the user's job. The first 26 words are used to hold:

- A response packet length in bytes
- A virtual circuit identifier
- An end packet when the command is complete

The second 26 words are set up by the caller and contain:

- A length word (length of command)
- A virtual circuit identifier (must have octal 1 (001) in high byte)
- A valid TMSCP command (48_{10}-byte command buffer)

Except for port initialization, the user program must do all command packet sequencing, error handling, and reinitialization when the bypass operations are complete.

2.9.2 Unit Support, CSR and Vectors

The distributed MU handler supports one unit. Using the system generation procedure, you can build an MU handler that supports up to four units. Each unit requires a separate controller and you can only boot RT–11 from unit MU0, which must be installed at CSR address 774500 and vector address 260. The addresses for MU1 through MU3 float; they depend on what other devices are on the bus. The default CSR and vector addresses are as follows:

CSR	Vector
774500	260
774504	340
774510	344
774514	350

2.10 NL (Null Handler)

The null handler accepts all read and write requests. On output operations, this handler acts as a data sink. When a program calls NL, the handler returns immediately to the monitor indicating that the output is complete. The handler returns no errors and causes no interrupts. On input operations, NL returns an immediate EOF indication for all requests; no data is transferred. Hence, the contents of the input buffer are unchanged.

2.11 NC, NQ, NU (Ethernet Handlers)

RT–11 includes three Ethernet handlers that provide support for Ethernet class controllers. The NC Ethernet handler supports the DECNA controller for CTI Bus-based processors. The NQ Ethernet handler supports the DELQA and DEQNA Ethernet controllers for Q-bus processors. The NU Ethernet handler supports the DELUA and DEUNA controllers for UNIBUS processors.

Each handler supports only one controller and a maximum of eight units. These unit numbers are used as a logical connection between a user program and an address /protocol pair to be recognized by the Ethernet hardware.

2.11.1 Restrictions

Observe the following Ethernet handler restrictions:

- The handlers run only under mapped monitors.
- The handlers cannot be fetched and must be loaded.
- Programs that call the Ethernet handlers must be written to perform with the following elements in the order indicated:
  1. Use the .LOOKUP programmed request to open a channel to the device unit.
  2. Allocate the unit using .SPFUN 200.
  3. Perform the Ethernet operation or operations.
  4. Deallocate the unit using .SPFUN 200.
  5. Use the .CLOSE programmed request to close the channel to the specified device unit.

2.11.2 Support for Special Functions

The Ethernet handlers support the following special functions. The special function names are from the .NALDF macro in the distributed file SYSTEM.MLB.

Code	Name	Section	Function
200	SF.NAL	2.11.2.1	Allocate/Deallocate unit
201	SF.PRO		Reserved
202	SF.NPR	2.11.2.2	Enable/Disable protocol type
203	SF.NMU	2.11.2.3	Enable/Disable multicast address
204	SF.NWR	2.11.2.4	Transmit Ethernet frame
205	SF.NRD	2.11.2.5	Receive Ethernet frame

Successful completion of a .SPFUN request clears the carry bit. Completion with error sets the carry bit, and the status word in the buffer contains an error code.

2.11.2.1 Allocate/Deallocate Unit (SF.NAL), Code 200

The allocate unit special function allocates a unit of the Ethernet handler for a job's exclusive use.

The deallocate unit special function deallocates the unit so it can be used by another job.

2.11.2.1.1 Allocate Unit

The following is the form of the special function allocate unit:

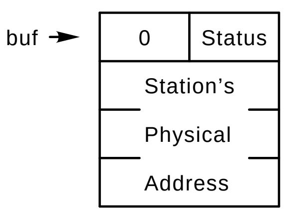
Macro Call: .SPFUN area,chan,#SF.NAL,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

SF.NAL is code 200 or the name SF.NAL if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 4-word buffer containing the status word and space for the station's physical address. The buffer contents are returned by the allocate unit special function.



The high byte of the status word contains a 0. Allocate unit returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success
2	Controller error while attempting to initialize the network interface (controller).
3	No resources (unit in use).
11	Reserved.

wcnt is #0.

blk is #1.

2.11.2.1.2 Deallocate Unit

The following is the form of the special function deallocate unit:

Macro Call: .SPFUN area,chan,#SF.NAL,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

SF.NAL is code 200 or the name SF.NAL if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 1-word buffer containing the status word.

buf →	0	Status

The high byte of the status word contains a 0. Deallocate unit returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).
11	Unit still active.

wcnt is #0.

blk is #0.

2.11.2.2 Enable/Disable Protocol Type (SF.NPR), Code 202

The enable protocol type special function adds a protocol type to the list of those to be recognized by the unit. Only one protocol type can be specified for each unit. At least one protocol type must be enabled to receive Ethernet frames.

The disable protocol type special function removes the protocol type from the list of those recognized by the unit.

2.11.2.2.1 Enable Protocol Type

The following is the form of the special function enable protocol type:

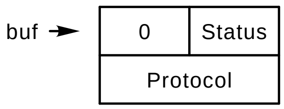
Macro Call: .SPFUN area,chan,#SF.NPR,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

func is code 202 or the name SF.NPR if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 2-word buffer that contains the status word followed by the protocol type word.



The high byte of the status word contains a 0. Enable protocol type returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).
3	No resources (unit's protocol table is full).
6	Reserved.
10	Protocol type in use.

The protocol type is specified by the user.

wcnt is #0.

blk is #1.

2.11.2.2.2 Disable Protocol Type

The following is the form of the special function disable protocol type:

Macro Call: .SPFUN area,chan,#SF.NPR,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

SF.NPR is code 202 or the name SF.NPR if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 2-word buffer that contains the status word, followed by the protocol type word.

buf →

0	Status
Protocol	

The high byte of the status word contains a 0. Disable protocol returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).

wcnt is #0.

blk is #0.

2.11.2.3 Enable/Disable Multicast Address (SF.NMU), Code 203

The enable multicast address special function adds a multicast address to the list of those to be recognized by that unit. You need not specify the unit's physical or broadcast address. RT–11 supports only one multicast address per handler unit.

The disable multicast address special function removes a multicast address from the list of those to be recognized by the unit.

2.11.2.3.1 Enable Multicast Address

The following is the form of the special function enable multicast address:

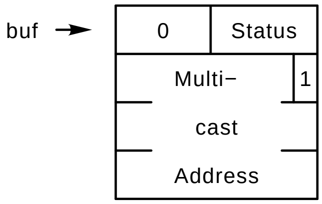
Macro Call: .SPFUN area,chan,#SF.NMU,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

func is code 203 or the name SF.NMU if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 4-word buffer that contains the status word, followed by the 3-word multicast address. The low-order bit of the first address word should be a 1.



The high byte of the status word contains a 0. Enable multicast address returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).
3	No resources (unit's address table is full, or hardware address table is full).

wcnt is #0.

blk is #1.

2.11.2.3.2 Disable Multicast Address

The following is the form of the special function disable multicast address:

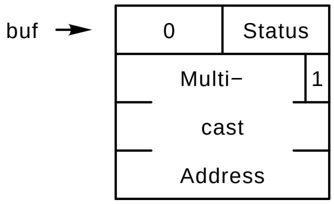
Macro Call: .SPFUN area,chan,#SF.NMU,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

func is code 203 or the name SF.NMU if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a 4-word buffer that contains the status word, followed by the 3-word multicast address. The low-order bit at the first address word should be a 1.



The high byte of the status word contains a 0. Disable multicast address returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).

wcnt is #0.

blk is #0.

2.11.2.4 Transmit Ethernet Frame (SF.NWR), Code 204

The special function transmit Ethernet frame transmits the Ethernet frame pointed to in the *buf* parameter argument. If the source address field of the frame is nonzero, it is kept and used. If the source field of the frame is zero, the unit's physical address is inserted in the source field before transmission.

The following is the form of the special function transmit Ethernet frame:

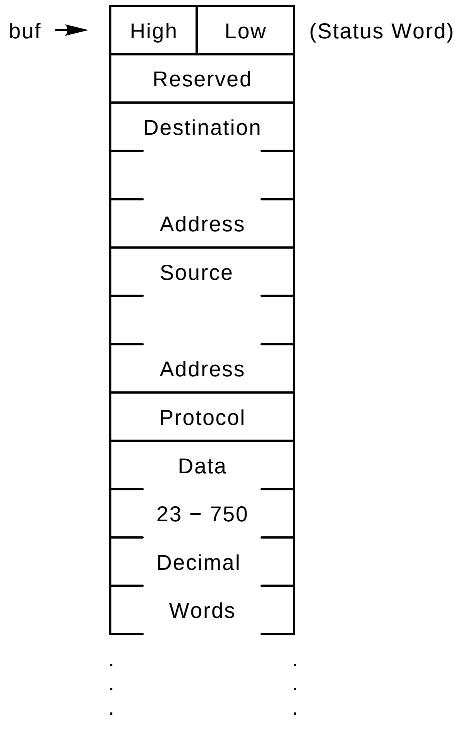
Macro Call: .SPFUN area,chan,#SF.NWR,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area is the address of a 6-word EMT argument block.

chan is a channel number in the range 0 to 376_8.

func is code 204 or the name SF.NWR if the program is assembled with the distributed file SYSTEM.MLB.

buf is the address of a variable-size buffer containing a word for returning status, a reserved word, and up to 757_{10} words comprising the Ethernet frame to be transmitted.



Transmit Ethernet frame returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).
13	Transmit failed. When status code 13 is returned in the low byte of the status word, transmit Ethernet frame returns one of the following octal status subcodes in the high byte of the status word: 1 = Invalid frame length. 2 = Excessive collisions. 3 = Carrier check failed.

wcnt	is determined by the variable size of the user buffer (including the status and reserved words). The packet size (including the status and reserved words) can vary between 32_{10} and 759_{10} words.
blk	is #0.

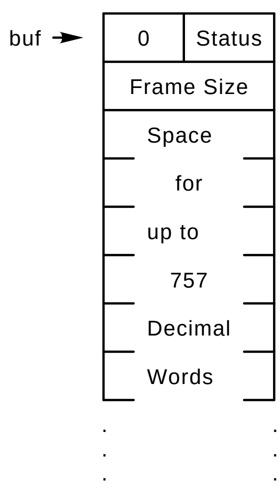
2.11.2.5 Receive Ethernet Frame (SF.NRD), Code 205

The receive Ethernet frame special function returns the next Ethernet packet with the desired unit address and protocol type to the buffer. The function does not return Ethernet frames that are received with errors.

The following is the form of the special function receive Ethernet frame:

Macro Call: .SPFUN area,chan,#SF.NRD,buf,wcnt,blk[,crtn][,BMODE=str][,CMODE=str]

area	is the address of a 6-word EMT argument block.
chan	is a channel number in the range 0 to 376_8.
func	is code 205 or the name SF.NRD if the program is assembled with the distributed file SYSTEM.MLB.
buf	is the address of a variable-size buffer containing a word for returned status, a word for returned fram size, and up to 757_{10} words to receive the Ethernet frame. The buffer contents are returned by the receive Ethernet frame special function.



The high byte of the status word contains a 0. The receive Ethernet frame special function returns one of the following octal status codes in the low byte of the status word:

Code	Meaning
0	Success.
1	Unknown unit. The specified unit was not opened by the job issuing the request.
2	Controller error while attempting to initialize the network interface (controller).

wcnt is the size of the user buffer including the status and frame size words. The maximum value allowed for the argument is 759_{10}; the minimum is 32_{10}.

blk is #0.

2.11.3 Example of Allocating an Ethernet Unit

The following example allocates a unit of the Ethernet handlers.

```
        CONFG2  = 370                   ;Config word 2
                                        ; (RMON fixed offset)
        PROS$   = 020000                ;RT is running on a PRO-3xx
        BUS$    = 000100                ;Q-bus/UNIBUS processor
                .
                .
                .
        .GVAL   #AREA,#CONFG2           ;Get contents of Config word 2
        MOV     #<^RNC >,DBLK         ;Assume PRO
        BIT     #PROS$,R0               ;Correct assumption?
        BNE     10$                     ;yes...
        MOV     #<^RNQ >,DBLK         ;No, so assume Q-bus
        BIT     #BUS$,R0                ;Correct assumption?
        BNE     10$                     ;yes...
        MOV     #<^RNU >,R0           ;Nope, must be
                                        ; UNIBUS after all
10$:    .GTJB   #AREA,#JOBDAT           ;Get info on this job
        MOV     JOBDAT,R0               ;R0 = job number (*2)
        ASR     R0                      ;Convert to job number 0-7
        ADD     #<^R  0>,R0           ;Make it final RAD50 digit
        ADD     R0,DBLK                 ; and add it to
                                        ; the device name
        .LOOKUP #AREA,#0,#DBLK          ;Open a channel to Ethernet
                .
                .                       ;.LOOKUP error processing
                .
        .SPFUN  #AREA,#0,#200,#BUFFER,#0,#1
                                        ;Allocate the unit to this job
                .
```

```
                .                    ;.SPFUN error processing
                .
AREA:   .BLKW   3
JOBDAT: .BLKW   12.
DBLK:   .WORD   0,0,0,0
BUFFER: .BLKW   4
                .
                .
                .                    ;END OF EXAMPLE
```

2.12 PI (CTI Bus-Based Processor Interface System Support Handler)

This section contains specific information about the PI system support handler and using RT–11 with CTI Bus-based processors. PI is called a *system support handler* because RT–11 requires PI to provide certain necessary connections with the computer hardware. At bootstrap time, the monitor loads PI before binding with the system device handler file on the system volume.

2.12.1 Support for Special Functions

The PI handler supports the following special functions which are used only with the GIDIS graphics package, as described in the *RT–11 System Subroutine Library Manual*:

Code	Name	Action
371	SF.PWR	Send command packet to GIDIS.
370	SF.PRD	Get status from GIDIS.

2.12.2 PI Keyboard Support

PI supports the keyboard in normal mode or function key mode.

2.12.2.1 Normal Mode

PI supports the following keys in normal mode:

- All keys on the main keypad.
- All keys on the numeric keypad.
- Cursor control (arrow) keys on the editing keypad.
- The following special function keypad keys: HOLD SCREEN (F1), PRINT SCREEN (F2), SETUP (F3), ESCAPE (F11), BACK SPACE (F12), and LINE FEED (F13).

  PRINT SCREEN (F2) prints a copy of the text from your terminal screen directly on your printer. PRINT SCREEN cannot be used to print graphics. You must be running the transparent spooling package (SPOOL) under a mapped monitor to use PRINT SCREEN.

  SETUP (F3) clears a locked keyboard and turns off the WAIT light when pressed. Note that the SETUP key has nothing to do with the setup utility.

The following keys do not function in normal mode:

- Special function keys F4 through F10, F14, HELP (F15), DO (F16), and F17 through F20.
- Editing keypad keys FIND, INSERT HERE, REMOVE, SELECT, PREV SCREEN, and NEXT SCREEN. Editing functions under RT–11 use the numeric keypad (see the *PDP–11 Keypad Editor User's Guide*.)

2.12.2.2 Function Key Mode (DECFKM)

Programs written for the PI handler can place the terminal in function key mode. In function key mode, each special function key sends an assigned control sequence to the processor. The control sequence is not assigned a specific function, but software can be programmed to recognize the control sequence.

A program places the terminal in function key mode by sending the 7-bit escape sequence:

```
ESC[?39h (transmitted as octal 033 133 077 063 071 150)
```

A program returns the terminal to normal key mode by sending the 7-bit escape sequence (note the lower-case l (?39l)):

```
ESC[?39l (transmitted as octal 033 133 077 063 071 154)
```

The following table lists control sequences for the special function keys:

Key	Control Sequence	Key	Control Sequence
F1	ESC[11~	DO (F16)	ESC[29~
F2	ESC[12~	F17	ESC[31~
F3	ESC[13~	F18	ESC[32~
F4	ESC[14~	F19	ESC[33~
F5	ESC[15~	F20	ESC[34~
F6	ESC[17~	COMPOSE CHARACTER	ESC[10~
F7	ESC[18~	FIND	ESC[1~
F8	ESC[19~	INSERT HERE	ESC[2~
F9	ESC[20~	REMOVE	ESC[3~
F10	ESC[21~	SELECT	ESC[4~
F11	ESC[23~	PREV SCREEN	ESC[5~
F12	ESC[24~	NEXT SCREEN	ESC[6~
F13	ESC[25~		
F14	ESC[26~		
HELP (F15)	ESC[28~		

2.12.3 Video Terminal Support

PI supports the CTI Bus-based processor's video terminal in the following manner:

2.12.3.1 Advanced Video Option Emulation

The PI handler supports a limited emulation of the VT100 implementation of the advanced video option, and uses the same escape sequences as the VT100 terminal. The limited emulation supports all VT100 character renditions (attributes) except BLINK; BLINK displays as BOLD. BOLD is not supported in 132-column mode, and 132-column mode is supported only by the mapped monitors.

2.12.3.2 Text Cursor Mode (DECTCEM)

Text cursor mode lets a program control whether the cursor is displayed on the video screen. Enabling text cursor mode displays the cursor and is the default. Text cursor mode is necessary when working with text because the cursor shows where the next character will be displayed.

A program places the terminal in text cursor mode by sending the 7-bit escape sequence:

ESC[?25h (transmitted as octal 033 133 077 062 065 150)

A program takes the terminal out of text cursor mode by sending the 7-bit escape sequence (note the lower-case l (?25l)):

ESC[?25l (transmitted as octal 033 133 077 062 065 154)

The cursor display can also be controlled using the SETUP CURSOR and SETUP NOCURSOR commands described in the *RT–11 Commands Manual*.

2.12.3.3 Device Attributes (DA)

A program uses the device attributes request/reply exchange to ask the terminal, "what are you?". The response sent by the terminal to the program can identify the terminal as a specific VT100 terminal (the default) or as a nonspecific member of the VT100 series of terminals. The SETUP modes VT100 and GENERIC100 (see the *RT–11 Commands Manual*) determine which of the two responses the terminal sends the program. Digital recommends that all programs recognize both the VT100 and the GENERIC100 device attributes reply.

A program can request information on two levels. The primary level DA requests basic compatibility information. The secondary level DA requests the specific version and edit level of the PI handler.

The terminal reply to primary and secondary DA requests gives this information, and also tells the program which monitor the system is running. The following is a complete DA interchange:

A program requests primary DA by sending the 7-bit escape sequence:

ESC[c (transmitted as octal 033 133 143)

- If the terminal is SETUP VT100, it responds by sending the 7-bit escape sequence:
  - When running under an unmapped monitor:

    ESC[?1;1c (transmitted as octal 033 133 077 061 073 061 143)
  - When running under a mapped monitor:

    ESC[?1;3c (transmitted as octal 033 133 077 061 073 063 143)
- If the terminal is SETUP GENERIC100 without 132-column capability (running under an unmapped monitor), it responds by sending the 7-bit escape sequence:

  ESC[?61c (transmitted as octal 033 133 077 066 061 143)
- If the terminal is SETUP GENERIC100 with 132-column capability (running under a mapped monitor), it responds by sending the 7-bit escape sequence:

  ESC[?61;1c (transmitted as octal 033 133 077 066 061 073 061 143)

A program requests the secondary DA by sending the 7-bit escape sequence:

ESC[>c (transmitted as octal 033 133 076 143)

- If the terminal is operating under an unmapped monitor, it responds by sending the 7-bit escape sequence:

  ESC[>7;VVnnc (transmitted as octal 033 133 076 067 073 V V n n 143)

  where *VV* is the version number, and *nn* is the edit level of the PI handler.
- If the terminal is operating under a mapped monitor, it responds by sending the 7-bit escape sequence:

  ESC[>8;Vnnc (transmitted as octal 033 133 076 070 073 V V n n 143)

  where *VV* is the version number, and *nn* is the edit level of the PI handler.

2.13 UB (UNIBUS Mapping Register (UMR) System Support Handler)

This section describes the UB handler that provides support for the UNIBUS mapping registers on UNIBUS processors. The UB handler provides DMA (direct memory access) support for 22-bit memory addressing during I/O operations.

UB is called a *system support handler* because RT–11 requires UB to provide certain necessary connections with the computer hardware. At bootstrap time, the monitor loads UB before binding with the system device handler file on the system volume. Therefore, UB cannot be installed with the INSTALL command. Instead, UB is automatically installed and loaded in memory on UNIBUS processors with the following configuration:

- The processor is running a mapped monitor.
- The processor contains more than 256K-bytes of memory.
- The processor contains UNIBUS Mapping Registers at addresses 170200 through 170400 to support 40_8 2-word UMRs.
- At least one device handler on the system uses DMA in performing I/O operations. All distributed RT–11 handlers that can perform DMA are so marked.

  Section 2.13.3 describes how to provide UMR support in a user-written DMA handler.
- All installed user-written (not distributed) device handlers are compatible with RT–11 support for UB. All installed device handlers must be marked as compatible with UB, whether or not they perform DMA operations.

  Section 2.13.2 describes how to make a non-DMA user-written device handler compatible with RT–11 UB support.

UNIBUS Mapping Registers function in a manner that is similar to the Memory Management Unit (MMU) registers that provide 22-bit address translation for the CPU. The UMRs provide address translation (mapping) from the 18-bit UNIBUS to the 22-bit memory bus.

2.13.1 UMR Support with Distributed Handlers

On supported UNIBUS system configurations, UB is automatically installed and loaded when the processor is booted. At that point, DMA I/O operations are handled transparently by the processor UMR hardware and the RT–11 operating system. Programs that use distributed RT–11 device handlers require no modification to support DMA access to a peripheral device.

The aspects of UMR support that apply to distributed handlers are:

- Permanent UMR allocation.

  Because of internal buffers, some RT–11 device handlers, such as DL, DM, DU, NU, and the various magtape handlers, require a preallocation of one or more permanent UMRs. RT–11 preallocates those permanent UMRs when the device handlers are installed at system boot. RT–11 reserves those permanent UMRs for those handlers when they are loaded. See Table 2–27. You can regain any preallocated permanent UMRs for handlers that install but you are not using, by renaming the device handler. Such a renamed handler does not install at the next system boot.

Contiguous permanent UMRs are allocated from the list of reserved permanent UMRs when handlers are loaded and returned to reserved status when handlers are unloaded. After numerous load/unload operations, the list of reserved permanent UMRs can become fragmented. A symptom of this condition is the inability to load a device handler that requires multiple permanent UMRs even when sufficient reserved permanent UMRs exist. Two courses of action are available if that condition occurs. You can reboot your system, or you can issue the SHOW UMR command and unload the device handlers that are displayed as occupying slots between the available reserved permanent UMRs. The system device handler resides at the top of the list. You should consolidate the list from the base upward.

- Temporary UMR allocation.

  Many distributed device handlers require one or more UMRs on a temporary basis to process I/O requests. RT–11 allocates temporary UMRs as the need occurs. Each processor contains 31_{10} accessible UMRs, and the allocation of UMRs can be displayed by the command SHOW UMR.

- Serialization of I/O request satisfaction.

  When UB is loaded in memory, RT–11 no longer always satisfies I/O requests in serial order.

  Of the distributed RT–11 device handlers, only DU and the magtape handlers (MM, MS, MT, and MU) require that I/O requests are satisfied in serial order. The guarantee of I/O request serialization is internal to those handlers and requires no user intervention.

  However, RT–11 does not guarantee that I/O requests for other device handlers are satisfied in serial order. Rather, I/O requests are satisfied in the quickest manner possible, which might or might not be serial. For example, an I/O request that requires four UMRs might be queued for a time waiting for UMR allocation, while a subsequent I/O request requiring fewer UMRs is satisfied. However, if required, you can force serialized I/O request processing, using the SET UB SERIAL=n command, described in the *RT–11 Commands Manual*.

You can control other aspects of UMR support by specifying conditions for the SET UB command. Other than those conditions, UMR support is totally transparent when using the distributed RT–11 device handlers.

Table 2–27: Distributed Handler Support for UMRs

Device Handler	DMA=	PERMUMR=
DL	YES	1
DM	YES	1
DU	YES	2
DW	NO	
DY	YES	1
MM	YES	If support for FSM included, requires 1 if no support for FSM, requires 0
MS	YES	If support for FSM included, requires 1 if no support for FSM, still requires 1
MT	YES	If support for FSM included, requires 1 if no support for FSM, requires 0
MU	YES	If support for FSM included, requires 3 if no support for FSM, requires 2
NU	YES	3
RK	YES	0
VM	NO	

2.13.2 Including Required UB Support in User-Written Non-DMA Handlers

All installed device handlers, including those that perform no DMA operations, must be modified for compatibility with UB. Otherwise, the RT–11 monitor bootstrap does not load UB and the system then operates with only the low 256K words of memory accessible to DMA operations.

You must explicitly specify whether each user-written device handler supports DMA, using the .DRDEF macro's *DMA=str* parameter. If a device handler does not perform DMA operations and, therefore, does not require UMR allocation, specify *DMA=NO*.

2.13.3 Including UMR Support in User-Written DMA Handlers

UMR support is appropriate for a device handler that performs I/O operations and is capable of DMA. Including UMR support in such a device handler lets the handler access computer memory beyond the 18-bit 256K-byte boundary during I/O operations.

The following paragraphs describe elements of the new UMR support that must be considered before you include UMR support in a device handler. Each element is either described when listed or you are pointed to the appropriate section of this manual where you will find the element description.

Including UMR support in any device handler requires that you understand the following items:

- The handler should not perform DMA operations from within its own install code. If a handler must be written to perform DMA from within its install code, you must turn off UB (SET UB NOINSTAL), reboot the system, and then install the handler.
- The handler must use the .DRDEF macro and include one or more of the parameters, *DMA=str, PERMUMR=n*, and *SERIAL=str*, as described in the *RT–11 System Macro Library Manual*.
- If the handler uses the .QELDF macro to define queue elements, you should read about the offset, Q.MEM, as described in *RT–11 System Macro Library Manual*.
- RMON automatically allocates temporary UMRs for all .READx and .WRITx requests to handlers that are marked as DMA=YES. RMON also automatically releases all such temporary UMRs. Both operations are completely transparent to the handler.
- If the handler previously used queue element offsets Q.PAR and Q.BUFF to calculate non-DMA I/O virtual addresses, it must now use the new offset Q.MEM in conjunction with Q.BUFF. Q.MEM is described in Section 1.2.1.1.2. The handler now uses Q.PAR to calculate only DMA I/O virtual addresses.
- If the handler previously used extended memory subroutines $GETBYT, $PUTBYT, $PUTWRD, or $MPPHY, read the paragraphs *Changes to extended memory subroutines for UMR support*, in *RT–11 System Release Notes*.
- You should examine the new RMON fixed offsets, $QHOOK, $H2UB, and the bits defined for UB in $CNFG3. They are described in *RT–11 System Internals Manual*.
- You should decide if I/O requests for the handler or the job must be satisfied in serial order. Once UB is loaded in memory, I/O requests are not guaranteed to be satisfied in serial order by default.

  If the handler requires serialized I/O request satisfaction, you must specify the .DRBEG macro *SERIAL=YES* parameter argument when you build the handler. See the .DRDEF macro information in the *RT–11 System Macro Library Manual*.

  If the job requires serialized I/O request satisfaction, see the SET UB SERIAL=n command described in the *RT–11 Commands Manual*.
- The device handler must use permanent or temporary UMRs for each special function that performs a DMA I/O operation.

  The handler uses permanent UMRs for processing special functions that result in a DMA I/O operation to the handler internal buffer.

  If the handler contains internal buffers that store command packets and responses, the handler has to use the ALLUMR routine to explicitly obtain at least one permanent UMR. The handler must explicitly release all permanent UMRs when it unloads, using the RLSUMR routine. Obtaining and releasing permanent UMRs is described in Sections 2.13.3.3 and 2.13.3.4.

The handler allocates at least one temporary UMR for each special function that performs a DMA I/O operation to the user buffer. The temporary UMRs are allocated either implicitly or explicitly.

Special functions (.SPFUNs) used by the handler are categorized as standard or nonstandard. A standard special function uses the .SPFUN *buf* parameter as the read/write buffer address and the *wcnt* parameter as the operation word count. Temporary UMRs for standard special functions are allocated implicitly. Defining standard special functions is described in Section 2.13.3.1.

A nonstandard special function does not use *buf* as the read/write buffer address or *wcnt* as the operation word count. The handler must explicitly obtain temporary UMRs for nonstandard special functions, requiring additional processing by UB. Processing nonstandard special functions is described in Section 2.13.3.3.

2.13.3.1 Defining Special Functions for Implicit UMR Allocation

The device handler should implicitly allocate UMRs for special functions that do the following:

- Perform DMA operations.
- Use the *buf* and *wcnt* paramaters in the documented manner; are standard special functions.

The handler supports implicit UMR allocation for standard special functions by using the .DRBEG *SPFUN=spsym* parameter and a list of those functions. The *spsym* argument is the label of the list of those functions. The list is structured in the same manner as that used for the .DRSPF *extension table* method. However, unlike the .DRSPF macro, no pointer to the list resides in block 0 of the handler and the concept of special function *type* has no meaning and is not included.

The list of standard special functions must continuously reside in the low-memory portion of the handler whenever the handler is loaded. For all special functions in the list, RMON performs the UMR allocation and the address translation.

Defining special functions for implicit UMR allocation is illustrated in the example program in this section.

2.13.3.2 Explicitly Allocating Permanent UMRs (ALLUMR)

If the device handler contains internal buffers that store command packets and responses, you must allocate at least one permanent UMR to the device handler.

RT–11 allows up to 22_{10} UMRs to be permanently allocated to handlers and one UMR is permanently allocated to the I/O page. When the system is booted, RT–11 allocates the one UMR to the system's I/O page and then reserves permanent UMRs for requesting device handlers as each handler is installed. Therefore, unless the 23_{10} limit is reached, RT–11 reserves sufficient permanent UMRs to support all installed device handlers that request permanent UMR allocation. However, reserved permanent UMRs are not allocated to a device handler until it is loaded. Unallocated reserved permanent UMRs are available for explicit allocation, using the ALLUMR routine. You can determine the current UMR allocation on your system by issuing the SHOW UMR command.

The ALLUMR routine, which resides in UB, is called to permanently allocate UMRs. If the handler requires UMRs for a single, contiguous chunk of memory, you need call ALLUMR only once. If the handler requires UMRs for noncontiguous chunks of memory, repeatedly call ALLUMR to allocate UMRs for each chunk.

You reference the UB entry vector through the $H2UB fixed offset (460) in RMON. The ALLUMR routine is offset 1 word ($H2UB+2) from the address pointed to by $H2UB.

Use the following procedure to allocate permanent UMRs:

1. Calculate the number of permanent UMRs you need for each contiguous chunk of memory. One permanent UMR is required for each 4096 words of contiguous internal buffer space.
2. Specify the total number of permanent UMRs the handler requires in the *PERUMR=n* parameter of the .DRDEF macro in your handler source code. The RT–11 monitor bootstrap (BSTRAP) uses that information to reserve the number of UMRs you permanently allocate to the handler.
3. Before calling ALLUMR to allocate permanent UMRs for an internal buffer space, set up the following registers:

Register	Contents
R0	Number of permanent UMRs to be allocated for this contiguous chunk of internal buffer space. If you request more than one permanent UMR, the address of the first is defined by R1 and R2, and each subsequent UMR is offset by a value of 20000_8.
R1	Bits 0–15 of the 22-bit physical memory base address (word aligned) of the internal buffer.
R2	Bits 16–21 of the 22-bit physical memory base address of the internal buffer.
R4	The address of a 1-word location in low memory that contains two RAD50 identifying characters. The SHOW UMR command displays these characters to identify this permanent UMR allocation. (In distributed handlers, is the device handler name.) The monitor must have continuous access to the specified memory location. If ALLUMR is called more than once for this handler, R4 in subsequent calls must contain a different address in low memory for each call. The 1-word location contents can be, but do not need to be, the same two RAD50 characters.

The contents of R3 and R5 are not defined or preserved across the call.

4. Within the device handler FETCH/LOAD code, call the ALLUMR routine. On return from ALLUMR:

   If the carry bit is clear:

   - R1 contains bits 0–15 of the 18-bit UNIBUS virtual address of the internal buffer.
   - R2 contains bits 16 and 17 of the 18-bit UNIBUS virtual address of the internal buffer.
   - The handler uses the address returned by ALLUMR (or some offset from that address) to program the device for DMA I/O to/from the handler internal buffer.

   If the carry bit is set, insufficient UMRs are available for allocation and the handler must fail its load code.

Once you have successfully called and returned from ALLUMR, your handler code should confirm that the FETCH/LOAD succeeded. If the fetch/load operation fails after successfully returning from ALLUMR, you must call RLSUMR to free the allocated UMRs.

2.13.3.3 Explicitly Obtaining Temporary UMRs (GETUMR)

Device handlers that support nonstandard .SPFUN I/O DMA operations to or from a user buffer must call GETUMR to explicitly obtain temporary UMRs to service those requests. The temporary UMRs are automatically released after the request is serviced. The handler uses the GETUMR routine, described in this section, to obtain the UMRs. Be sure to call GETUMR before removing the queue element from the handler's current queue element (xxCQE) list.

The handler supports explicit UMR allocation for nonstandard special functions by using the .DRBEG *NSPFUN=nspsym* parameter and a list of those functions. The *nspsym* argument is a unique symbol name that is the same as the label at the list of those functions. The list is structured in the same manner as that used for the .DRSPF *extension table* method. However, unlike the .DRSPF macro, no pointer to the list resides in block 0 of the handler and the concept of special function *type* has no meaning and is not included.

The list of nonstandard special functions must continuously reside in the low-memory portion of the handler whenever the handler is loaded. Also, the handler must call GETUMR (with a word count of zero) even when a listed nonstandard special function performs no I/O and no UMRs are needed.

Defining special functions for explicit UMR allocation is illustrated in the example program in this section.

The handler calls the GETUMR routine, which resides in UB, to obtain temporary UMRs. You reference the UB entry vector through the $H2UB fixed offset (460) in RMON. The GETUMR routine is located at the address pointed to by $H2UB (offset 0).

Use the following procedure to explicitly obtain temporary UMRs:

1. Before calling GETUMR, set up the following registers:

Register	Contents	
R0	Number of words to be transferred; the word count. If no DMA I/O is to be performed by this request, R0=0.	
R1	Contents determined by R3:	
	R3 = 0	R1 contains the Q.PAR value that is calculated by the handler. RMON cannot calculate the Q.PAR value because the special function's *buf* parameter contains a nonstandard argument.
	R3 = 1	R1 contains bits 0-15 of the 22-bit physical memory base address (word aligned).
R2	Contents determined by R3:	
	R3 = 0	R2 is unused.
	R3 = 1	R2 contains bits 16-21 of the 22-bit physical memory base address.
R3	Contents indicate the type of address being specified:	
	R3 = 0	Address is PAR value, specified in R1. R2 is not used.
	R3 = 1	Address is 22-bit physical address, specified in R1 and R2.
R4	Queue element offset Q.BLKN.	

   The contents of all unused registers are not defined or preserved across the call.

2. Within the device handler code that processes nonstandard special functions, call the GETUMR routine. On return from GETUMR:

   - If the carry bit is clear, the contents on return for R1 and R2 are defined by the contents of R3 when GETUMR was called. If GETUMR is called with R3 = 0, on return, R1 contains the new Q.PAR equivalent value and R2 is not defined. If GETUMR is called with R3 = 1, on return, R1 contains bits 0–15 and R2 contains bits 16 and 17 of the 18-bit UNIBUS virtual address.
   - If the carry bit is set, UB is unable to immediately allocate the requested UMRs for the queue element and the handler should simply return to the monitor.

2.13.3.4 Explicitly Releasing Permanent UMRs (RLSUMR)

All permanent UMRs that are allocated by a handler must be explicitly released by the handler when the handler is unloaded. A corresponding RLSUMR routine must be called for each ALLUMR routine that was called.

The RLSUMR routine, which resides in UB, releases permanent UMRs. You reference the UB entry vector through the $H2UB fixed offset (460) in RMON. The RLSUMR routine is offset 2 words ($H2UB+4) from the address pointed to by $H2UB.

Use the following procedure to explicitly release permanent UMRs:

1. Before calling RLSUMR, set up the following register:

Register	Contents
R1	The address of the 2-character RAD50 device handler name specified in R4 of the corresponding ALLUMR routine. (The contents of RLSUMR R1 match the contents of corresponding ALLUMR R4.)

   The contents of R0 and R2–R5 are not defined or preserved across the call.

2. Within the device handler RELEASE/UNLOAD code, call the RLSUMR routine.

   On return from RLSUMR, all UMRs that were permanently allocated to the handler by the corresponding ALLUMR routine are released.

2.13.4 Example (Skeletal) Handler

The following example skeletal handler illustrates the macros and routines required to support UMRs.

```
        .SBTTL  CONDITIONAL ASSEMBLY SUMMARY
;+
;COND
;
;       MMG$T = 1               Std conditional (XM only)
;       TIM$T                   Std conditional (no code effects)
;       ERL$G                   Std conditional (no code effects)
;-

.MACRO  ...
.ENDM

.MCALL  .DRDEF  .ASSUME .ADDR .DRSPF
.LIBRARY "SRC:SYSTEM"
.MCALL  .SYCDF  .FIXDF  .HANDF  .UBVDF  .P1XDF

        .SYCDF
        .FIXDF
        .HANDF
        .UBVDF
        .P1XDF

; UB Definitions

;  XB internal DMA buffer equates

        BUFSIZ  =: 20000                ; Size of XB internal DMA buffer

        NOUMRS  =: <BUFSIZ+7777/10000>  ; Number of permanent UMRs required

; Special function definitions
; All special functions are DMA except for FN$SIZ and FN$MPM.
; FN$WRT AND FN$RED go in UBTAB. FN$REP uses a permanent UMR.
; FM$NSP is nonstandard so it goes in UBNTAB.
```

```
        FN$MPM  =: 370                  ; Illustrate use of $MPMEM (not DMA)
        FN$NSP  =: 371                  ; Nonstandard SPFUN (DMA to
                                        ;  user buffer)
        FN$SIZ  =: 373                  ; Get device size (not DMA)
        FN$REP  =: 374                  ; Force reread of replacement table
        FN$WRT  =: 376                  ; Absolute write (no bad block)
        FN$RED  =: 377                  ; Absolute read  (replacement)

        .DRSPF  <FN$MPM>                ; Illustrate use of $MPMEM
        .DRSPF  <FN$NSP>                ; Nonstandard SPFUN (DMA to
                                        ;  user buffer)
        .DRSPF  <FN$SIZ>                ; Get device size
        .DRSPF  <FN$REP>                ; Force reread of replacement table
        .DRSPF  <FN$WRT>                ; Absolute write (no bad block)
        .DRSPF  <FN$RED>                ; Absolute read  (replacement)

; DRDEF'S serial argument must be set equal to yes since XB calls
; GETUMR and depends on receiving queue elements from RMON in serial order.
; Calls to GETUMR can interfere with the serial ordering of queue elements
; unless "SERIAL = YES" is specified here.

        .DRDEF  XB,0,SPFUN$,0,0,0,DMA=YES,PERMUMR=NOUMRS,SERIAL=YES
        .DRPTR  FETCH=FETCH,LOAD=FETCH,RELEASE=RELEAS,UNLOAD=RELEAS
        .DREST  CLASS=DVC.NL

; Start of handler

        .DRBEG  XB,SPFUN=UBTAB,NSPFUN=UBNTAB
XBBASE=XBSTRT+6
        BR      BEGIN                   ; Branch around data area

; Data area

$ENTPT: .WORD   0                       ; Pointer to $ENTRY table
$PNMPT: .WORD   0                       ; Pointer to $PNAME table
H2UB:   .WORD   0                       ; Pointer to UBVECT
XBSLOT: .WORD   0                       ; XB'S offset in device tables
XBENT:  .WORD   0                       ; XB'S $ENTRY table entry pointer
XBPNA:  .WORD   0                       ; XB'S $PNAME table entry pointer

;+
;  Definition of the handler internal buffer and the words that are
;  used to program DMA devices that transfer data to and from it.
;-

XBDBUF: .WORD   BUFSIZE                 ; XB DMA buffer - it is
                                        ;  mapped by permanent UMRs
BUFADH: .WORD   0                       ; Bits 0-15 of UNIBUS virtual
                                        ;  Pointer to XBDBUF
BUFADL: .WORD   0                       ; Bits 16-18 of UNIBUS virtual
                                        ;  Pointer to XBDBUF

; Table of standard DMA SPFUNs that do DMA transfers to areas of
; memory not mapped by XB's permanent UMRS.  UB will intercept these requests
; and assign temporary UMRs to them in the same manner as for .READx and
; .WRITx requests.

UBTAB:  .DRSPF  -,<FN$WRT>              ; Absolute write, no bad block
        .DRSPF  -,<FN$RED>              ; Absolute read (replacement)
        .WORD   0                       ; Table terminator

; Table of nonstandard DMA SPFUNs that do DMA transfers to areas of
; memory not mapped by XB's permanent UMRs.  XB MUST explicitly allocate
; UMRs for the nonstandard SPFUNs listed here by calling UB's GETUMR
; routine.  If no DMA transfer will take place (because of error, for
; example) XB should call GETUMR with a word count of 0.  IF XB processes
; a nonstandard DMA SPFUN listed in UBNTAB without calling GETUMR,
; the job's I/O stream will hang.

UBNTAB: .DRSPF  -,<FN$NSP>              ; DMA to user buffer
        .WORD   0                       ; Table terminator
```

```
BEGIN:  MOV     XBCQE,R4                ; Point to current queue element
        MOVB    Q$FUNC(R4),R2           ; Get function code / unit number
        CMPB    R2,#FN$MPM              ; Dispatch to function routine
        BEQ     FNMPM
        CMPB    R2,#FN$NSP
        BEQ     FNNSP
        CMPB    R2,#FN$SIZ
        BEQ     FNSIZ
        CMPB    R2,#FN$REP
        BEQ     FNREP
        CMPB    R2,#FN$WRT
        BEQ     FNWRT
        CMPB    R2,#FN$RED
        BEQ     FNRED
        TST     R2                      ; Normal request?
        BNE     XBEXIT                  ; No, unknown SPFUN
        BR      XBRDWR                  ; Yes, process read,write

                ...

;       Routines to perform SPFUN operations
;       at entry, R4 -> queue element

FNNSP:
        MOV     #4000,R0                ; R0 = word count
        MOV     Q$PAR(R4),R1            ; Get address from QEL
        MOV     @#$SYPTR,R3             ; Get start of RMON
        MOV     $H2UB(R3),R5            ; R5 = UB entry vector
        CLR     R3                      ; Address type is PAR value
        CALL    UB.GET(R5)              ; Try to get UMRS
                                        ; (Note that at time of call, the
                                        ; Queue element must be on xxCQE)
        BCS     RETURN                  ; Unable to get UMRs-do simple RETURN

        ...                             ; Got UMRs, initiate transfer

        BR      XBEXIT                  ; DRFIN because this is an example
                                        ; Handler and there are really no
                                        ; Interrupts associated with it.
                                        ; If there were, the DRFIN would be
                                        ; Issued at interrupt time when
                                        ; The DMA transfer is finished.
                                        ; This is true for the other SPFUN
                                        ; Routines below, as well.

FNMPM:
;       This routine illustrates how to call $MPMEM.  $MPMEM is used
;       to map KT-11 virtual addresses (as described by Q.MEM and Q.BUFF
;       offsets in the queue element) to 18 or 22-bit physical addresses.
;       $MPMEM must be used for this purpose instead of $MPPHY when the
;       handler has DMA = YES.  (When DMA = NO, the handler may use
;       either $MPMEM or $MPPHY.)
;
;       At entry:       R4 -> Q.BLKN offset in queue element

        MOV     @#$SYPTR,R3             ; Get start of RMON
        MOV     P1$EXT(R3),R3           ; R3 -> $P1EXT
        MOV     R4,R5                   ; Make R5 -> 5TH word (Q.BUFF) of
        CMP     (R5)+,(R5)+             ; Queue element
        CALL    $MPMEM(R3)              ; Map KT-11 virtual to physical
        MOV     (SP)+,R2                ; R2 = low 16 bits physical address
        MOV     (SP)+,R3                ; R3 = HIGH 2 (OR 6) bits physical
                                        ;  address
        ...                             ; Fall through to DRFIN
FNSIZ:
FNREP:
FNWRT:
FNRED:
XBRDWR:

XBEXIT: .DRFIN  XB                      ; Return to monitor, done with
                                        ; queue element

RETURN: RETURN                          ; Return to monitor, not done with
                                        ; queue element

XBINT:                                  ; Dummy ISR for XB
        ...

        .DREND  XB
```

```
.SBTTL  FETCH/LOAD CODE
;+
;       FETCH
;
;       ENTRY:  R0  = Starting address of this handler service routine.
;               R1  = Address of GETVEC routine.
;               R2  = Value $SLOT*2. (length of the $PNAME table in bytes.)
;               R3  = Type of entry.
;               R4  = Address of SY read routine.
;               R5 -> $ENTRY slot for this handler.
;
;-

FETCH:  MOV     R5,R1                   ; Save PTR to XB'S $ENTRY slot
        MOV     @R1,R0                  ; Get address of XBLQE
        MOV     @#$SYPTR,R4             ; Get start of RMON
        MOV     $H2UB(R4),R3            ; R3 = UBVECT pointer
        MOV     R3,<H2UB-XBBASE>(R0)    ; H2UB = address of UBVECT
        MOV     $PNPTR(R4),R3           ; R3 = RMON offset to PNAME table
        ADD     R4,R3                   ; R3 -> PNAME table address
        MOV     R3,<$PNMPT-XBBASE>(R0)  ; $PNMPT -> PNAME table address
        ADD     R2,R3                   ; R3 -> $ENTRY table
        MOV     R3,<$ENTPT-XBBASE>(R0)  ; $ENTPT -> $ENTRY table
        MOV     R5,<XBENT-XBBASE>(R0)   ; XBENT -> XB'S $ENTRY table entry
        SUB     R2,R5                   ; R5 -> XB'S $PNAME table entry
        MOV     R5,<XBPNA-XBBASE>(R0)   ; XBPNA -> XB'S $PNAME table entry

;+
;       Allocate permanent UMRs to point into XB's internal DMA buffers,
;       XBDBUF and XBFILL, and get the UNIBUS virtual address.
;-

        MOV     #<XBDBUF-XBBASE>,R1     ; R1 = LOW 16 bits of DMABUF address
        ADD     R0,R1                   ;
        CLR     R2                      ; R2 = HIGH 6 Bits of DMABUF address
        MOV     <XBPNA-XBBASE>(R0),R4   ; R4 -> PNAME entry for XB
        MOV     <H2UB-XBBASE>(R0),R5    ; Get UB entry address
        MOV     R0,-(SP)                ; Save XB starting address
        MOV     #NOUMRS,R0              ; R0 = number of UMRS required
        CALL    UB.ALL(R5)              ; Call ALLUMR
        MOV     (SP)+,R0                ; Restore XB starting address
        BCS     30$                     ; Couldn't get UMR, fail the load
        MOV     R1,<BUFADL-XBBASE>(R0)  ; Store UNIBUS virtual address low
        MOV     R2,<BUFADH-XBBASE>(R0)  ; Store UNIBUS virtual address high
        CLC                             ; Load succeeded
30$:    RETURN

;+
;       RELEAS
;
;       Routine to unload XB
;
;       Entry:  same as for load.
;
;-

.ENABL  LSB
RELEAS::
        MOV     R5,R1                   ; R1 = $ENTRY slot for DM
        SUB     R2,R1                   ; R2 -> $PNAME SLOT for DM
        MOV     @#$SYPTR,R4             ; Get start of RMON
        MOV     $H2UB(R4),R5            ; R5 = UB entry vector
        CALL    UB.RLS(R5)              ; Release UMRs
        RETURN                          ; And exit

.END
```

2.14 VM (Virtual Memory Handler)

This section contains specific programming information for the VM device. The *Introduction to RT–11* contains complete information on using the VM device. You should read the VM chapter in the *Introduction to RT–11* first.

The VM handler installation code determines the size of memory when the handler is installed. After determining the size of memory, the handler installation code reserves all extended memory above the handler's base address. The handler does not need to perform this operation each time it is loaded, thereby speeding the handler load process.

If you do not want to use VM and do not want VM to reserve memory for its own use, you have several options. You can remove the VM handler from your system disk so that it will not be installed when you bootstrap your system. You can set the base address above the high limit of available memory, which will prevent handler installation. Or, you can put a command in your startup command file to remove the VM handler from your system after the bootstrap has installed it. Otherwise, the VM handler installation code will always reserve extended memory for its own use, thereby making it unavailable to your program.

The base address (*n*) used in the SET VM BASE=n command is the desired base address in octal, divided by 100_8. For example, the value 1600 sets the base address at the 28K-word address boundary, or 10000 sets the base address at the 128K-word address boundary; any other value between 1600 and the physical memory high limit is also acceptable. Lowering the value at which you set the VM base increases the region size. The table below gives a list of some K-word memory sizes and corresponding values for n.

K-words	N
28	1600
32	2000
64	4000
96	6000
128	10000
256	20000
512	40000
1024	100000

Figure 2–8 shows a 22-bit system with a VM base address of 10000 (128K words).

If you are using a mapped monitor and your hardware does not have 22-bit addressing, the default VM handler will not install; you will have to change the base address to a lower value before using VM with your mapped system. You can

Figure 2–8: VM Handler in a 22-Bit System

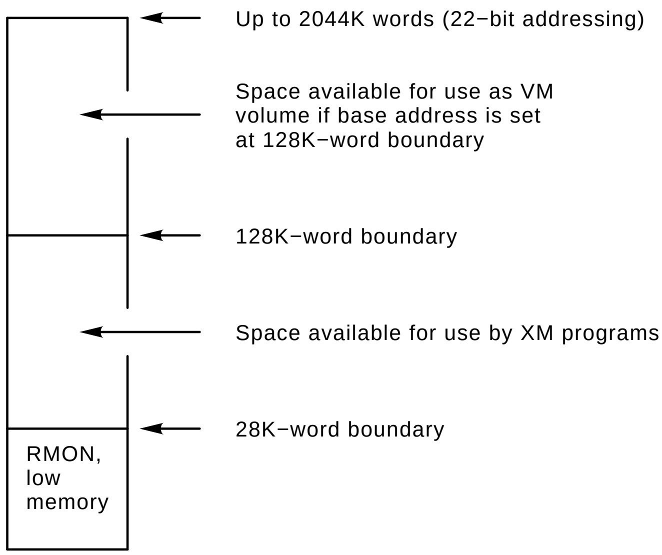
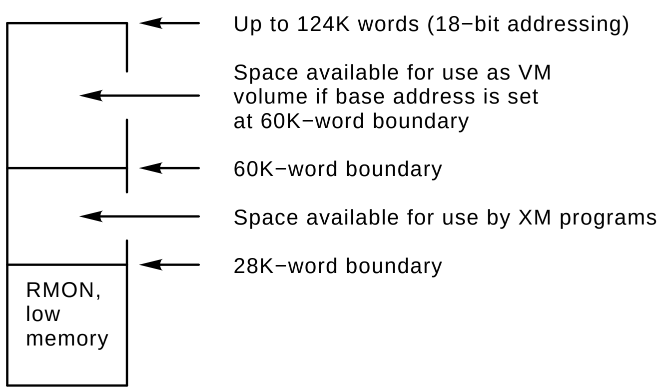


Figure 2–9: VM Handler in an 18-Bit System



still use extended memory for both an extended memory program and a VM volume, but the space available for one will be reduced by the space occupied by the other. Refer to Figure 2–9, showing an 18-bit system with the VM base address set to 3600 (60K words).

2.15 XC and XL (Communication Port (VTCOM) Handlers)

XC and XL are non-file-structured communications handlers. They support the virtual terminal communication package, VTCOM. However, their design does not preclude their use in other communication programs. The XC handler supports the CTI Bus-based computer communication port. The XL handler supports a variety of ports. See the RT–11 Software Product Description (SPD), included with your documentation set, for a list of supported ports.

XC or XL (depending on your system) is required when you use VTCOM.

XC and XL support the VTCOM utility, using .READx, .WRITx, and .SPFUN programmed requests.

2.15.1 .READx and .WRITx Support

The XC and XL handlers support the .READ, .READC, .READW, .WRITE, .WRITC, and .WRITW requests. You use the .READx and .WRITx requests with XC and XL handlers as described in the *RT–11 System Macro Library Manual*. Note, however, the following additional information:

- You should specify the value 0 in the *blk* argument for the first request to XC or XL. All subsequent calls should specify a nonzero value for the *blk* argument.
- NULL characters are ignored by XC and XL during both .READs from and .WRITEs to the handlers.
- XC and XL pass only 7-bit data. The eighth (high-order) bit is stripped from each byte.

2.15.2 Special Functions (.SPFUN) Support

In general, the XC and XL handlers support the .SPFUN request as described in the *RT–11 System Macro Library Manual*. Note, however, the following general information:

- You should specify the value 0 in the *blk* argument for the first request to XC or XL. All subsequent calls should specify a nonzero value for the *blk* argument.
- NULL characters are ignored by the XC and XL handlers; NULL characters are not stored or sent. However, SF.SRD (code 203) uses a NULL character to signal the end of available data (see SF.SRD in Table 2–28).
- XC and XL pass only 7-bit data. The eighth (high-order) bit is stripped from each byte.

The XC and XL handlers support the following special function codes. Specific information about using each special function is included in the description for that request.

Table 2–28: XC/XL Special Function Codes

<table>
<tr><th>Code</th><th>Name</th><th>Description</th></tr>
<tr><td>201</td><td>SF.CLR</td><td>Resets the internal flag, indicating a received XOFF. Then sends an XON to the host.

Example:

<code>.SPFUN #area,#chan,#SF.CLR,#buf,#wcnt,#blk[,#crtn][,BMODE=str][,CMODE=str]</code></td></tr>
<tr><td>202</td><td>SF.BRK</td><td>Sets or resets the state of the BREAK bit in the serial interface. Transition of the BREAK bit from 0 to 1 to 0 can get the attention of certain communications devices, such as terminal concentrators.

The wcnt argument is a flag that indicates whether the BREAK bit should be set or reset. Specify a value of 1 for the wcnt argument to set the BREAK bit; specify 0 to reset it. Digital recommends you use some time delay between turning the bit on and turning it off; do that by sending one or two characters.

Examples:

To turn on (set) the BREAK bit:

<code>.SPFUN #area,#chan,#SF.BRK,#buf,#1,#blk[,#crtn][,BMODE=str][,CMODE=str]</code>

To turn off (reset) the BREAK bit:

<code>.SPFUN #area,#chan,#SF.BRK,#buf,#0,#blk[,#crtn][BMODE=str][,CMODE=str]</code></td></tr>
<tr><td>203</td><td>SF.SRD</td><td>Performs a special read from the handler. The wcnt argument specifies the number of bytes to be read. The read is completed when one of the following conditions is met:

• The number of bytes specified in the wcnt argument have been transferred.

• The available characters have been transferred, when the number of available characters was less than the value specified in the wcnt argument.

• One character has been transferred, when no characters were available when the request was issued.

The byte following the last transferred character contains a NULL. You must allow for that NULL byte in your buffer.

Example:

The following example reads no more than six (but at least one) characters from XC or XL and places them in the buffer RCVBUF. RCVBUF must be at least seven bytes in length to receive the six characters and the NULL byte.

<code>.SPFUN #area,#chan,#SF.SRD,#RCVBUF,#6,#blk[,#crtn][,BMODE=str][,CMODE=str]</code></td></tr>
</table>

Table 2–28 (Cont.): XC/XL Special Function Codes

Code	Name	Description
204	SF.STS	Returns the driver status in the first word of the specified buffer. SF.STS always returns one word. The high byte of the returned word contains the driver support level. The driver support level number will be updated as support is changed in the XC and XL handlers. Programs should verify operation with an established driver support level. The current (V5.6) driver support level is 18_{10}. The low byte contains the status of two internal flags and a modem control signal. The significant bits of the low byte are: **Bit** / **Meaning** 0 Set if an XOFF has been sent to the host. 1 Set if an XOFF has been received from the host. 2 Set if the CLEAR TO SEND line is set. 3 Set if Carrier Detect is high (on); clear if Carrier Detect is low (off). 4 Set if Ring Indicator is high (on); clear if Ring Indicator is low (off). 5-7 Reserved. Example: The following example returns the driver support level in the high byte and the status of internal flags in the low byte of the 1-word buffer STATUS. `.SPFUN #area,#chan,#SF.STS,#STATUS,#1,#blk[,#crtn][,BMODE=str][,CMODE=str]`
205	SF.OFF	Sets a flag that disables interrupts when the program exits. Digital recommends you issue .SPFUN SF.OFF before your program exits. Example: `.SPFUN #area,#chan,#SF.OFF,#buf,#wcnt,#blk[,#crtn][,BMODE=str][,CMODE=str]`

Table 2–28 (Cont.): XC/XL Special Function Codes

Code	Name	Description

206	SF.DTR	Sets or resets the state of the DTR modem control signal. Setting (asserting) DTR can cause modems to answer an incoming call. Resetting (deasserting) DTR can cause modems to terminate a current call. DTR can also get the attention of certain communications devices, such as the Mini-Exchange. Specify a value of 1 for the *wcnt* argument to set the DTR control signal; specify 0 to reset the DTR control signal.
		
		Not all interfaces support the DTR control signal. On interfaces that do not support DTR, the setting or resetting of DTR has no effect.
		
		Example:
		
		The following example sets the DTR control signal:
		
		`.SPFUN #area,#chan,#SF.DTR,#buf,#1,#blk[,#crtn][,BMODE=str][,CMODE=str]`
		
		The following example resets the DTR control signal:
		
		`.SPFUN #area,#chan,#SF.DTR,#buf,#0,#blk[,#crtn][,BMODE=str][,CMODE=str]`

2.15.3 EOF (End-of-File) Detection

A CTRL/Z within data being read is treated as end-of-file (EOF) by the .READ request. At least two .READ requests are necessary to return the EOF error (carry bit set and byte 52 containing error code 0). The first .READ request transfers into your buffer all data up to (but not including) the CTRL/Z. The rest of the buffer is padded with nulls. A second .READ request is required to get the EOF error. Subsequent .READ requests can return additional characters.

Appendix A

DX, DL, and XL Device Handlers

This appendix contains annotated assembly listings of the commented DX, DL, and XL device handler source files. Besides showing good handler writing practice and demonstrating the various device handler macros, each listing illustrates certain specific device handler features:

- DX illustrates a fairly simple serial device handler.
- DL illustrates software bad block replacement.
- XL illustrates internal queuing and multiterminal handler hooks.

Each device handler was assembled with both SYSMAC.SML and SYSTEM.MLB.

Figure A–1: DX Diskette Handler

```
DX - RX01 Floppy Disk Handler  MACRO V05.05  Tuesday 26-Feb-91 14:15

Table of contents

    3-   1      CONDITIONAL ASSEMBLY SUMMARY
    4-   1      DEFINITIONS
    5-   1      INSTALLATION CHECKS
    6-   1      SET OPTIONS
    7-   1      DRIVER REQUEST ENTRY POINT
    8-   1      START TRANSFER OR RETRY
    9-   1      SILOFE - FILL OR EMPTY THE SILO
   10-   1      TABLES, FORK BLOCK, END OF DRIVER
   11-   1      BOOTSTRAP DRIVER

                000001  mmg$t=  1

                        .MCALL  .MODULE
       000000           .MODULE DX,VERSION=17,COMMENT=<RX01 Floppy Disk Handler>,AUDIT=YES

                        ;                        COPYRIGHT (c) 1989 BY
                        ;           DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
                        ;                         ALL RIGHTS RESERVED
                        ;
                        ;THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
                        ;ONLY  IN  ACCORDANCE  WITH  THE TERMS  OF  SUCH  LICENSE AND WITH THE
                        ;INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY OTHER
                        ;COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
                        ;OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS HEREBY
                        ;TRANSFERRED.
                        ;
                        ;THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT NOTICE
                        ;AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT
                        ;CORPORATION.
```

```
                ;
                ;DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF ITS
                ;SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
```

CONDITIONAL ASSEMBLY SUMMARY

```
                .SBTTL  CONDITIONAL ASSEMBLY SUMMARY
                ;+
                ;COND
                ;       DXT$O   (0)             Two controller support
                ;               0               support 1 controller
                ;               1               support 2 controllers
                ;
                ;       DX$CSR  (177170)        primary CSR
                ;       DX$CS2  (177174)        second CSR
                ;
                ;       DX$VEC  (264)           primary Vector
                ;       DX$VC2  (270)           second Vector
                ;
                ;       MMG$T                   std conditional
                ;       TIM$IT                  std conditional (no code effects)
                ;       ERL$G                   std conditional
                ;-
```

Preamble Section

```
                        .SBTTL  DEFINITIONS

                        .ENABL  LC

```

Monitor offsets and SYSCOM locations are defined with mnemonics so that references to them can be found easily:

```
                ; SOME RT-11 MACROS WE WILL USE

                        .MCALL  .DRDEF  .ASSUME .BR     .ADDR

        000342  .DSTATUS=:342                           ;EMT code for .DSTATUS
        000375  .READ   =:375                           ;EMT code for .READ
        000010          ..READ  =:010                   ; subcode for .READ
        000375  .WRITE  =:375                           ;EMT code for .WRITE
        000011          ..WRIT  =:011                   ; subcode for .WRITE

        000017  SYSCHN  =:17                            ; system channel

                ; RT-11 SYSCOM LOCATIONS

        000044          JSW     =:44                    ;JOB STATUS WORD
        000054          SYSPTR  =:54                    ;POINTER TO BASE OF RMON
        000432                  P1EXT   =: 432          ;OFFSET FROM $RMON TO EXTERNAL ROUTINE

```

If DXT$O=1, there are two controllers:

```
                ; RX01 CONTROLLER DEFAULTS

                .IIF NDF DXT$O, DXT$O=0                 ;DEFAULT TO ONLY ONE CONTROLLER

                .IIF NDF DX$CS2, DX$CS2 == 177174       ;2ND CONTROLLER CSR
                .IIF NDF DX$VC2, DX$VC2 == 270          ;2ND CONTROLLER VECTOR

```

The .DRDEF macro (with macro expansion):

```
     30 000000                  .DRDEF  DX,22,FILST$!SPFUN$!DX$COD,494.,177170,264,DMA=NO
                        .MCALL  .DRAST,.DRBEG,.DRBOT,.DREND,.DREST,.DRFIN,.DRFMS,.DRFMT
                        .MCALL  .DRINS,.DRPTR,.DRSET,.DRSPF,.DRTAB,.DRUSE,.DRVTB
                        .MCALL  .FORK,.QELDF
                        .IIF NDF RTE$M RTE$M=0
                        .IIF NE RTE$M RTE$M=1
                        .IIF NDF TIM$IT TIM$IT=0
                        .IIF NE TIM$IT TIM$IT=1
                        .IIF NDF MMG$T MMG$T=0
                000001  .IIF NE MMG$T MMG$T=1
                        .IIF NDF ERL$G ERL$G=0
                        .IIF NE ERL$G ERL$G=1
                        .IIF NE TIM$IT, .MCALL  .TIMIO,.CTIMI
        000000          .QELDF
                        .IIF NDF MMG$T,MMG$T=1
                000001  .IIF NE MMG$T,MMG$T=1
                000000  Q.LINK=:0
                000002  Q.CSW=:2.
                000004  Q.BLKN=:4.
                000006  Q.FUNC=:6.
                000007  Q.JNUM=:7.
                000007  Q.UNIT=:7.
                000010  Q.BUFF=:^o10
                000012  Q.WCNT=:^o12
                000014  Q.COMP=:^o14
                        .IRP    X,<LINK,CSW,BLKN,FUNC,JNUM,UNIT,BUFF,WCNT,COMP>
                        Q$'X=:Q.'X-^o4
                        .ENDR
DX - RX01 Floppy Disk Handler   MACRO V05.05  Tuesday 26-Feb-91 19:46  Page 4-1
DEFINITIONS

                177774  Q$LINK=:Q.LINK-^o4
                177776  Q$CSW=:Q.CSW-^o4
                000000  Q$BLKN=:Q.BLKN-^o4
                000002  Q$FUNC=:Q.FUNC-^o4
                000003  Q$JNUM=:Q.JNUM-^o4
                000003  Q$UNIT=:Q.UNIT-^o4
                000004  Q$BUFF=:Q.BUFF-^o4
                000006  Q$WCNT=:Q.WCNT-^o4
                000010  Q$COMP=:Q.COMP-^o4
                        .IF EQ MMG$T
                        Q.ELGH=:^o16
                        .IFF
                000016  Q.PAR=:^o16
                000020  Q.MEM=:^o20
                        .IRP    X,<PAR,MEM>
                        Q$'X=:Q.'X-^o4
                        .ENDR
                000012  Q$PAR=:Q.PAR-^o4
                000014  Q$MEM=:Q.MEM-^o4
                000024  Q.ELGH=:^o24
                        .ENDC
                000001  HDERR$=:1
                020000  EOF$=:^o20000
                000400  VARSZ$=:^o400
                001000  ABTIO$=:^o1000
                002000  SPFUN$=:^o2000
                004000  HNDLR$=:^o4000
                010000  SPECL$=:^o10000
                020000  WONLY$=:^o20000
                040000  RONLY$=:^o40000
                100000  FILST$=:^o100000
                000756  DXDSIZ=:494.
                000022  DX$COD=:22
                102022  DXSTS=:<22>!<FILST$!SPFUN$!DX$COD>
                        .IIF NDF DX$VEC,DX$VEC=264
                        .GLOBL  DX$VEC
```

The .DRPTR macro with no parameters:

```
     31 000200                  .DRPTR
```

The .DREST macro to define handler class and class modifier:

```
   000022                 .DREST  CLASS=DVC.DK,MOD=DVM.DX
```

The .DRSPF macro to define supported special functions:

```
   000076                 .DRSPF  <377>                  ;Read Absolute
   000032                 .DRSPF  <376>                  ;Write Absolute
   000032                 .DRSPF  <375>                  ;Write Deleted

                  ; CONTROL AND STATUS REGISTER BIT DEFINITIONS

           000001         CSGO    =:     1               ;INITIATE FUNCTION
           000020         CSUNIT  =:    20               ;UNIT BIT
           000040         CSDONE  =:    40               ;DONE BIT
           000100         CSINT   =:   100               ;INTERUPT ENABLE
           000200         CSTR    =:   200               ;TRANSFER REQUEST
           004000         CSRX02  =:  4000               ;CONTROLLER IS RX02 (ALWAYS 0)
           040000         CSINIT  =: 40000               ;RX11 INITIALIZE
           100000         CSERR   =:100000               ;ERROR

                  ; CSR FUNCTION CODES IN BITS 1-3

           000000         CSFBUF  =:0*2                  ;0 - FILL SILO (PRE-WRITE)
           000002         CSEBUF  =:1*2                  ;1 - EMPTY SILO (POST-READ)
           000004         CSWRT   =:2*2                  ;2 - WRITE SECTOR
           000006         CSRD    =:3*2                  ;3 - READ SECTOR
                                                         ;4 - UNUSED
           000012         CSRDST  =:5*2                  ;5 - READ STATUS
           000014         CSWRTD  =:6*2                  ;6 - WRITE SECTOR WITH DELETED DATA
           000016         CSMAIN  =:7*2                  ;7 - MAINTENANCE

           000002         CSREAD  =:CSEBUF&CSRD&CSRDST&CSMAIN

   000032         .ASSUME CSRD&2          NE 0    ;2 BIT MUST BE ON  IN READ
   000032         .ASSUME CSWRT&2         EQ 0    ;2 BIT MUST BE OFF IN WRITE
   000032         .ASSUME CSWRTD&2        EQ 0    ;2 BIT MUST BE OFF IN WRITE

                  ; ERROR AND STATUS REGISTER BIT DEFINITIONS

           000001         ESCRC   =:     1               ;CRC ERROR
           000002         ESPAR   =:     2               ;PARITY ERROR
           000004         ESID    =:     4               ;INITIALIZE DONE
           000100         ESDD    =:   100               ;DELETED DATA MARK
           000200         ESDRY   ==   200               ;DRIVE READY

                  ; ERROR LOG VALUES

           000003         DXNREG  =:3                    ;# OF REGISTERS TO READ FOR ERROR LOG.
           000010         RETRY   =:8.                   ;RETRY COUNT

           100000         SPFUNC  =:100000               ;SPECIAL FUNCTIONS FLAG
                                                         ; (IN COMMAND WORD)

                  ; GENERAL COMMENTS:
                  ;
                  ;  THIS HANDLER SERVES AS THE STANDARD RT-11 RX01 DEVICE HANDLER AS
                  ;  BOTH THE SYSTEM DEVICE HANDLER AND NON-SYSTEM HANDLER.  IT ALSO PRO-
                  ;  VIDES THREE SPECIAL FUNCTION CAPABILITIES TO SUPPORT PHYSICAL I/O
                  ;  ON THE FLOPPY AS A FOREIGN VOLUME.  THE SPECIAL FUNCTIONS ARE:
                  ;       CODE    ACTION
                  ;       377     ABSOLUTE SECTOR READ. WCNT=TRACK, BLK=SECTOR, BUFFER=65
                  ;               WORD BUFFER OF WHICH WORD 1 IS DELETED DATA FLAG.
                  ;       376     ABSOLUTE SECTOR WRITE. ARGUMENTS SAME AS READ.
                  ;       375     ABSOLUTE SECTOR WRITE WITH DELETED DATA. 1ST WORD
                  ;               OF 65 WORD BUFFER ALWAYS SET TO 0.
                  ;
                  ;  IN STANDARD RT-11 MODE A 2:1 INTERLEAVE IS USED ON A SINGLE TRACK AND
                  ;  A 6 SECTOR SKEW IS USED ACROSS TRACKS.  TRACK 0 IS LEFT ALONE FOR
                  ;  PROPOSED ANSI COMPATABILITY.
```

Installation checks:

```
                        .SBTTL  INSTALLATION CHECKS
                .IF EQ DXT$O
 000032                 .DRINS  DX
                .IFF
                        .DRINS  DX,<DX$CS2>
                .ENDC ;EQ DXT$O
 000200  000240         NOP                     ;SAME CHECK FOR SYSTEM AND NON-SYSTEM HANDLER
 000202  032777         BIT     #CSRX02,@INSCSR ;IS THE RX02 BIT ON?
         004000
         177766
 000210  001561         BEQ     O.GOOD          ;NOPE, IS AN RX01, INSTALL IT
 000212  000561         BR      O.BAD           ;YES, AN RX02, DON'T INSTALL IT
                ; Routine to find the entry for DX in the monitor device tables
 000214         FINDRV:
 000214                 .ADDR   #DEVNAM,R0              ;R0->DEVICE NAME
 000222                 .ADDR   #DAREA+1,-(SP)          ;(SP)->.DSTATUS INFO AREA(+physical)
 000230  104342         EMT     .DSTATUS                ;*** (.DSTAT #DAREA+1,#DEVNAM) ***
 000232  103551         BCS     O.BAD                   ;IN CASE IT'S NOT KNOWN
 000234  016701         MOV     DAREA+4,R1              ;RETURN THE ENTRY POINT
         000010
 000240  001145         BNE     O.GOOD
 000242  000545         BR      O.BAD                   ;UNLESS HANDLER'S NOT LOADED
 000244         DAREA:  .BLKW   4                       ;.DSTAT INFORMATION BLOCK
 000254  016300 DEVNAM: .RAD50  /DX /                   ;DEVICE NAME
                ; The emt area for reads/writes of the handler is placed here
                ; to leave room for code for the set options
 000256     017 BAREA:  .BYTE   SYSCHN,..READ           ;CHANNEL 17, READ
 000257     010
 000260                 .BLKW                           ;BLOCK NUMBER
 000262                 .BLKW                           ;BUFFER
 000264  000400         .WORD   256.                    ;WORD COUNT
 000266  000000         .WORD   0                       ;COMPLETION (WAIT)
                ; NOW ALTER THE CODE WHICH WILL BE WRITTEN BACK TO DISK
 000270         X.WP:
 000270                 .ADDR   #DXWPRO,R0              ;R0-> THE WRITE PROTECT TABLE
 000276  060300         ADD     R3,R0                   ; POINT TO ENTRY
 000300  112710         MOVB    (PC)+,(R0)              ; AND SET IT THE WAY THE USER WANTS IT
 000302         O.WPF:  .BLKW   1
                ; NOW TO ALTER THE IN-CORE COPY OF THE PROTECTION TABLE
 000304  004767         CALL    FINDRV                  ;IS THE HANDLER LOADED?
         177704
 000310  103521         BCS     O.GOOD                  ;NOPE...
 000312  023701         CMP     @#SYSPTR,R1             ;is this the system handler?
         000054
 000316  101003         BHI     10$                     ; no, then leave 1-shot as is
 000320  012761         MOV     #100000,DXW1-DXLQE(R1)  ; yes, set it
         100000
         000076
 000326         10$:
 000326  060301         ADD     R3,R1                   ;ADD IN UNIT OFFSET
 000330  116761         MOVB    O.WPF,DXWPRO-DXLQE(R1)  ;SET THE WRITE-PROTECT STATUS
         177746
         000010
 000336  000506         BR      O.GOOD
                        .IIF GT,<.-376> .ERROR  ;INSTALLATION CODE IS TOO LARGE;
```

The DX handler supports several SET options. Immediately following the installation code, the .DRSET macro is used to define the parameter table for each SET option:

```
                        .SBTTL  SET OPTIONS

                ; The write-protect/enable SET option makes use of the new
                ; calling convention, i.e. the unit number (DXn, n=0 if a space)
                ; passed in R1.

  000340                .DRSET  CSR,    160000, O.CSR,  OCT
  000412                .DRSET  VECTOR, 500,    O.VEC,  OCT

                        .IF NE DXT$O
                        .DRSET  CSR2,   160000, O.CSR2, OCT
                        .DRSET  VEC2,   500,    O.VEC2, OCT
                        .ENDC;NE DXT$O

  000422                .DRSET  RETRY,  127.,   O.RTRY, NUM

                        .IF NE ERL$G
                        .DRSET  SUCCES, -1,     O.SUCC, NO
                        .ENDC ;NE ERL$G

  000432                .DRSET  WRITE,  1,       O.WP,  NO

          002256        BTCSR = <DXEND-DXSTRT>+<BOTCSR-DXBOOT>+1000

```

The code to process each SET options follows the .DRSET macro calls. Normally, SET options change only the disk-resident copy of a handler, not the memory-resident copy. The DX handler SET options include special code to modify both the memory-resident and the disk-resident copy of the handler.

```
  000442  020003  O.CSR:  CMP     R0,R3                   ;IS CSR IN RANGE? (>160000)
  000444  103444          BLO     O.BAD                   ;NOPE...
  000446  010067          MOV     R0,INSCSR               ;YES, INSTALLATION CODE NEEDS IT
          177524
  000452  010067          MOV     R0,DISCSR               ;FILL IN DISPLAY CSR
          177516

                ; When the csr for units 0 and 1 is changed, the bootstrap must
                ; be altered such that it will use the correct controller.

                                                          ;R1->READ/WRITE EMT AREA
  000456                  .ADDR   #BAREA+4,R1             ; (BUFFER ADDRESS WORD)
                                                          ;BUILD ADDRESS OF BUFFER
  000464                  .ADDR   #1000,R2                ; (WHICH WILL OVERWRITE CORE
                                                          ;  COPY OF BLOCK 1)
  000472  010211          MOV     R2,(R1)                 ;SET THE BUFFER ADDRESS
  000474  012741          MOV     #BTCSR/1000,-(R1)       ;SET TO BLOCK NUMBER TO READ/WRITE
          000002
                                                          ; (BOOT BLOCK THAT NEEDS MODIFICATION)
  000500  005741          TST     -(R1)                   ;R1->EMT AREA
  000502  010003          MOV     R0,R3                   ;SAVE CSR ELSEWHERE, EMT NEEDS R0
  000504  010100          MOV     R1,R0                   ;R0->EMT AREA FOR READ
  000506  104375          EMT     .READ                   ; *** (.READW) ***
  000510  103422          BCS     O.BAD
  000512  010362          MOV     R3,<BTCSR&777>(R2)      ;SET THE NEW CSR
          000256
  000516  010100          MOV     R1,R0                   ;R0->EMT AREA FOR WRITE
  000520                                                  .ASSUME ..READ+1 EQ ..WRIT
  000520  105260          INCB    1(R0)                   ;BUMP FROM ’READ’ TO ’WRITE’
          000001
  000524  104375          EMT     .WRITE                  ; *** (.WRITW) ***
  000526  103415          BCS     O.SYWL                  ; SY: write-locked
  000530  010100          MOV     R1,R0                   ;R0->EMT AREA
  000532                                                  .ASSUME ..WRIT-1 EQ ..READ
  000532  105360          DECB    1(R0)                   ;CHANGE FROM ’WRITE’ TO ’READ’
          000001
  000536  012760          MOV     #1,2(R0)                ; OF BLOCK 1 OF HANDLER
          000001
          000002
  000544  104375          EMT     .READ                   ; *** (.READW) ***
  000546  103403          BCS     O.BAD

                          .IF EQ DXT$O
  000550  010367          MOV     R3,RXCSA
          000504’
```

```
                         .IFF
                         MOV     R3,DXCSR
                         .ENDC ;EQ DXT$O

000554  005727  O.GOOD:  TST     (PC)+                  ;GOOD RETURN (CARRY CLEAR)
000556  000261  O.BAD:   SEC                            ;ERROR RETURN (CARRY SET)
000560  000207           RETURN

000562          O.SYWL:
000562  011600           MOV     @SP,R0                 ; copy return address
000564  005200           INC     R0                     ; point to opcode at return
000566  122720           CMPB    #BR/400,(R0)+          ; is it a BR xxx?
        000001
000572  001371           BNE     O.BAD                  ; NO, old style SET
000574  010016           MOV     R0,@SP                 ; use alternate return (RET+2)
000576  000767           BR      O.BAD                  ; with carry set

000600  020003  O.VEC:   CMP     R0,R3                  ;VECTOR IN RANGE?
000602  103365           BHIS    O.BAD                  ;NOPE...
000604  032700           BIT     #3,R0                  ;YES, BUT ON A VECTOR BOUNDRY?
        000003
000610  001362           BNE     O.BAD                  ;NOPE...

                         .IF EQ DXT$O
000612  010067           MOV     R0,DXSTRT              ;YES, SET IT IN ENTRY AREA
        000000'
                         .IFF
                         MOV     R0,DX$VTB              ;PLACE IT IN MULTI-VECTOR TABLE
                         .ENDC ;NE DXT$O

000616  000756           BR      O.GOOD

                         .IF NE DXT$O
                O.CSR2:  CMP     R0,R3                  ;CSR IN RANGE?
                         BLO     O.BAD                  ;NOPE...
                         MOV     R0,DXCSR2              ;YES, PLACE IT IN CODE
                         MOV     R0,DISCS2              ;SET DISPLAY CSR
                         BR      O.GOOD

                O.VEC2:  CMP     R0,R3                  ;VECTOR IN RANGE?
                         BHIS    O.BAD                  ;NOPE...
                         BIT     #3,R0                  ;YES, BUT IS IT ON A VECTOR BOUNDARY?
                         BNE     O.BAD                  ;NOPE...
                         MOV     R0,DX$VTB+6            ;YES, PLACE IN MULTI-VECTOR TABLE
                         BR      O.GOOD
                         .ENDC ;NE DXT$O

000620  020003  O.RTRY:  CMP     R0,R3                  ;ASKING FOR TOO MANY?
000622  101355           BHI     O.BAD                  ;YES, USER IS BEING UNREASONABLE
000624  010067           MOV     R0,DRETRY              ;NOPE, SO TELL THE HANDLER
        000034'
000630  001351           BNE     O.GOOD                 ;OKAY IF NON-ZERO
000632  000751           BR      O.BAD                  ;CAN'T ASK FOR NO RETRIES

                         .IF NE ERL$G
                O.SUCC:  MOV     #0,R3                  ;'SUCCESS' ENTRY POINT
                                                        ; (MUST BE TWO WORDS)
                N.SUCC:  MOV     R3,SCSFLG              ;'NOSUCCES' ENTRY POINT
                                                        .ASSUME O.SUCC+4 EQ N.SUCC
                         BR      O.GOOD
                         .ENDC ;NE ERL$G

000634  000240  O.WP:    NOP                            ;'WRITE' ENTRY POINT
000636  005003           CLR     R3                     ;CLEAR FLAG
000640          N.WP:                                   ;'NOWRITE' ENTRY POINT
000640                                                  .ASSUME O.WP+4 EQ N.WP
000640  010367           MOV     R3,O.WPF               ;SAVE THE USER'S SELECTION
        177436
000644  010103           MOV     R1,R3                  ; save unit number
000646  020327           CMP     R3,#DXT$O*2+1          ;IS IT A VALID UNIT
        000001
000652  101341           BHI     O.BAD                  ;NOPE...
000654  000167           JMP     X.WP                   ; go to rest of the code
        177410
```

All of the code to process SET options must fit within the first block of the handler. The following line tests to make sure that this condition is satisfied:

```
                        .IIF GT,<.-1000> .ERROR ;SET CODE IS TOO LARGE;
```

Header Section

```
                        .SBTTL  DRIVER REQUEST ENTRY POINT

                        .ENABL  LSB

```

The .DRBEG macro:

```
000660                  .DRBEG  DX
```

I/O Initiation Section

```
000014  000401          BR      DXENT           ;BRANCH AROUND PROTECTION TABLE

000016          DXWPRO:
        000001          .REPT   DXT$O+1
                        .BYTE   0,0
                        .ENDR
000020                                          .ASSUME . LE DXSTRT+1000

                        .IF NE ERL$G
                SCSFLG: .WORD   0               ; :SUCCESSFUL LOGGING FLAG (DEFAULT=YES)
                                                ; =0 - LOG SUCCESSES,
                                                ; <>0 - DON'T LOG SUCCESSES
                                                .ASSUME . LE DXSTRT+1000
                        .ENDC ;NE ERL$G

                        .IF NE  DXT$O
                        .DRVTB  DX,DX$VEC,DXINT
                        .DRVTB  ,DX$VC2,DXINT
                        .ENDC ;NE DXT$O

000020          DXENT:
                .IF NE  MMG$T
000020  013704          MOV     @#SYSPTR,R4     ; R4 -> MONITOR BASE
        000054
000024  016427          MOV     P1EXT(R4),(PC)+ ; GET ADDRESS OF EXTERNALIZATION ROUTINE
        000432
000030  000432  $P1EXT:  .WORD  P1EXT           ; POINTER TO EXTERNALIZATION ROUTINE
                .ENDC ;NE MMG$T

000032  012727          MOV     (PC)+,(PC)+     ;INITIALIZE RETRY COUNT
000034  000010  DRETRY:  .WORD  RETRY           ; :RETRY MAXIMU
000036                                          .ASSUME . LE DXSTRT+1000
000036  000000  RXTRY:   .WORD  0               ; :CURRENT RETRY COUNT
```

The following instructions assemble the controller function to start up an operation and sort out special functions.

```

000040  016703          MOV     DXCQE,R3        ;GET POINTER TO QUEUE ELEMENT
        177744
000044  012305          MOV     (R3)+,R5        ;GET BLOCK NUMBER
000046  012704          MOV     #CSRD!CSGO,R4   ;GUESS THAT CONTROLLER FUNCTION IS READ
        000007
000052                                          .ASSUME Q$BLKN+2 EQ Q$FUNC
000052  112301          MOVB    (R3)+,R1        ;PICK UP SPECIAL FUNCTION CODE (SIGN EXTENDED)
000054                                          .ASSUME Q$FUNC+1 EQ Q$UNIT
000054  112300          MOVB    (R3)+,R0        ;PICK UP THE UNIT NUMBER
000056  106200          ASRB    R0              ;SHIFT IT TO CHECK FOR ODD UNIT
000060  103002          BCC     1$              ;BRANCH IF EVEN UNIT
000062  052704          BIS     #CSUNIT,R4      ;SELECT ODD UNIT FOR TRANSFER
        000020
000066          1$:
                        .IF EQ  DXT$O   ;ONE CONTROLLER
```

```
000066  132700          BITB    #6/2,R0         ;ANY UNITS BUT 0 OR 1?
        000003
000072  001163          BNE     RXERR           ;BRANCH IF YES, ERROR
                        .IFF
                        MOV     (PC)+,-(SP)     ;ASSUME FIRST DX CONTROLLER
                DXCSR = .
                         .WORD  DX$CSR
                                                .ASSUME . LE DXSTRT+1000
                        ASRB    R0              ;SHIFT UNIT TO CHECK FOR SECOND CONTROLLER
                        BCC     2$              ;NOPE, FIRST CONTROLLER
                        MOV     (PC)+,(SP)      ;CHANGE CSR TO USE SECOND CONTROLLER
                DXCSR2 = .
                         .WORD  DX$CS2
                                                .ASSUME . LE DXSTRT+1000
                2$:     MOV     (SP)+,RXCSA
                        ASRB    R0              ;BUT WAS IT UNIT 4 TO 7?
                        BCS     RXERR           ;ERROR IF SO
                        .ENDC ;EQ DXT$O
''  000074                                        .ASSUME Q$UNIT+1 EQ Q$BUFF
000074  012300          MOV     (R3)+,R0        ;GET THE USER'S BUFFER ADDRESS
000076                                          .ASSUME Q$BUFF+2 EQ Q$WCNT
000076  012302          MOV     (R3)+,R2        ;GET WORD COUNT
000100  100017          BPL     3$              ;POSITIVE MEANS READ, SO ALL SET UP

                ; HERE TO CHECK IF UNIT IS WRITE-PROTECTED

000102  006327          ASL     (PC)+           ; CHECK WRITE ANYWAY ONE-SHOT
000104  000000  DXW1:   .WORD   .-.             ; 100000 MEANS WRITE ANYWAY
000106                                          .ASSUME . LE DXSTRT+1000
000106  103412          BCS     33$             ; SKIP TEST IF WRITE ANYWAY
000110  005046          CLR     -(SP)           ;SET TO GET UNIT
000112                                          .ASSUME Q$WCNT+2 EQ Q$COMP
000112  116316          MOVB    Q$UNIT-Q$COMP(R3),(SP) ;GET IT (PLUS OTHER CRUFT
        177773
000116  042716          BIC     #<^C3>,(SP)     ; WHICH WE DISCARD NOW)
        177774
                                                ;ADD ADDRESS OF WRITE-PROTECT TABLE
000122                  .ADDR   #DXWPRO,(SP),ADD; TO UNIT OFFSET
000130  105736          TSTB    @(SP)+          ;CHECK UNIT WRITE STATUS
000132  001143          BNE     RXERR           ;IT'S WRITE-PROTECTED, USER CAN'T DO THIS
000134                                          .ASSUME CSRD-2 EQ CSWRT
000134  124444  33$:    CMPB    -(R4),-(R4)     ;CHANGE CSRD (3*2) TO CSWRT (2*2) FOR WRITE
```

Ensure that a write equals a read code minus 2:

```
000136          .ASSUME CSWRT   EQ      CSRD-2
000136  005402          NEG     R2              ; AND MAKE WORD COUNT POSITIVE
000140  006301  3$:     ASL     R1              ;DOUBLE THE SPECIAL FUNCTION CODE
000142  060701          ADD     PC,R1           ;FORM PIC REFERENCE TO CHGTBL
```

The codes for read and write operations stay the same. If the operation is for a special function, this routine sets the sign bit of the function code word, and modifies the function:

```
000144  066104          ADD     CHGTBL-.(R1),R4 ;MODIFY THE CODE, SET SIGN BIT IF SPFUN
        000740
000150  010467          MOV     R4,RXFUN2       ;SAVE THE FUNCTION CODE AND SPFUN FLAG
        000320
000154  100435          BMI     7$              ;IF SPFUN, GO DO SPECIAL SETUP

                ; NORMAL I/O, CONVERT TO TRACK AND SECTOR NUMBER AND INTERLEAVE

```

FILLCT indicates whether a multiple of four sectors has been written. If not, the handler will later zero-fill to reach a multiple of four.

```
000156  110267          MOVB    R2,FILLCT       ;SAVE WORD COUNT IN CASE WE HAVE TO FILL
        000537
000162  105367          DECB    FILLCT          ; EXTRA SECTORS ON WRITE
        000533
000166  006302          ASL     R2              ;MAKE WORD COUNT UNSIGNED BYTE COUNT
000170  006305          ASL     R5              ;NORMAL READ/WRITE. COMPUTE REAL SECTOR NUMBER
000172  006305          ASL     R5              ; AS BLOCK*4
000174  012704          MOV     (PC)+,R4        ;LOOP COUNT FOR 8 BIT DIVISION
000176     371           .BYTE  -7,-26.         ;COUNT BECOMES 1, -26 IN HIGH BYTE FOR LATER
000177     346
000200  022705  4$:     CMP     #26.*200,R5     ;DOES 26 GO INTO DIVIDEND?
        006400
000204  101002          BHI     5$              ;BRANCH IF NOT, C CLEAR
000206  062705          ADD     #-26.*200,R5    ;SUBTRACT 26 FROM DIVIDEND, SET C
        171400
000212  006105  5$:     ROL     R5              ;SHIFT DIVIDEND AND QUOTIENT
000214  105204          INCB    R4              ;DECREMENT LOOP COUNT
000216  003770          BLE     4$              ;BRANCH UNTIL DIVIDE DONE
000220  110501          MOVB    R5,R1           ;COPY TRACK NUMBER 0:75, ZERO EXTEND
000222  060405          ADD     R4,R5           ;BUMP TRACK TO 1-76, MAKE SECTOR<0
000224  010104          MOV     R1,R4           ;COPY TRACK NUMBER
000226  006301          ASL     R1              ;MULTIPLY
000230  060401          ADD     R4,R1           ; BY
000232  006301          ASL     R1              ;  6
000234  162701  6$:     SUB     #26.,R1         ;REDUCE TRACK NUMBER * 6 MOD 26
        000032
000240  003375          BGT     6$              ; TO FIND OFFSET FOR THIS TRACK, -26:0
000242  010167          MOV     R1,TRKOFF       ;SAVE IT
        000132
000246  000412          BR      8$              ;GO SAVE PARAMETERS AND START

                ; SPECIAL FUNCTION REQUEST, SET TRACK AND SECTOR AND BYTE COUNT

```

The routine passes a 65-word buffer. The first word is 0 if there is no deleted data mark.

```
000250  000305  7$:     SWAB    R5              ;PUT PHYSICAL SECTOR IN HIGH BYTE
000252  150205          BISB    R2,R5           ; AND PHYSICAL TRACK IN LOW BYTE
000254  012702          MOV     #128.,R2        ;SET THE BYTE COUNT TO 128
        000200

                        .IF EQ  MMG$T
                        CLR     (R0)+           ;CLEAR DELETED DATA FLAG WORD, BUMP USER ADDR
                        .IFF
000260  016704          MOV     DXCQE,R4        ;POINT TO QUEUE ELEMENT AT Q.BLKN
        177524
000264  005046          CLR     -(SP)           ;STACK A ZERO AND STORE IT IN FIRST WORD OF
000266  004777          CALL    @$PTWRD         ; BUFFER. NOTE THAT Q.BUFF GETS BUMPED BY 2
        000634
000272  005720          TST     (R0)+           ;ADD 2 TO OUR COPY OF USER BUFFER ADDRESS
                        .ENDC ;EQ MMG$T

                ; MERGE HERE TO START OPERATION
```

Save the user virtual buffer address, the track, the byte count, and the PAR1 value for mapped systems:

```

000274  010027  8$:     MOV     R0,(PC)+        ;SAVE BUFFER ADDRESS
000276  000000  BUFRAD:  .WORD  0               ; : USER VIRTUAL BUFFER ADDRESS
000300  010567          MOV     R5,TRACK        ;SAVE IT FOR STARTING I/O
        000126
000304  010227          MOV     R2,(PC)+        ;  AND BYTE COUNT.
000306  000000  BYTCNT:  .WORD  0               ; : BYTE COUNT FOR TRANSFER

                        .IF NE  MMG$T
000310  005723          TST     (R3)+           ;SKIP THE COMPLETION ROUTINE ADDRESS
000312  011367          MOV     @R3,PARVAL      ;SAVE THE PAR1 VALUE FOR MAPPING USER BUFFER
        000542
                        .ENDC ;NE MMG$T

000316                  .BR     RXINIT          ;GO TO FORK LEVEL AND START IT UP

```

```
                      .DSABL  LSB
```

The calculations are done; the routine can now start an operation or a retry. Before it starts, however, it arranges transfer routines for interrupt entry. To get to the ready state, force one interrupt, then return to 1$:

```
                      .SBTTL  START TRANSFER OR RETRY
                      .ENABL  LSB
000316  012767  RXINIT: MOV     #100000,RXIRTN  ;SET RETURN AFTER INITIAL INTERRUPT
        100000
        000172
000324  016704          MOV     RXCSA,R4        ;ENSURE THAT WE POINT TO THE CSR
        000154
000330  000441          BR      RXIENB          ;GO INTERRUPT, RETURN TO 1$ LATER
000332  032700  1$:     BIT     #CSREAD,R0      ;READ OR WRITE FUNCTION?
        000002
000336  001005          BNE     3$              ;IF READ, GO FILL THE SILO FROM DISK
000340  004067  2$:     JSR     R0,SILOFE       ;WRITE, LOAD THE SILO FROM THE USER BUFFER
        000440
```

Parameters for SIOFE routine:

```
000344  000001           .WORD  CSFBUF!CSGO     ; FILL BUFFER COMMAND
000346  112215           MOVB   (R2)+,@R5       ; MOVB TO BE PLACED IN-LINE IN SILOFE
000350  010115           MOV    R1,@R5          ; ZERO-FILL INSTRUCTION FOR SHORT WRITES
```

The following routine changes a sector number to an interleaved sector number:

```
000352  116702  3$:     MOVB    SECTOR,R2       ;GET THE SECTOR NUMBER
        000055
000356  003014          BGT     5$              ;POSITIVE MEANS SPFUN, DON'T INTERLEAVE
000360  162702          SUB     #-14.,R2        ;ADD 14 TO DO INTERLEAVING
        177762
000364  003003          BGT     4$              ;IF > 0, MAP -13:-1 TO 2:26, NOTE C=0
000366  062702          ADD     #12.,R2         ; ELSE MAP -26:-14 TO 1:25
        000014
000372  000261          SEC                     ;ADD 1 WHEN DOUBLING
000374  006102  4$:     ROL     R2              ;DOUBLE AND INTERLEAVE, SECTOR 1:26
000376  062702          ADD     (PC)+,R2        ;ADD IN THE TRACK OFFSET, SECTOR -25:26
000400  000000  TRKOFF:  .WORD  0               ; : TRACK OFFSET = TRACK*6 MOD 26, RANGE -26:0
000402  003002          BGT     5$              ;NO MODULUS PROBLEMS
000404  062702          ADD     #26.,R2         ;FIX TO PUT SECTOR IN 1:26 RANGE
        000032
000410  010014  5$:     MOV     R0,@R4          ;SET THE FUNCTION IN THE FLOPPY CONTROLLER
000412  105714  6$:     TSTB    @R4             ;WAIT FOR
000414  001776          BEQ     6$              ; TRANSFER READY
000416  100161          BPL     RXRTRY          ;TRANSFER DONE WITHOUT TRANSFER READY, ERROR
000420  110215          MOVB    R2,@R5          ;SET SECTOR NUMBER
000422  105714  7$:     TSTB    @R4             ;WAIT AGAIN FOR
000424  001776          BEQ     7$              ; TRANSFER READY
000426  100155          BPL     RXRTRY          ;TRANSFER DONE WITHOUT TRANSFER READY, ERROR
000430  112715          MOVB    (PC)+,@R5       ;SET THE TRACK NUMBER
000432     000  TRACK:   .BYTE  0               ;TRACK NUMBER
000433     000  SECTOR:  .BYTE  0               ;SECTOR NUMBER, KEPT < 0 UNLESS SPFUN
```

Start the operation and return to the monitor:

```
000434  052714  RXIENB: BIS     #CSINT,@R4      ;SET IE TO CAUSE AN INTERRUPT WHEN DONE IS UP
        000100
000440  000207          RETURN                  ;RETURN, WE'LL BE BACK WITH AN INTERRUPT
000442  016704  RXERR:  MOV     DXCQE,R4        ;R4 -> CURRENT QUEUE ELEMENT
        177342
000446  052754          BIS     #HDERR$,@-(R4)  ;SET HARD ERROR IN CSW
        000001
000452  000524          BR      13$             ;EXIT ON HARD ERROR
```

Interrupt Service Section

The .DRAST macro:

```
000454                  .DRAST  DX,5,RXABRT     ;AST ENTRY POINT TABLE
```

Drop to fork level rather than device priority because the routine is lengthy and it needs all the registers.

```
000464                  .FORK   DXFBLK          ;REQUEST FORK LEVEL IMMEDIATELY
```

Load registers; if the transfer is successful, this routine dispatches to the appropriate section for this interrupt. The three possibilities are: the first interrupt occurred; a read operation completed; a write operation completed. (A seek operation is treated as a zero-length read.)

```
000472  012700          MOV     (PC)+,R0        ;GET A VERY USEFUL FLAG WORD
000474  000000  RXFUN2:  .WORD  0               ; : READ OR WRITE COMMAND ON CORRECT UNIT
000476  012703          MOV     #128.,R3        ;LOAD A HANDY CONSTANT
        000200
000502  012704          MOV     (PC)+,R4        ;GET ADDRESS OF RX CONTROLLER
000504  177170  RXCSA:   .WORD  DX$CSR          ; : ADDRESS OF CONTROLLER
000506                                          .ASSUME . LE DXSTRT+1000
000506  010405          MOV     R4,R5           ;POINT R5 TO RX DATA BUFFER
000510  005725          TST     (R5)+           ;CHECK FOR ERROR, R5 -> DX REGISTER WITH ERROR
000512  100523          BMI     RXRTRY          ;ERROR, PROCESS IT
000514  006327          ASL     (PC)+           ;NO ERROR, DISPATCH AFTER INTERRUPT
000516  000000  RXIRTN:  .WORD  0               ;OFFSET TO INTERRUPT CONTINUATION
000520  103704          BCS     1$              ;FIRST INTERRUPT, START I/O
000522  032700          BIT     #CSREAD,R0      ;READ OR WRITE?
        000002
000526  001442          BEQ     10$             ;WRITE, DON'T EMPTY SILO
000530  005700          TST     R0              ;READ, IS THIS A SPECIAL FUNCTION?
```

The silo is a 128-byte (decimal) storage area in the diskette logic.

```
000532  100033          BPL     9$              ;NO, SIMPLY EMPTY THE SILO THAT WAS JUST READ
000534  032715          BIT     #ESDD,@R5       ;IF SPFUN READ, IS DELETED DATA FLAG PRESENT?
        000100
000540  001430          BEQ     9$              ;NOPE, JUST EMPTY THE SILO
```

This routine puts a 1 in the first word of the user buffer if a deleted data mark was present on a special function read operation.

```
                        .IF EQ  MMG$T
                        MOV     BUFRAD,R2       ;GET ADDRESS OF USER BUFFER AREA
                        INC     -(R2)           ;SET FLAG WORD TO 1 TO INDICATE DELETED DATA
                        .IFF
000542  010401          MOV     R4,R1           ;SAVE R4
000544  016704          MOV     DXCQE,R4        ;POINT TO QUEUE ELEMENT
        177240
000550  012746          MOV     #1,-(SP)        ;STACK A 1 TO PUT INTO FLAG WORD
        000001
000554  162764          SUB     #2,Q$BUFF(R4)   ;MOVE BUFFER POINTER BACK TO FIRST WORD.
        000002
        000004
000562  026427          CMP     Q$BUFF(R4),#20000 ;POINTER OUT OF THIS PAR'S RANGE?
        000004
        020000
000570  103011          BHIS    85$                     ;NOPE...
000572  062764          ADD     #20000,Q$BUFF(R4) ;YES, GET IT BACK IN RANGE
        020000
        000004
000600  162764          SUB     #200,Q$PAR(R4)  ; IN THE PREVIOUS PAR
        000200
        000012
000606  162764          SUB     #200,Q$MEM(R4)  ; IN THE PREVIOUS PAR
```

```
       000200
       000014
000614 004777  85$:    CALL    @$PTWRD         ;STORE IN 1ST WORD. Q.BUFF IS AGAIN ORIGINAL+2
       000306
000620 010104          MOV     R1,R4           ;RESTORE R4.
                       .ENDC ;EQ MMG$T
000622 004067  9$:     JSR     R0,SILOFE       ;FOR READ, MOVE THE DATA FROM SILO TO BUFFER
       000156
000626 000003           .WORD  CSEBUF!CSGO     ; EMPTY BUFFER COMMAND
000630 111522           MOVB   @R5,(R2)+       ; MOVB TO BE PLACED IN LINE IN SILOFE
000632 011502           MOV    @R5,R2          ; DATA SLUFFER TO BE USED FOR SHORT READ
```

This point marks the successful completion of one sector for a read or write operation. The next routine increments the pointers for the next interleaved sector.

```
000634 105267  10$:    INCB    SECTOR          ;RETURN HERE AFTER WRITES. BUMP SECTOR NUMBER
       177573
000640 001012          BNE     11$             ;NOT OFF END OF TRACK YET
000642 062767          ADD     #-26.*400+1,TRACK ;RESET SECTOR, BUMP TO NEXT TRACK
       163001
       177562
000650 062767          ADD     #6,TRKOFF       ;BUMP TRACK OFFSET VALUE
       000006
       177522
000656 003403          BLE     11$             ;OK IF STILL IN RANGE -25:0
000660 162767          SUB     #26.,TRKOFF     ;RESET TO PROPER RANGE MOD 26
       000032
       177512
```

The following routine increments the buffer address by 128 bytes, and reduces the byte count by 128. If the operation is not complete, it transfers another sector.

```
000666         11$:
                       .IF EQ  MMG$T
                       ADD     R3,BUFRAD       ;UPDATE BUFFER ADDRESS
                       .IFF
000666 062767          ADD     #2,PARVAL       ;CHANGE MAP TO BUMP ADDRESS FOR NEXT TIME
       000002
       000164
                       .ENDC ;EQ MMG$T
000674 160367          SUB     R3,BYTCNT       ;REDUCE THE AMOUNT LEFT TO TRANSFER
       177406
000700 101214          BHI     1$              ;LOOP IF WE ARE NOT DONE
```

The transfer is done. The routine sets the byte count to 0, and goes to 12$ if this was a read or a special function operation.

```
000702 005067          CLR     BYTCNT          ;FIX BYTE COUNT SO THAT WRITES ARE ALL 0-FILLS
       177400
000706 032700          BIT     #CSREAD!SPFUNC,R0 ;READ OR SPECIAL FUNCTION OPERATION?
       100002
000712 001004          BNE     12$             ;IF SO, NO ZERO-FILLING, SO WE'RE DONE
```

The operation was a write. The routine may need to be zero-filled up to three sectors (see FILLCT above).

```
000714 062727          ADD     #040000,(PC)+   ;CHECK ORIGINAL WORD COUNT FOR # OF SECTORS
       040000
000720    000           .BYTE  0               ;   FILLER
000721    000  FILLCT:  .BYTE  0               ; : ORIGINAL WORD COUNT LOW BYTE IN HIGH BYTE
000722 103206          BCC     2$              ;YES, LOOP FOR ZERO-FILLING ON WRITE
000724         12$:                            ;AHH, A SUCCESSFUL TRANSFER IS DONE
                       .IF NE  ERL$G
```

Log a successful transfer:

```
                          TST     SCSFLG          ;LOGGING SUCCESSFUL TRANSFERS?
                          BNE     13$             ;NOPE...
                          MOV     #DX$COD*400+377,R4 ;SET UP R4 = ID/-1
                          MOV     DXCQE,R5        ; AND R5 -> CURRENT QUEUE ELEMENT
                          CALL    @$ELPTR         ;CALL ERROR LOGGER TO REPORT SUCCESS
                          .ENDC ;EQ ERL$G
000724  005077  13$:      CLR     @RXCSA          ;DISABLE FLOPPY INTERRUPTS
        177554
```

I/O Completion Section

```
000730          14$:      .DRFIN  DX              ;GO TO I/O COMPLETION
```

The abort routine:

```
                ; ABORT TRANSFER
000746  012777  RXABRT:   MOV     #CSINIT,@RXCSA  ;PERFORM AN RX11 INITIALIZE
        040000
        177530
000754  005067            CLR     DXFBLK+2        ;CLEAR FORK BLOCK TO AVOID A DISPATCH
        000130
```

Go to .DRFIN if no error:

```
000760  000763            BR      14$             ; AND FINISH UP THIS I/O
```

If error logging was built:

```
                          .DSABL  LSB
                ; TRANSFER ERROR HANDLING
000762          RXRTRY:
                          .IF NE  ERL$G
                          .ADDR   #DXRBUF,R3      ;R3 -> LOCATION TO STORE REGISTER INFO.
                          MOV     R3,R2           ;SAVE IN R2 FOR LATER
                          MOV     @R4,(R3)+       ;STORE RXCS
                          MOV     @R5,(R3)+       ;STORE STATUS RXES
                          MOV     #CSMAIN!CSGO,@R4 ;READ ERROR REGISTER (NO INTERRUPTS)
                1$:       BIT     #CSDONE,@R4     ;WAIT FOR READ COMPLETION
                          BEQ     1$
                          MOV     @R5,@R3         ;STORE IN BUFFER
                          MOV     DRETRY,R3
                          SWAB    R3
                          ADD     #DXNREG,R3      ;R3 = MAX RETRIES/# OF REGS
                          MOV     #DX$COD*400,R4  ;R4 = DEVICE ID IN HIGH BYTE
                          BISB    RXTRY,R4        ; AND CURRENT RETRY COUNT IN LOW BYTE
                          DECB    R4              ;  -1 FOR THIS ERROR
                          MOV     DXCQE,R5        ;R5 -> QUEUE ELEMENT
                          CALL    @$ELPTR         ;CALL ERROR LOGGER
                          MOV     RXCSA,R4        ;RESTORE R4 = RXCS ADDRESS
                          .ENDC ;NE ERL$G
```

See if a retry is allowed:

```
000762  005367            DEC     RXTRY           ;SHOULD WE TRY AGAIN?
        177050
000766  003002            BGT     2$              ;YES
000770  000167            JMP     RXERR           ;NOPE, REPORT AN ERROR
        177446
000774  012714  2$:       MOV     #CSINIT,@R4     ;START A RECALIBRATE
        040000
```

Retry the operation:

```
001000  000167          JMP     RXINIT          ;EXIT THROUGH START OPERATION CODE
        177312
                        .SBTTL  SILOFE - FILL OR EMPTY THE SILO
                ;+
                ; SILOFE - FILL OR EMPTY THE SILO, DUMPING OR ZERO-FILLING IF NEEDED
                ;
                ;       R3 =  128.
                ;       R4 -> FLOPPY CSR
                ;       JSR     R0,SILOFE
                ;        COMMAND: CSFBUF!CSGO FOR FILL (WRITE)
                ;                 CSEBUF!CSGO FOR EMPTY (READ)
                ;        FILL/EMPTY INSTRUCTION:  (R2 -> USER BUFFER, R5 -> RXDB)
                ;                                MOVB (R2)+,@R5 FOR FILL (WRITE)
                ;                                MOVB @R5,(R2)+ FOR EMPTY (READ)
                ;        SLUFF INSTRUCTION: (R1 = 0, R5 -> RXDB)
                ;                           CLRB @R5    FOR FILL (WRITE)
                ;                           MOVB @R5,R2 FOR EMPTY (READ)
                ;       R1 =  RANDOM
                ;       R2 =  RANDOM
                ;
                ; NOTE: 1. THIS ROUTINE ASSUMES ERROR CAN NOT COME UP DURING A FILL OR EMPTY!!
                ;       2. SEEK DOES A SILO EMPTY, A TIME WASTER
                ;-
                        .ENABL  LSB
```

The diskette deals only in units of 128 decimal bytes. If a request to read is for fewer than 128 bytes, the handler reads 128 bytes and sloughs the extra bytes. If a request to write is for fewer than 128 bytes, the handler zero-fills to reach 128 bytes.

```
001004  012014  SILOFE: MOV     (R0)+,@R4       ;INITIATE FILL OR EMPTY BUFFER COMMAND
001006  012067          MOV     (R0)+,3$        ;PUT CORRECT MOV INSTRUCTION IN FOR FILL/EMPTY
        000036
001012  012067          MOV     (R0)+,5$        ;PUT IN INSTRUCTION TO SLUFF DATA
        000052
001016  016701          MOV     BYTCNT,R1       ;GET BYTE COUNT
        177264
001022  001417          BEQ     4$              ;IF ZERO, WE ARE SEEKING OR ZERO FILLING
001024  020103          CMP     R1,R3           ;IS THE BYTE COUNT <= 128?
001026  101401          BLOS    1$              ;OK IF SO
001030  010301          MOV     R3,R1           ;DO ONLY 128 BYTES AT A TIME
001032  016702  1$:     MOV     BUFRAD,R2       ;GET USER VIRTUAL BUFFER ADDRESS IN R2
        177240
```

The following section of code can be executed in two different ways. If the handler is assembled for an unmapped monitor, the code between the symbols 2$ and PARVAL is simply executed in-line. If the handler is assembled for a mapped monitor, the JSR to PIEXT and the word PARVAL are included. In this situation, the routine P1EXT copies the code between 2$ and PARVAL to the monitor stack, uses the value passed in PARVAL to map to the user buffer, and executes the code from the monitor stack. This is done to ensure that the code is not in the PAR1 area when it is executed, since PAR1 is used to map to the user buffer.

```
                .IF NE  MMG$T
001036  004077          JSR     R0,@$P1EXT      ;Let the monitor execute the following code.
        176766
001042  000016          .WORD   PARVAL-.        ;Number of instructions in bytes plus 2.
                .ENDC ;NE MMG$T
001044  105714  2$:     TSTB    @R4             ;**EXT** TRY FOR THE TRDY
001046  100376          BPL     2$              ;**EXT** TRANSFER READY
001050  000000  3$:     HALT                    ;**EXT** INSTRUCTION TO MOV OR SLUFF DATA FROM
001052  105714          TSTB    @R4             ;**EXT** TOUCH THE CSR TO GET IT READY
001054  105301          DECB    R1              ;**EXT** CHECK FOR COUNT DONE
001056  001372          BNE     2$              ;**EXT** STILL MORE TO TRANSFER
                .IF NE  MMG$T
001060  000000  PARVAL: .WORD   0               ;using this value for the PAR 1 bias.
                .ENDC ;NE MMG$T
```

The slough routine:

```
001062  105714  4$:     TSTB    @R4             ;WAIT FOR TRANSFER READY OR TRANSFER DONE
001064  003003          BGT     6$              ;TDNE UP WITH NO TRDY, SO ALL DONE
001066  001775          BEQ     4$              ;LOOP
001070  000000  5$:     HALT                    ;TRANSFER READY, SO SLUFF DATA
001072  000773          BR      4$              ;LOOP TO SLUFF MORE
001074  000200  6$:     RTS     R0              ;RETURN
                        .DSABL  LSB

                        .SBTTL  TABLES, FORK BLOCK, END OF DRIVER

                ; CHANGES TO CSR CODE FOR SPECIAL FUNCTIONS

001076  100006          .WORD   CSWRTD-CSRD+SPFUNC      ;375: READ+GO -> WRITE DELETED+GO
001100  077776          .WORD   CSWRT-CSRD+SPFUNC       ;376: READ+GO -> WRITE+GO
001102  100000          .WORD   CSRD-CSRD+SPFUNC        ;377: READ+GO -> READ+GO
001104  000000  CHGTBL: .WORD   0                       ; READ/WRITE STAY THE SAME

001106  000000  DXFBLK: .WORD   0,0,0,0         ;DX FORK QUEUE ELEMENT
001110  000000
001112  000000
001114  000000

                        .IF NE  ERL$G
                DXRBUF: .BLKW   DXNREG          ;ERROR LOG STORAGE
                        .ENDC ;NE ERL$G
```

Bootstrap driver

```
                        .SBTTL  BOOTSTRAP DRIVER

```

The .DRBOT macro:

```
001116                  .DRBOT  DX,BOOT1,READ
```

Termination Section

The .DREND macro generated by .DRBOT (the macro expansion):

```
001116                  .DREND  DX,0,
                .IF B <>
001116          .PSECT  DXDVR
                .IFF
                .PSECT
                .ENDC
                .IIF NDF DX$END,DX$END::
                .IF EQ  .-DX$END
                .IF NE MMG$T!<0&2.>
001116  000000  $RLPTR::.WORD   0
001120  000000  $MPPTR::.WORD   0
001122  000000  $GTBYT::.WORD   0
001124  000000  $PTBYT::.WORD   0
001126  000000  $PTWRD::.WORD   0
                .ENDC
                .IF NE ERL$G!<0&1>
                $ELPTR::.WORD   0
                .ENDC
                .IF NE TIM$IT!<0&4.>
                $TIMIT::.WORD   0
                .ENDC
001130  000000  $INPTR::.WORD   0
001132  000000  $FKPTR::.WORD   0
                .IIF NDF ...V22 ...V22=0
                .IF NE  ...V22&^o40000
                DX$X64 =:.
                .REPT   16.
                        .WORD   0
                .ENDR
                .ENDC
                .GLOBL  DXSTRT
```

The following line marks the end of the loadable portion of the handler. It is used to determine the handler’s length.

```
             001134' DXEND==.
                     .IFF
                     .PSECT  DXBOOT
                     .IIF LT <DXBOOT-.+^o664>,.ERROR;?SYSMAC-E-Primary boot too large;
                     .=DXBOOT+^o664
                     BIOERR: JSR     R1,REPORT
                              .WORD  IOERR-DXBOOT
                     REPORT: MOV     #BOOTF-DXBOOT,R0
                             MOV     #30002$-DXBOOT,R2
                             CALL    @R2
                             MOV     @R1,R0
                             CALL    @R2
                             MOV     #CRLFLF-DXBOOT,R0
                             CALL    @R2
                     30001$: HALT
                             BR      30001$
                     30002$: TSTB    @#TPS
                             BPL     30002$
                             MOVB    (R0)+,@#TPB
                             BNE     30002$
                             RETURN
                     BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"
                     IOERR:  .ASCII  "I/O error"
                     CRLFLF: .ASCIZ  <CR><LF><LF>
                             .EVEN
                     .IIF NDF ...V7,...V7=-1
                     .REPT   4.
                             .WORD   ...V7
                     .ENDR
                     DXBEND::
                     .ENDC
                     .IIF NDF TPS,TPS=:^o177564
                     .IIF NDF TPB,TPB=:^o177566
             000012  LF=:^o12
             000015  CR=:^o15
             001000  B$BOOT=:^o1000
             004716  B$DEVN=:^o4716
             004722  B$DEVU=:^o4722
             004730  B$READ=:^o4730
                     .IF NDF B$DNAM
                     .IF EQ  MMG$T
                     B$DNAM=:^RDX
                     .IFF
                     B$DNAM=:^RDXX
                     .ENDC ; EQ MMG$T
                     .ENDC ; NDF B$DNAM
     000062          .ASECT
             000062  .=^o62
     000062  000000'         .WORD   DXBOOT,DXBEND-DXBOOT,READ-DXBOOT
     000064  001000
     000066  000224
     000000          .PSECT  DXBOOT
     000000  000240  DXBOOT::NOP
     000002  000413          BR      BOOT1-2.
             000100  ...V2=^o100
                     .IRP    X       <UBUS,QBUS>
                     ...V3=0
                     .IIF    IDN     <X>     <UBUS>  ...V3=1.
                     .IIF    IDN     <X>     <QBUS>  ...V3=2.
                     .IIF    IDN     <X>     <CBUS>  ...V3=4.
                     .IIF    IDN     <X>     <UMSCP> ...V3=^o10
                     .IIF    IDN     <X>     <QMSCP> ...V3=^o20
                     .IIF    IDN     <X>     <CMSCP> ...V3=^o40
                     .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
                     ...V2=...V2!...V3
                     .ENDR
             000000  ...V3=0
             000001  .IIF    IDN     <UBUS>  <UBUS>  ...V3=1.
                     .IIF    IDN     <UBUS>  <QBUS>  ...V3=2.
                     .IIF    IDN     <UBUS>  <CBUS>  ...V3=4.
                     .IIF    IDN     <UBUS>  <UMSCP> ...V3=^o10
                     .IIF    IDN     <UBUS>  <QMSCP> ...V3=^o20
                     .IIF    IDN     <UBUS>  <CMSCP> ...V3=^o40
```

```
                       .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
               000101  ...V2=...V2!...V3
               000000  ...V3=0
                       .IIF    IDN     <QBUS>  <UBUS>  ...V3=1.
               000002  .IIF    IDN     <QBUS>  <QBUS>  ...V3=2.
                       .IIF    IDN     <QBUS>  <CBUS>  ...V3=4.
                       .IIF    IDN     <QBUS>  <UMSCP> ...V3=^o10
                       .IIF    IDN     <QBUS>  <QMSCP> ...V3=^o20
                       .IIF    IDN     <QBUS>  <CMSCP> ...V3=^o40
                       .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
               000103  ...V2=...V2!...V3
               000026' .=BOOT1-6.
       000026     020          .BYTE   ^o20,...V2,^o20,^o^C<20+...V2+20>
       000027     103
       000030     020
       000031     234
                       .IF     EQ      <1-1>
       000032  000400          BR      BOOT1
                       .IFF
                       .IF     EQ      <1-2.>
                               BMI     BOOT1
                       .IFF
                               .ERROR;?SYSMAC-E-Invalid S I D E S, expecting 1/2, found - 1;
                       .ENDC
                       .ENDC

               000014'         . = DXBOOT+14
       000014  000120          .WORD   READS-DXBOOT
       000016  000340          .WORD   340
       000020  000070          .WORD   WAIT-DXBOOT
       000022  000340          .WORD   340
```

Locations 34 through 52 are reserved for Digital.

```
               000034'         . = DXBOOT+34           ;34-52 USEABLE
       000034  116067  BOOT1:  MOVB    UNITRD-DXBOOT(R0),RDCMD ;SET READ FUNCTION FOR CORRECT UNIT
               000056
               000066
       000042  011706  REETRY: MOV     @PC,SP          ;INIT SP WITH NEXT INSTRUCTION
       000044  012702          MOV     #200,R2         ;AREA TO READ IN NEXT PART OF BOOT
               000200
       000050  005000          CLR     R0              ;SET TRACK NUMBER
       000052  000446          BR      B2$             ;OUT OF ROOM HERE, GO TO CONTINUATION

               000056'         . = DXBOOT+56
       000056     007  UNITRD: .BYTE   CSGO+CSRD       ;READ FROM UNIT 0, SETS WEIRD BUT OK PS
       000057     027          .BYTE   CSGO+CSRD+CSUNIT;READ FROM UNIT 1

               000070'         . = DXBOOT+70           ;PAPER TAPE VECTORS
       000070  005714  WAIT:   TST     @R4             ;IS TR, ERR, DONE UP? INT ENB CAN'T BE
       000072  001776          BEQ     WAIT            ;LOOP TILL SOMETHING
       000074  100762          BMI     REETRY          ;START AGAIN IF ERROR
       000076  000002  RTIRET: RTI                     ;RETURN

               000120'         . = DXBOOT+120
       000120  012704  READS:  MOV     (PC)+,R4        ;R4 -> RX STATUS REGISTER
       000122  177170  BOTCSR:  .WORD  DX$CSR
       000124  010405          MOV     R4,R5           ;R5 WILL POINT TO RX DATA BUFFER
       000126  012725          MOV     (PC)+,(R5)+     ;INITIATE READ FUNCTION
       000130  000000  RDCMD:  .WORD   0               ;GETS FILLED WITH READ COMMAND
       000132  000004          IOT                     ;CALL WAIT SUBROUTINE
       000134  010315          MOV     R3,@R5          ;LOAD SECTOR NUMBER INTO RXDB
       000136  000004          IOT                     ;CALL WAIT SUBROUTINE
       000140  010015          MOV     R0,@R5          ;LOAD TRACK NUMBER INTO RXDB
       000142  000004          IOT                     ;CALL WAIT SUBROUTINE
       000144  012714          MOV     #CSGO+CSEBUF,@R4;LOAD EMPTY BUFFER FUNCTION INTO RXCS
               000003
               000220  BROFFS  =       READF-.         ;USE FOR COMPUTING BR OFFSET
       000150  000004  RDX:    IOT                     ;CALL WAIT SUBROUTINE
       000152  105714          TSTB    @R4             ;IS TRANSFER READY UP?
       000154  100350          BPL     RTIRET          ;BRANCH IF NOT, SECTOR MUST BE LOADED
       000156  111522          MOVB    @R5,(R2)+       ;MOVE DATA BYTE TO MEMORY
       000160  005301          DEC     R1              ;CHECK BYTE COUNT
       000162  003372          BGT     RDX             ;LOOP AS LONG AS WORD COUNT NOT UP
       000164  005002          CLR     R2              ;KLUDGE TO SLUFF BUFFER IF SHORT WD CNT
```

```
000166  000770          BR      RDX             ;LOOP

000170  010601  B2$:    MOV     SP,R1           ;SET TO BIG WORD COUNT
000172  005200          INC     R0              ;SET TO ABSOLUTE TRACK 1
000174  011703          MOV     @PC,R3          ;ABSOLUTE SECTOR 3 FOR NEXT PART
000176                                          .ASSUME BPT EQ 3
                .IF EQ  <<BPT>>-<<3>>
                .IFF
                .IF B <>
                .ERROR;?SYSMAC-W-"BPT EQ 3" is not true;
                .IFF
                .ERROR  ;?SYSMAC-;
                .ENDC
                .ENDC
000176  000003          BPT                     ;CALL READS SUBROUTINE
                ;SECTOR 2 OF RX BOOT
000200  122323  BOOT2:  CMPB    (R3)+,(R3)+     ;BUMP TO SECTOR 5
000202  000003          BPT                     ;CALL READS SUBROUTINE
000204  122323          CMPB    (R3)+,(R3)+     ;BUMP TO SECTOR 7
000206  000003          BPT                     ;CALL READS SUBROUTINE
000210  032767          BIT     #CSUNIT,RDCMD   ;CHECK UNIT ID
        000020
        177712
000216  001173          BNE     BOOT            ;BRANCH IF BOOTING UNIT 1, R0=1
000220  005000          CLR     R0              ;SET TO UNIT 0
000222  000571          BR      BOOT            ;NOW WE ARE READY TO DO THE REAL BOOT

000224  012737  READ:   MOV     (PC)+,@(PC)+    ;MODIFY READ ROUTINE
000226  000167           .WORD  167
000230  000150           .WORD  RDX-DXBOOT
000232  012737          MOV     (PC)+,@(PC)+
000234  000214           .WORD  READF-RDX-4
000236  000152           .WORD  RDX-DXBOOT+2
000240  012737          MOV     #READ1-DXBOOT,@#B$READ ;CALLS TO B$READ WILL GO TO READ1
        000300
        004730
000246  012737          MOV     #TRWAIT-DXBOOT,@#20 ;LETS HANDLE ERRORS DIFFERENTLY
        000416
        000020
000254  005037          CLR     @#JSW           ;CLEAR JSW SINCE THE DX BOOT IN SYSCOM AREA
        000044
000260  005767          TST     HRDBOT          ;DID WE REACH HERE VIA A HARDWARE BOOT?
        000346
000264  001405          BEQ     READ1           ;YES, DON'T SET UP UNIT NUMBER
000266  013703          MOV     @#B$DEVU,R3     ;NO, SET UP UNIT NUMBER
        004722
000272  116367          MOVB    UNITRD-DXBOOT(R3),RDCMD ;STORE UNIT NUMBER
        000056
        177630
000300  006300  READ1:  ASL     R0              ;CONVERT BLOCK TO LOGICAL SECTOR
000302  006300          ASL     R0              ;LSN=BLOCK*4
000304  006301          ASL     R1              ;MAKE WORD COUNT BYTE COUNT
000306  010046  1$:     MOV     R0,-(SP)        ;SAVE LSN FOR LATER
000310  010003          MOV     R0,R3           ;WE NEED 2 COPIES OF LSN FOR MAPPER
000312  010004          MOV     R0,R4
000314  005000          CLR     R0              ;INIT FOR TRACK QUOTIENT
000316  000402          BR      3$              ;JUMP INTO DIVIDE LOOP

000320  162703  2$:     SUB     #23.,R3         ;PERFORM MAGIC TRACK DISPLACEMENT
        000027
000324  005200  3$:     INC     R0              ;BUMP QUOTIENT, STARTS AT TRACK 1
000326  162704          SUB     #26.,R4         ;TRACK=INTEGER(LSN/26)
        000032
000332  100372          BPL     2$              ;LOOP - R4=REM(LSN/26)-26
000334  022704          CMP     #-14.,R4        ;SET C IF SECTOR MAPS TO 1-13
        177762
000340  006103          ROL     R3              ;PERFORM 2:1 INTERLEAVE
000342  162703  4$:     SUB     #26.,R3         ;ADJUST SECTOR INTO RANGE -1,-26
        000032
000346  100375          BPL     4$              ;(DIVIDE FOR REMAINDER ONLY)
000350  062703          ADD     #27.,R3         ;NOW PUT SECTOR INTO RANGE 1-26
        000033
000354  000003          BPT                     ;CALL READS SUBROUTINE
000356  012600          MOV     (SP)+,R0        ;GET THE LSN AGAIN
000360  005200          INC     R0              ;SET UP FOR NEXT LSN
000362  005701          TST     R1              ;WHATS LEFT IN THE WORD COUNT
000364  003350          BGT     1$              ;BRANCH TO TRANSFER ANOTHER SECTOR
000366  000207          RETURN
```

```
    000370  005714  READF:  TST     @R4             ;ERROR, DONE, OR TR UP?
    000372  001776          BEQ     READF           ;BR IF NOT
    000374  100533          BMI     BIOERR          ;BR IF ERROR
    000376  105714          TSTB    @R4             ;TR OR DONE?
    000400  100011          BPL     READFX          ;BR IF DONE
    000402  111522          MOVB    @R5,(R2)+       ;MOVE DATA BYTE TO MEMORY
    000404  005301          DEC     R1              ;CHECK BYTE COUNT
    000406  003370          BGT     READF           ;LOOP IF MORE
    000410  012702          MOV     #1,R2           ;SLUFF BUFFER IF SHORT WD CNT
            000001
                                                    ;DON'T DESTROY LOC 0
    000414  000765          BR      READF           ;LOOP

    000416  005714  TRWAIT: TST     @R4             ;ERROR, DONE, OR TR UP?
    000420  100521          BMI     BIOERR          ;HARD HALT ON ERROR
    000422  001775          BEQ     TRWAIT          ;BR IF NOT
    000424  000002  READFX: RTI

            000606'         . = DXBOOT+606
    000606  012706  BOOT:   MOV     #10000,SP       ;SET STACK POINTER
            010000
    000612  010046          MOV     R0,-(SP)        ;SAVE THE UNIT NUMBER
    000614  012700          MOV     #2,R0           ;READ IN SECOND PART OF BOOT
            000002
    000620  012701          MOV     #<4*400>,R1     ;EVERY BLOCK BUT THE ONE WE ARE IN
            002000
    000624  012702          MOV     #1000,R2        ;INTO LOCATION 1000
            001000
    000630  005027          CLR     (PC)+           ;CLEAR TO SHOW HARDWARE BOOT
    000632  000001  HRDBOT:  .WORD  1               ;INITIALLY SET TO 1
    000634  004767          CALL    READ            ;GO READ IT IN
            177364
    000640  012737          MOV     #READ1-DXBOOT,@#B$READ ;STORE START LOCATION FOR READ ROUTINE
            000300
            004730
    000646  012737          MOV     #B$DNAM,@#B$DEVN ;STORE RAD50 DEVICE NAME
            016330
            004716
    000654  012637          MOV     (SP)+,@#B$DEVU  ;STORE THE UNIT NUMBER
            004722
    000660  000137          JMP     @#B$BOOT        ;START SECONDARY BOOT
            001000

    000664                  .DREND  DX
                    .IF B <>
    001134          .PSECT  DXDVR
                    .IFF
                    .PSECT
                    .ENDC
                    .IIF NDF DX$END,DX$END::
                    .IF EQ  .-DX$END
                    .IF NE MMG$T!<0&2.>
                   $RLPTR::.WORD   0
                   $MPPTR::.WORD   0
                   $GTBYT::.WORD   0
                   $PTBYT::.WORD   0
                   $PTWRD::.WORD   0
                    .ENDC
                    .IF NE ERL$G!<0&1>
                   $ELPTR::.WORD   0
                    .ENDC
                    .IF NE TIM$IT!<0&4.>
                   $TIMIT::.WORD   0
                    .ENDC
                   $INPTR::.WORD   0
                   $FKPTR::.WORD   0
                    .IIF NDF ...V22 ...V22=0
                    .IF NE  ...V22&^o40000
                   DX$X64 =:.
                    .REPT   16.
                            .WORD   0
                    .ENDR
                    .ENDC
                    .GLOBL  DXSTRT
                   DXEND==.
                    .IFF
    000664          .PSECT  DXBOOT
```

```
                       .IIF LT <DXBOOT-.+^o664>,.ERROR;?SYSMAC-E-Primary boot too large;
               000664' .=DXBOOT+^o664
       000664  004167  BIOERR: JSR     R1,REPORT
               000002
       000670  000753           .WORD  IOERR-DXBOOT
       000672  012700  REPORT: MOV     #BOOTF-DXBOOT,R0
               000740
       000676  012702          MOV     #30004$-DXBOOT,R2
               000722
       000702  004712          CALL    @R2
       000704  011100          MOV     @R1,R0
       000706  004712          CALL    @R2
       000710  012700          MOV     #CRLFLF-DXBOOT,R0
               000764
       000714  004712          CALL    @R2
       000716  000000  30003$: HALT
       000720  000776          BR      30003$
       000722  105737  30004$: TSTB    @#TPS
               177564
       000726  100375          BPL     30004$
       000730  112037          MOVB    (R0)+,@#TPB
               177566
       000734  001372          BNE     30004$
       000736  000207          RETURN
       000740     015  BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"
       000741     012
       000742     077
       000743     102
       000744     117
       000745     117
       000746     124
       000747     055
       000750     125
       000751     055
       000752     000
       000753     111  IOERR:  .ASCII  "I/O error"
       000754     057
       000755     117
       000756     040
       000757     145
       000760     162
       000761     162
       000762     157
       000763     162
       000764     015  CRLFLF: .ASCIZ  <CR><LF><LF>
       000765     012
       000766     012
       000767     000
                               .EVEN
                       .IIF NDF ...V7,...V7=-1
               000004  .REPT   4.
                               .WORD   ...V7
                       .ENDR
       000770  177777          .WORD   ...V7
       000772  177777          .WORD   ...V7
       000774  177777          .WORD   ...V7
       000776  177777          .WORD   ...V7
       001000          DXBEND::
                       .ENDC
   135
   136         000001          .END

Symbol table
```

```
ABTIO$  001000             DVC.VT  000015             O.GOOD  000554
BAREA   000256             DVM.DM  000002             O.RTRY  000620
BIOERR  000664R        003 DVM.DX  000001             O.SYWL  000562
BOOT    000606R        003 DVM.NF  000200             O.VEC   000600
BOOTF   000740R        003 DVM.NS  000001             O.WP    000634
BOOT1   000034R        003 DV2.V2  040000             O.WPF   000302
BOOT2   000200R        003 DXBEND  001000RG       003 PARVAL  001060R        002
BOTCSR  000122R        003 DXBOOT  000000RG       003 P1EXT   000432
BROFFS= 000220             DXCQE   000010RG       002 Q$BLKN  000000
BTCSR = 002256             DXDSIZ  000756             Q$BUFF  000004
BUFRAD  000276R        002 DXEND = 001134RG       002 Q$COMP  000010
BYTCNT  000306R        002 DXENT   000020R        002 Q$CSW   177776
B$BOOT  001000             DXFBLK  001106R        002 Q$FUNC  000002
B$DEVN  004716             DXINT   000456RG       002 Q$JNUM  000003
B$DEVU  004722             DXLQE   000006RG       002 Q$LINK  177774
B$DNAM  016330             DXNREG  000003             Q$MEM   000014
B$READ  004730             DXSTRT  000000RG       002 Q$PAR   000012
B2$     000170R        003 DXSTS   102022             Q$UNIT  000003
CHGTBL  001104R        002 DXSYS   000006RG       002 Q$WCNT  000006
CR      000015             DXT$O = 000000             Q.BLKN  000004
CRLFLF  000764R        003 DXWPRO  000016R        002 Q.BUFF  000010
CSDONE  000040             DXW1    000104R        002 Q.COMP  000014
CSEBUF  000002             DX$COD  000022             Q.CSW   000002
CSERR   100000             DX$CSR= 177170 G           Q.ELGH  000024
CSFBUF  000000             DX$CS2= 177174 G           Q.FUNC  000006
CSGO    000001             DX$END  001116RG       002 Q.JNUM  000007
CSINIT  040000             DX$NAM= 016300             Q.LINK  000000
CSINT   000100             DX$VC2= 000270 G           Q.MEM   000020
CSMAIN  000016             DX$VEC= 000264 G           Q.PAR   000016
CSRD    000006             EOF$    020000             Q.UNIT  000007
CSRDST  000012             ERL$G = 000000             Q.WCNT  000012
CSREAD  000002             ESCRC   000001             RDCMD   000130R        003
CSRX02  004000             ESDD    000100             RDX     000150R        003
CSTR    000200             ESDRY = 000200 G           READ    000224R        003
CSUNIT  000020             ESID    000004             READF   000370R        003
CSWRT   000004             ESPAR   000002             READFX  000424R        003
CSWRTD  000014             FILLCT  000721R        002 READS   000120R        003
DAREA   000244             FILST$  100000             READ1   000300R        003
DEVNAM  000254             FINDRV  000214             REETRY  000042R        003
DISCSR  000174             HDERR$  000001             REPORT  000672R        003
DRETRY  000034R        002 HNDLR$  004000             RETRY   000010
DVC.CT  000006             HRDBOT  000632R        003 RONLY$  040000
DVC.DE  000010             HS2.BI  000001             RTE$M = 000000
DVC.DK  000004             HS2.KI  000002             RTIRET  000076R        003
DVC.DL  000012             HS2.KL  000004             RXABRT  000746R        002
DVC.DP  000011             HS2.KU  000010             RXCSA   000504R        002
DVC.LP  000007             HS2.MO  000020             RXERR   000442R        002
DVC.MT  000005             INSCSR  000176             RXFUN2  000474R        002
DVC.NI  000013             INSDAT  000200             RXIENB  000434R        002
DVC.NL  000001             INSSYS  000202             RXINIT  000316R        002
DVC.PS  000014             IOERR   000753R        003 RXIRTN  000516R        002
DVC.SB  000020             JSW     000044             RXRTRY  000762R        002
DVC.SI  000016             LF      000012             RXTRY   000036R        002
DVC.SO  000017             MMG$T = 000001             SECTOR  000433R        002
DVC.TP  000003             N.WP    000640             SILOFE  001004R        002
DVC.TT  000002             O.BAD   000556             SPECL$  010000
DVC.UK  000000             O.CSR   000442             SPFUNC  100000
SPFUN$  002000             $MPPTR  001120RG       002 ...V15= 000340
SYSCHN  000017             $PTBYT  001124RG       002 ...V16= 000000
SYSPTR  000054             $PTWRD  001126RG       002 ...V17= 000000
TIM$IT= 000000             $P1EXT  000030R        002 ...V18= 000001
TPB     177566             $RLPTR  001116RG       002 ...V19= 000000
TPS     177564             .AUDIT  107123 G           ...V2 = 000103
TRACK   000432R        002 .DSTAT  000342             ...V20= 000000
TRKOFF  000400R        002 .DX     000021 G           ...V21= 000000
TRWAIT  000416R        003 .READ   000375             ...V22= 000000
UNITRD  000056R        003 .WRITE  000375             ...V27= 000000
VARSZ$  000400             ..READ  000010             ...V28= 000270
WAIT    000070R        003 ..WRIT  000011             ...V3 = 000002
WONLY$  020000             ...V10= 000040             ...V4 = 000000
X.WP    000270             ...V11= 000370             ...V5 = 000114
$FKPTR  001132RG       002 ...V12= 000370             ...V6 = 000270
$GTBYT  001122RG       002 ...V13= 000000             ...V7 = 177777
$INPTR  001130RG       002 ...V14= 000000             ...V9 = 000000
```

```
. ABS.  000660    000   (RW,I,GBL,ABS,OVR)
        000000    001   (RW,I,LCL,REL,CON)
DXDVR   001134    002   (RW,I,LCL,REL,CON)
DXBOOT  001000    003   (RW,I,LCL,REL,CON)
```

Figure A–2: DL Disk Handler

In the interests of clarity, code from the DL handler that does not apply to PDP–11 processors has been removed. Further, the contents of some of the macro expansions has been removed when those contents served no instructive purpose. In both cases, the removed lines are indicated by ellipses.

```
DL - RL01/RL02 Disk Handler     MACRO V05.05  Thursday 28-Feb-91 15:01

Table of contents

    CONDITIONAL ASSEMBLY SUMMARY
    MACROS AND DEFINITIONS
    *** THIS HANDLER SUPPORTS 2 UNITS ***
    HANDLER MACROS
    HARDWARE DEFINITIONS
    INSTALLATION CODE
    SET OPTIONS
    REQUEST ENTRY POINT
    INITIALIZE FOR TRANSFER, SET FUNCTION CODE, FIX WORD COUNT
    COMPUTE DISK ADDRESS AND START TRANSFER
    ENSURE THAT DISK IS ON TRACK BEFORE TRANSFER
    DLXFER - START AN I/O TRANSFER
    DLINT - INTERRUPT ENTRY POINT
    HANDLE THE ERRORS
    FINISH SUCCESSFUL OPERATION
    GET DEVICE SIZE
    DLXCT - FUNCTION EXECUTION ROUTINES
    DLSQUE - SETUP PSEUDO QUEUE ELEMENT
    DATA AREAS
    BOOTSTRAP DRIVER
    BOOTSTRAP READ ROUTINE
    BOOTSTRAP CONTINUED
    FETCH/LOAD CODE
```

Mapped monitor conditional:

```
                000001  MMG$T = 1

                        .MCALL .MODULE
       000000           .MODULE DL,VERSION=42,COMMENT=<RL01/RL02 Disk Handler>,AUDIT=YES

                        ;                       COPYRIGHT 1989, 1990 BY
                        ;           DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
                        ;                         ALL RIGHTS RESERVED
                        ;
                        ;THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
                        ;ONLY  IN  ACCORDANCE  WITH  THE TERMS  OF  SUCH  LICENSE AND WITH THE
                        ;INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY OTHER
                        ;COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
                        ;OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS HEREBY
                        ;TRANSFERRED.
                        ;
                        ;THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT NOTICE
                        ;AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT
                        ;CORPORATION.
                        ;
                        ;DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF ITS
```

```
                ;SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
```

Conditional Assembly Summary

```
                .SBTTL  CONDITIONAL ASSEMBLY SUMMARY
                ;+
                ;COND
                .
                .
                .
                ;       DL$UN   (2)             unit to support (additive only)
                ;               1-4             valid range
                ;
                ;       EIS$I   (MMG$T)         use SOB instruction (no code effects!)
                ;               0               simulate SOB
                ;               2               use SOB
                ;
                ;       DL$CSR  (174400)        CSR
                ;       DL$VEC  (160)           Vector

                ;
                ;       MMG$T                   std conditional
                ;       TIM$IT                  std conditional (no code effects)
                ;       ERL$G                   std conditional
                ;-
```

Preamble Section

Each macro you use in the handler requires the .MCALL statement, as line 6 shows. Note that .DRDEF issues many of the .MCALL statements for you so you need not explicitly call them.

Macros and Definitions

```
 1
 2                      .SBTTL  MACROS AND DEFINITIONS
 3
 4                      .ENABL  LC
 5
 6                      .MCALL  .DRDEF, .MFPS, .MTPS, .ASSUME, .ADDR, .BR
```

A call is made to a macro (.UBVDF) in the system definition library SYSTEM.MLB. SYSTEM.MLB is always found on logical device SRC:

```
 7
 8                      .LIBRARY "SRC:SYSTEM.MLB"
 9              .MCALL  .UBVDF
10 000000               .UBVDF
11
```

Various monitor offsets and locations are defined with mnemonics so that references to them can be found easily:

```
12              ; VECTOR DEFINITIONS
13
14      000004          NXM.V   =: 4                    ;NON-EXISTENT MEMORY TRAP VECTOR
15      000020          IOT.V   =: 20                   ;IOT TRAP VECTOR
16
                .
                .
                .
29
30              ; SYSTEM GENERATION OPTION
31
32              .IIF NDF DL$UN, DL$UN   == 2            ;NUMBER OF UNITS SUPPORTED
33              .IIF GT DL$UN-4, DL$UN  == 4            ;CAN'T HAVE MORE THAN 4 UNITS
34              .IIF LE DL$UN,  DL$UN   == 1            ;CAN'T HAVE NO UNITS
35
36                      .IRP    X,<\DL$UN>
```

Handler Unit Support

```
                        .SBTTL  *** THIS HANDLER SUPPORTS X UNITS ***
                        .ENDR

                ; SPECIAL FUNCTION DEFINITIONS
                ; ALL SPECIAL FUNCTIONS ARE DMA EXCEPT FOR FN$SIZ AND FN$GET
                ; FN$WRT AND FN$RED GO IN UBTAB. FN$REP USES A PERMANENT UMR

                        FN$GET =: 370           ;GET DEVICE STATUS
        000373          FN$SIZ  =: 373                  ;GET DEVICE SIZE
        000374          FN$REP  =: 374                  ;FORCE RE-READ OF REPLACEMENT TABLE
```

Use the replacement table with:

```
        000376          FN$WRT  =: 376                  ;ABSOLUTE WRITE (NO BAD BLOCK)
        000377          FN$RED  =: 377                  ;ABSOLUTE READ  (REPLACEMENT)
                ;NOTE: if you add a SPFUN code also add it to .DRSPF

                ; ERROR LOGGING DEFINITIONS

        000010          DLRCNT  =: 8.                   ;ERROR RETRY COUNT
        000006          DLREG   =: 6                    ;REGISTERS TO LOG ON ERROR

                ; RL11/RL01 PARAMETERS
                ; GEOMETRY:     256 CYLINDERS (512 ON RL02)
                ;               2   TRACKS PER CYLINDER
                ;               20  BLOCKS PER TRACK
                ;               2   128-WORD SECTORS PER BLOCK

        000024          DLBPT   =: 20.                  ;NUMBER OF BLOCKS PER TRACK
        012000          DLWPT   =: 256.*DLBPT           ;WORDS PER TRACK
        000012          DLNBAD  =: 10.                  ;NUMBER ALLOWABLE BAD BLOCKS PER DISK
        023742          DLSIZE  =: <256.*2-1>*DLBPT-DLNBAD ;BLOCKS PER RL01 (LESS BSF)
        047742          DLSIZ2  =: <512.*2-1>*DLBPT-DLNBAD ;BLOCKS PER RL02 (LESS BSF)
        000052          DLTSIZ  =: DLNBAD*4.+2          ;SIZE OF BAD BLOCK TABLE
                                                        ; (PLUS END OF TABLE FENCE)

                ; UB DEFINITIONS

                ;  FIXED OFFSETS EQUATES (.FIXDF)

        000404          $PNPTR  =:      000404  ;RMON OFFSET OF PNAME TABLE
        000432          P1$EXT  =:      000432  ;RMON OFFSET OF $P1EXT ADDRESS
        000460          $H2UB   =:      000460  ;RMON OFSET OF UB ENTRY VECTOR PTR

                ;  EXTENDED MEMORY SUBROUTINE OFFSETS FROM $P1EXT  (.PIXDF)

        177752          $MPMEM  =:      -22.    ;OFFSET TO MAP KT-11 VIRTUAL TO PHYSICAL

                ;  UB ENTRY VECTOR EQUATES (.UBVDF)

                ;       UB.IDV  =:      0               ; IDENTIFICATION WORD
                ;       UB.VDV  =:      <^rUBV>         ; IDENTIFICATION WORD VALUE
                ;       UB.GET  =:      2               ; JUMP TO GETUMR
                ;       UB.ALL  =:      6               ; JUMP TO ALLUMR
                ;       UB.RLS  =:      12              ; JUMP TO RLSUMR

                ;  DL INTERNAL DMA BUFFER EQUATES

        000054          BUFSIZ  =: 54/2*DL$UN   ; SIZE OF DL INTERNAL DMA BUFFER
                                                ; WORD SIZE OF DLBBUF*DL$UN
        000001          NOUMRS  =: <BUFSIZ+7777/10000> ; NUMBER OF PERMANENT UMRS REQUIRED
```

The .DRDEF performs much of the work of the preamble section. It is called with different parameters depending on whether or not the handler supports memory mapping (MMG$T=1). The following includes much of the macro expansion:

Handler Macros

```
                        .SBTTL  HANDLER MACROS

                .IF     EQ      MMG$T
```

The .DRDEF macro (with macro expansion) for unmapped monitors:

```
                        .DRDEF  DL,5,FILST$!SPFUN$!VARSZ$,DLSIZE,174400,160,DMA=NO
                .MCALL  .DRAST,.DRBEG,.DRBOT,.DREND,.DREST,.DRFIN,.DRFMS,.DRFMT
                .MCALL  .DRINS,.DRPTR,.DRSET,.DRSPF,.DRTAB,.DRUSE,.DRVTB
                .MCALL  .FORK,.QELDF
                .IIF NDF RTE$M RTE$M=0
                .IIF NE RTE$M RTE$M=1
                .IIF NDF TIM$IT TIM$IT=0
                .IIF NE TIM$IT TIM$IT=1
                .IIF NDF MMG$T MMG$T=0
                .IIF NE MMG$T MMG$T=1
                .IIF NDF ERL$G ERL$G=0
                .IIF NE ERL$G ERL$G=1
                .IIF NE TIM$IT, .MCALL  .TIMIO,.CTIMI
  000000        .QELDF
                .IIF NDF MMG$T,MMG$T=1
                .IIF NE MMG$T,MMG$T=1
        000000  Q.LINK=:0
        000002  Q.CSW=:2.
        000004  Q.BLKN=:4.
        000006  Q.FUNC=:6.
        000007  Q.JNUM=:7.
        000007  Q.UNIT=:7.
        000010  Q.BUFF=:^o10
        000012  Q.WCNT=:^o12
        000014  Q.COMP=:^o14
                .IRP    X,<LINK,CSW,BLKN,FUNC,JNUM,UNIT,BUFF,WCNT,COMP>
                Q$'X=:Q.'X-^o4
                .ENDR
        177774  Q$LINK=:Q.LINK-^o4
        177776  Q$CSW=:Q.CSW-^o4
        000000  Q$BLKN=:Q.BLKN-^o4
        000002  Q$FUNC=:Q.FUNC-^o4
        000003  Q$JNUM=:Q.JNUM-^o4
        000003  Q$UNIT=:Q.UNIT-^o4
        000004  Q$BUFF=:Q.BUFF-^o4
        000006  Q$WCNT=:Q.WCNT-^o4
        000010  Q$COMP=:Q.COMP-^o4
                .IF EQ MMG$T
        000016  Q.ELGH=:^o16
                .IFF
                Q.PAR=:^o16
                Q.MEM=:^o20
                .IRP    X,<PAR,MEM>
                Q$'X=:Q.'X-^o4
                .ENDR
                Q.ELGH=:^o24
                .ENDC
        000001  HDERR$=:1
        020000  EOF$=:^o20000
        000400  VARSZ$=:^o400
        001000  ABTIO$=:^o1000
        002000  SPFUN$=:^o2000
        004000  HNDLR$=:^o4000
        010000  SPECL$=:^o10000
        020000  WONLY$=:^o20000
        040000  RONLY$=:^o40000
        100000  FILST$=:^o100000
        023742  DLDSIZ=:DLSIZE
        000005  DL$COD=:5
        102405  DLSTS=:<5>!<FILST$!SPFUN$!VARSZ$>
                .IIF NDF DL$VEC,DL$VEC=160
                .GLOBL  DL$VEC
                .
                .
                .
```

The .DRPTR macro with no parameters:

```
                     .DRPTR
          .
          .
          .
          .IFF    ;EQ MMG$T
```

The .DRDEF macro (with macro expansion) for mapped monitors. The handler is defined for the RL01; if it is for an RL02, the size is changed later. Note that handler supports UMRs.

```
   000000                 .DRDEF  DL,5,FILST$!SPFUN$!VARSZ$,DLSIZE,174400,160,DMA=YES,PERMUMR=NOUMRS
```

The .DRPTR macro with parameters:

```
   000200                 .DRPTR  FETCH=FETCH,LOAD=FETCH,RELEASE=RELEAS,UNLOAD=RELEAS
                  .ENDC   ;EQ     MMG$T
```

The .DREST macro (with macro expansion). Argument *REPLACE=RTABLE* shows DL does a software bad-block replacement—see installation code:

```
   000022                 .DREST  CLASS=DVC.DK,REPLACE=RTABLE
           000000  DVC.UK  =:0
           000001  DVC.NL  =:1
           000002  DVC.TT  =:^o2
           000003  DVC.TP  =:^o3
           000004  DVC.DK  =:^o4
           000005  DVC.MT  =:^o5
           000006  DVC.CT  =:^o6
           000007  DVC.LP  =:^o7
           000010  DVC.DE  =:^o10
           000011  DVC.DP  =:^o11
           000012  DVC.DL  =:^o12
           000013  DVC.NI  =:^o13
           000014  DVC.PS  =:^o14
           000015  DVC.VT  =:^o15
           000016  DVC.SI  =:^o16
           000017  DVC.SO  =:^o17
           000020  DVC.SB  =:^o20

           000001  DVM.NS  =:1
           000001  DVM.DX  =:1
           000002  DVM.DM  =:^o2
           000200  DVM.NF  =:^o200

           040000  DV2.V2  =:^o40000

           000001  HS2.BI  =:1
           000002  HS2.KI  =:^o2
           000004  HS2.KL  =:^o4
           000010  HS2.KU  =:^o10
           000020  HS2.MO  =:^o20
                   .
                   .
                   .
```

Point to special functions for UNIBUS mapping register support:

```
                  .IF NE MMG$T
   000076                 .DRSPF  +UBTAB                 ;SPFUN FOR UB GOES IN TABLE UBTAB
                  .ENDC ;NE MMG$T

```

Define special functions:

```
   000032                 .DRSPF
                          .DRSPF  <FN$GET>               ;GET DEVICE STATUS
   000032                 .DRSPF  <FN$SIZ>               ;GET DEVICE SIZE
   000032                 .DRSPF  <FN$REP>               ;FORCE RE-READ OF REPLACEMENT TABLE
   000032                 .DRSPF  <FN$WRT>               ;ABSOLUTE WRITE (NO BAD BLOCK)
   000032                 .DRSPF  <FN$RED>               ;ABSOLUTE READ  (REPLACEMENT)

                  .IIF    NDF     EIS$I   EIS$I = MMG$T
                  .IIF    EQ      EIS$I   .MCALL  SOB
```

Define hardware offsets:

```
                        .SBTTL  HARDWARE DEFINITIONS

                ; RL11 DEVICE REGISTER OFFSETS

                                                ;DEFINE THE OFFSETS
        000000          RLCS    =: 0                            ;CONTROL STATUS REGISTER
        000002          RLBA    =: 2                            ;BUS ADDRESS REGISTER
        000004          RLDA    =: 4                            ;DISK ADDRESS REGISTER
        000006          RLMP    =: 6                            ;MULTI-PURPOSE REGISTER
        000010          RLBAE   =: 10                           ;BUS ADDRESS REGISTER (EXTENDED)
        .
        .
        .

                ; RLCS BIT ASSIGNMENTS

        100000          CSERR   =: 100000                       ;ERROR SUMMARY
        040000          CSDE    =: 040000                       ;DRIVE ERROR
        036000          CSERRC  =: 036000                       ;ERROR CODE MASK
        020000          CSNXM   =: 020000                       ;NON-EXISTENT MEMORY
        010000          CSDLT   =: 010000                       ;DATA LATE
        010000          CSHNF   =: 010000                       ;HEADER NOT FOUND
        004000          CSDCRC  =: 004000                       ;DATA CRC ERROR
        004000          CSHCRC  =: 004000                       ;HEADER CRC ERROR
        002000          CSOPI   =: 002000                       ;OPERATION INCOMPLETE
        001400          CSDS01  =: 001400                       ;DRIVE SELECT BITS 0 AND 1
        000400          CSDS0   =: 000400                       ;DRIVE SELECT BIT 0
        000200          CSCRDY  =: 000200                       ;CONTROLLER READY
        000100          CSIE    =: 000100                       ;INTERRUPT ENABLE
        000040          CSBA17  =: 000040                       ;BUS ADDRESS BIT 17
        000020          CSBA16  =: 000020                       ;BUS ADDRESS BIT 16
        000016          CSFUN   =: 000016                       ;FUNCTION CODE
        000001          CSDRDY  =: 000001                       ;DRIVE READY

                ; RLCS FUNCTION CODE VALUES

        000000          FNNOP   =: 0*2                          ;NO OPERATION
        000002          FNWCHK  =: 1*2                          ;WRITE CHECK
        000004          FNGSTS  =: 2*2                          ;GET DRIVE STATUS
        000006          FNSEEK  =: 3*2                          ;SEEK
        000010          FNRDH   =: 4*2                          ;READ HEADERS
        000012          FNWRITE =: 5*2                          ;WRITE DATA
        000014          FNREAD  =: 6*2                          ;READ DATA
        000016          FNRDNH  =: 7*2                          ;READ DATA WITH NO HEADER CHECK

                ; RLMP GET STATUS RETURNED BIT ASSIGNMENTS

        100000          STWDE   =: 100000                       ;WRITE DATA ERROR
        040000          STCHE   =: 040000                       ;CURRENT HEAD ERROR
        020000          STWL    =: 020000                       ;WRITE LOCK STATUS
        010000          STSKTO  =: 010000                       ;SEEK TIMEOUT ERROR
        004000          STSP    =: 004000                       ;SPEED ERROR
        002000          STWGE   =: 002000                       ;WRITE GATE ERROR

        001000          STVC    =: 001000                       ;VOLUME CHECK
        000400          STDSE   =: 000400                       ;DRIVE SELECT ERROR
        000200          STDT    =: 000200                       ;DRIVE TYPE
        000100          STHS    =: 000100                       ;HEAD SELECT STATUS
        000040          STCO    =: 000040                       ;COVER OPEN
        000020          STHO    =: 000020                       ;HEADS HOME
        000010          STBH    =: 000010                       ;BRUSHES HOME
        000007          STST    =: 000007                       ;STATE BIT MASK
        000005          STSLM   =: 000005                       ;DRIVE IN SEEK-LINEAR MODE STATE

                ; RLDA BIT VALUES FOR SEEK COMMANDS

        077600          SKCADF  =: 077600                       ;CYLINDER ADDRESS DIFFERENCE
        000200          SKCA0   =: 000200                       ;CYLINDER ADDRESS DIFFERENCE BIT 0
        000020          SKHS    =: 000020                       ;HEAD SELECT (SURFACE 0 OR 1)
        000004          SKDIR   =: 000004                       ;DIRECTION (0 => OUTWARD, 1 => INWARD)
        000001          SKMARK  =: 000001                       ;MARK BIT MUST BE 1 TO INDICATE A SEEK

                ; RLDA BIT VALUES FOR I/O COMMANDS

        077600          IOCA    =: 077600                       ;CYLINDER ADDRESS
```

```
        000200          IOCA0   =: 000200               ;CYLINDER ADDRESS BIT 0
        000100          IOHS    =: 000100               ;HEAD SELECT
        000077          IOSA    =: 000077               ;SECTOR ADDRESS MASK

                ; RLDA BIT VALUES FOR GET STATUS COMMAND

        000010          GSRST   =: 000010               ;RESET DRIVE
        000002          GSGS    =: 000002               ;GET STATUS INDICATOR MUST BE 1
        000001          GSMARK  =: 000001               ;THIS MUST BE 1 TO INDICATE GET STATUS

```

More RMON references:

```
                ; RMON REFERENCES

        000054  SYSPTR  =:      54              ; SYSCOM pointer to RMON
        000370          CONFG2  =:      370     ; second configuration word
        000100                  BUS$    =:      000100 ;
        020000                  PROS$   =:      020000 ;
        020100                          BUS$M   =:      BUS$!PROS$      ;Mask for type bits
        020100                          BUS$X   =:      BUS$!PROS$      ;Strange (busless) KXJ
        020000                          BUS$C   =:      PROS$           ;CTI bus
        000100                          BUS$Q   =:      BUS$            ;QBUS
        000000                          BUS$U   =:      0               ;UNIBUS


        000375  .READ   =:      375             ; EMT code for .READ
        000010          ..READ  =:      010     ; subcode for .READ
        000375  .WRITE  =:      375             ; EMT code for .WRITE
        000011          ..WRIT  =:      011     ; subcode for .WRITE

        000017  SYSCHN  =:      17              ; system channel
```

Installation checks (RL01/02 run on UNIBUS or Q-bus only):

```
                        .SBTTL  INSTALLATION CODE

 000032                 .DRINS  DL

 000200 000401          BR      10$             ;Data device installation check
 000202                                         .ASSUME . EQ INSSYS
 000202 000414          BR      20$             ;System device installation check (none)

 000204 013700  10$:    MOV     @#SYSPTR,R0     ; get address of RMON
        000054
 000210 016000          MOV     CONFG2(R0),R0   ;Get configuration word for BUS check
        000370
 000214 042700          BIC     #^C<BUS$M>,R0   ;Isolate bus bits
        157677
 000220 022700          CMP     #<BUS$X>,R0     ;Running on KXJ?
        020100
 000224 001404          BEQ     30$             ;Yes, don't install
 000226 022700          CMP     #<BUS$C>,R0     ;CTI?
        020000
 000232 001401          BEQ     30$             ;Yes, don't install
 000234 005727  20$:    TST     (PC)+           ; clear carry, skip setting carry
 000236 000261  30$:    SEC                     ; set carry
 000240 000207          RETURN
```

The following is SET code. If there is insufficient room in the SET code area, some code can be moved up into the installation code area.

```

 000242         O.SYWL:
 000242 011600          MOV     @SP,R0                  ; copy return address
 000244 005200          INC     R0                      ; point to opcode at return
 000246 122720          CMPB    #BR/400,(R0)+           ; is it a BR xxx?
        000001
 000252 001135          BNE     O.BAD                   ; NO, old style SET
 000254 010016          MOV     R0,@SP                  ; use alternate return (RET+2)
 000256 000533          BR      O.BAD                   ; with carry set

```

The following sets up the table for software bad-block replacement:

```
000260     002  RTABLE: .BYTE   2,10.,5.,2.,40.,1.      ; Replacement factors table
000261     012
000262     005
000263     002
000264     050
000265     001
```

All blocks can be replaced. This defines the geometry of the disk:

```
                                                        ; all replacable
                                                        ; 10. blocks to skip
                                                        ; 5. sectors of bad sector file
                                                        ; 2. tracks per cylinder
                                                        ; 40. sectors per track
                                                        ; 2**1 sectors per block
```

Installation code area size check:

```
000266          .Assume . LE 400,MESSAGE=<;Install code overflow>
```

The DL handler supports several SET command conditions:

Set Options

```
                        .SBTTL  SET OPTIONS

000266                  .DRSET  CSR,    160000, O.CSR,  OCT
000412                  .DRSET  VECTOR, 500,    O.VEC,  OCT

000422                  .DRSET  RETRY,  127.,   O.RTRY, NUM

                        .IF NE ERL$G
                        .DRSET  SUCCES, -1,     O.SUCC, NO
                        .ENDC ;NE ERL$G

        004124          BTCSR   = <DLEND-DLSTRT>+<BOTCSR-DLBOOT>+1000

                ; SET DL CSR=address

000432  020003  O.CSR:  CMP     R0,R3                   ;CSR IN RANGE?
000434  103444          BLO     O.BAD                   ;NOPE...
000436  010067          MOV     R0,INSCSR               ;YES, INSTALLATION CODE NEEDS IT
        177534
000442  010067          MOV     R0,DISCSR               ;AND RESORC DOES TOO
        177526

                ; When the CSR is changed, we must also alter the bootstrap so
                ; that it will use the correct CSR.

                                                        ;R1->READ/WRITE EMT AREA
000446                  .ADDR   #BAREA+4,R1             ; (BUFFER ADDRESS WORD)
                                                        ;R2->BUFFER
000454                  .ADDR   #1000,R2                ; (OVERWRITES CORE COPY OF BLOCK 1)
000462  010211          MOV     R2,(R1)                 ;SET THE BUFFER ADDRESS
000464  012741          MOV     #BTCSR/1000,-(R1)       ; THE BLOCK TO READ/WRITE
        000004
                                                        ; (BOOT BLOCK THAT NEEDS ALTERING)
000470  005741          TST     -(R1)                   ;R1->EMT AREA
000472  010003          MOV     R0,R3                   ;SAVE CSR ELSEWHERE, EMT NEEDS R0
000474  010100          MOV     R1,R0                   ;R0->EMT AREA FOR READ
000476  104375          EMT     .READ                   ; *** (.READW) ***
000500  103422          BCS     O.BAD
000502  010362          MOV     R3,<BTCSR&777>(R2)      ;SET THE NEW CSR
        000124
000506  010100          MOV     R1,R0                   ;R0->EMT AREA FOR WRITE
000510                                                  .ASSUME ..READ+1 EQ ..WRIT
000510  105260          INCB    1(R0)                   ;CHANGE FROM 'READ' TO 'WRITE'
        000001
000514  104375          EMT     .WRITE                  ; *** (.WRITW) ***
000516  103651          BCS     O.SYWL
000520  010100          MOV     R1,R0                   ;R0->EMT AREA (LAST TIME, HONEST)
```

```
44 000522                                               .ASSUME ..WRIT-1 EQ ..READ
45 000522  105360          DECB    1(R0)                ;CHANGE FROM 'WRITE' TO 'READ'
           000001
46 000526  012760          MOV     #1,2(R0)             ; OF HANDLER BLOCK 1
           000001
           000002
47 000534  104375          EMT     .READ                ; *** (.READW) ***
48 000536  103403          BCS     O.BAD
49 000540  010367          MOV     R3,DLCSR             ;TELL HANDLER ABOUT NEW CSR

           000032'
50 000544  005727  O.GOOD: TST     (PC)+                ;GOOD RETURN (CARRY CLEAR)
51 000546  000261  O.BAD:  SEC                          ;ERROR RETURN (CARRY SET)
52 000550  000207          RETURN
53
54                 ; SET DL VECTOR=address
55
56 000552  020003  O.VEC:  CMP     R0,R3                ;VECTOR IN RANGE? (<500)
57 000554  103374          BHIS    O.BAD                ;NOPE...
58 000556  032700          BIT     #3,R0                ;YES, BUT ON A VECTOR BOUNDRY?
           000003
59 000562  001371          BNE     O.BAD                ;NOPE...
60 000564  010067          MOV     R0,DLSTRT            ;TELL HANDLER ABOUT NEW VECTOR
           000000'
61 000570  000765          BR      O.GOOD
62
63                 ; SET DL RETRY=count
64
65 000572  020003  O.RTRY: CMP     R0,R3                ;Test retry limits
66 000574  101364          BHI     O.BAD                ;Branch if out of bounds
67 000576  010067          MOV     R0,DRETRY            ;Store the user selected retry count
           000742'
68 000602  001761          BEQ     O.BAD                ;Zero retries not allowed
69 000604  000757          BR      O.GOOD               ;Otherwise, good
70
71                         .IF NE ERL$G
72
73                 ; SET DL [NO]SUCCES
74
75                 O.SUCC: MOV     #0,R3                ;'SUCCESS' ENTRY POINT
76                                                      ; (MUST BE TWO WORDS)
77                         MOV     R3,SCSFLG            ;'NOSUCCESS' ENTRY POINT
78                         BR      O.GOOD
79                         .ENDC ;NE ERL$G
80
81 000606     017  BAREA:  .BYTE   SYSCHN,..READ        ;CHANNEL 17, READ
   000607     010
82 000610                  .BLKW                        ;BLOCK NUMBER
83 000612                  .BLKW                        ;BUFFER ADDRESS
84 000614  000400          .WORD   256.                 ;WORD COUNT
85 000616  000000          .WORD   0                    ;COMPLETION (WAIT)
86
```

SET code overflow check:

```
87 000620          .Assume . LE 1000,MESSAGE=<;Set area overflow>
88                 .ENDC
```

Header Section

Request Entry Point

```
 1                         .SBTTL  REQUEST ENTRY POINT
 2
 3                         .ENABL  LSB
 4
 5         .IF EQ MMG$T
```

The .DRBEG macro for unmapped monitors:

```
 6                         .DRBEG  DL
 7                 .IFF ;EQ MMG$T
```

The .DRBEG macro for mapped monitors:

```
  000620                  .DRBEG  DL,SPFUN=UBTAB
                    .ENDC ;EQ MMG$T
```

I/O Initiation Section

```
          000006' DLBASE=DLSTRT+6

  000024  016705          MOV     DLCQE,R5                ;POINT TO CURRENT QUEUE ELEMENT
          177760
  000030  012704          MOV     (PC)+,R4                ;POINT TO CONTROLLER CSR
  000032                                                  .ASSUME .-DLSTRT LT 1000
  000032  174400  DLCSR:   .WORD  DL$CSR                  ;ADDRESS OF CONTROLLER
  000034  016500          MOV     Q$FUNC(R5),R0           ;GET FUNCTION CODE / UNIT NUMBER
          000002
  000040  110002          MOVB    R0,R2                   ;GET SPECIAL FUNCTION CODE
                  .
                  .
                  .
  000042  120227          CMPB    R2,#FN$SIZ              ;.SPFUN LESS THAN 373 (SIGNED BYTE)
          000373
  000046  002403          BLT     5$                      ;YES, .SPFUN 200 THRU 372 INVALID
  000050  120227          CMPB    R2,#FN$REP+1            ;IS THIS .SPFUN 375
          000375
  000054  001002          BNE     10$                     ;NO, HAVE VALID SPFUN REQUEST
  000056  000167  5$:     JMP     DLQCOM                  ;DISMISS QUEUE REQUEST
          001572

  000062          10$:

  000062  042700          BIC     #^C<7*400>,R0           ;ISOLATE UNIT NUMBER BITS
          174377
  000066  020027          CMP     R0,#DL$UN*400           ;DO WE SUPPORT THIS UNIT?
          001000
  000072  103136          BHIS    DLELNK                  ;NO, ERROR NOW

  000074  010067          MOV     R0,DLUNIT               ;SAVE UNIT NUMBER
          001062
  000100                                                  .ASSUME CSDS01  EQ      3*400
  000100  012767          MOV     #FNREAD!CSIE,DLCODE     ;ASSUME READ (FOR TABLE)
          000114
          001050

                          .IF NE  MMG$T
  000106  120227          CMPB    R2,#FN$SIZ              ;SEE IF .SPFUN GET SIZE
          000373
  000112  001407          BEQ     15$                     ;YES -- DON'T CHANGE Q.BUFF AND Q.PAR
  000114                                                  .ASSUME Q$BLKN+4 EQ Q$BUFF
  000114  022525          CMP     (R5)+,(R5)+             ;POINT TO Q.BUFF IN QUEUE ELEMENT
  000116                                                  .ASSUME Q$BUFF+2 EQ Q$WCNT ; done by MPPTR
  000116  004777          CALL    @$MPPTR                 ;CONVERT ADDRESS TO 18 BIT PHYSICAL

          002160
  000122                                                  .ASSUME Q$WCNT-2 EQ Q$BUFF
  000122  012645          MOV     (SP)+,-(R5)             ;REPLACE Q.BUFF WITH BITS <15:00>
  000124                                                  .ASSUME Q$BUFF-4 EQ Q$BLKN
  000124  024545          CMP     -(R5),-(R5)             ;FIX QUEUE ELEMENT POINTER
  000126  012665          MOV     (SP)+,Q$PAR(R5)         ;SAVE BITS <21:16> IN Q.PAR WORD
          000012
                          .ENDC ;NE MMG$T
```

The software bad-block replacement table is named *DLBBUF*:

```
  000132          15$:
  000132                  .ADDR   #DLBBUF-<DLTSIZ+2>,R3   ; GET BIASED ADDRESS OF TABLE BUFFER
  000140  000300          SWAB    R0                      ;GET UNIT NUMBER
  000142  062703  20$:    ADD     #DLTSIZ+2,R3            ;POINT TO NEXT UNIT'S TABLE
          000054
  000146  005300          DEC     R0                      ; REDUCE UNIT NUMBER
  000150  100374          BPL     20$                     ; ALL GONE?
  000152  010327          MOV     R3,(PC)+                ;SAVE POINTER TO UNIT'S
  000154  000000  DLCC:    .WORD  0                       ; CURRENT CYLINDER TABLE (LOW ADDR)
  000156  005723          TST     (R3)+                   ;POINT TO REPLACEMENT TABLE
  000160                                                  .ASSUME .+4 EQ DLUSIZ
```

```
000160  012727          MOV     #DLSIZE,(PC)+           ;ASSUME RL01
        023742
                .
                .
                .
```

Test for RL01 or RL02; select correct size:

```
000164  000000  DLUSIZ:  .WORD  0

000166  004767          CALL    DLGST                   ;GET DISK STATUS
        001546
000172  105701          TSTB    R1                      ;SINGLE DENSITY?
000174  100003          BPL     25$                     ;IF ZERO, RL01 SINGLE DENSITY
000176  012767          MOV     #DLSIZ2,DLUSIZ          ;IF SET, RL02 DOUBLE DENSITY
        047742
        177760
000204  005700  25$:    TST     R0                      ;Now, error in get status?
000206  100403          BMI     30$                     ;Yes, invalidate everything
000210  032701          BIT     #STVC,R1                ;IS THERE A NEW DISK IN THIS DRIVE?
        001000
000214  001403          BEQ     35$                     ;NO, SAME AS LAST TIME
000216  012743  30$:    MOV     #-1,-(R3)               ;INVALIDATE CURRENT CYLINDER
        177777
000222  012313          MOV     (R3)+,@R3               ; AND INVALIDATE REPLACEMENT TABLE
                .IFF
                        CMPB    R2,#FN$GET              ;SEE IF .SPFUN GET SPECIAL STATUS
                        BEQ     DLGSTA                  ;YES, GO DO IT!
                        CALL    DLGST                   ;GET DISK STATUS (NORMAL)
                        TST     R0                      ;Now, error in get status?
                        BMI     30$                     ;Yes, invalidate everything
                        CALL    INVVC                   ;INVALIDATE IF VOLUME CHECK ON
                        BR      35$                     ;SKIP NEXT

                30$:    CALL    INVAL                   ;UNCONDITIONAL INVALIDATION
                .ENDC

```

Following code decides if we use bad-block replacement table (only for special functions). DLSQUE, DLADDR, and DLEXFR are used for replacement table read.

```
000224  120227  35$:    CMPB    R2,#FN$REP              ;CHECK OUT THE SPECIAL FUNCTION

        000374
000230  002002          BGE     40$                     ;BRANCH IF NOT 'GET SIZE'
                                                        ; (NOTE SIGNED COMPARE)
000232  000167          JMP     DLGSIZ                  ;GO DO 'GET SIZE'
        001434

000236  001410  40$:    BEQ     50$                     ;GO READ BAD-BLOCK REPLACEMENT TABLE
000240  101045          BHI     55$                     ;GO DO ABSOLUTE BLOCK READ/WRITE
000242  005765          TST     Q$WCNT(R5)              ;NORMAL REQUEST, SEEK?
        000006
000246  001002          BNE     45$                     ;BRANCH IF NOT
000250  000167  DLFLNK: JMP     DLQCOM                  ;.DRFIN TIME
        001400

000254  005713  45$:    TST     @R3                     ;IS TABLE IN MEMORY YET?
000256  100046          BPL     DLTRAN                  ;YES, WE CAN GO DO THE TRANSFER
```

Reread the replacement table.

1. Read replacement table into memory if it's not there.
   a. Save current queue element.
   b. Build pseudoqueue element to read the replacement table (DLSQUE).
   c. Allow transfer to start (DLADDR).
   d. Eventually, the request gets to the end of the I/O initiation section and returns to monitor.

e. Request is completed and returns to interrupt entry (.DRAST).

f. Continues down to DLEXFR to determine if we were rereading the table and dismiss the the pseudoqueue element if we were. The queue element that prompted the reading of the replacement table still exists. It can now be processed.

2. Replacement table already in memory—use it. Go to DLTRAN to use it.

```
                            ;
                            ; WE ALWAYS COME HERE TO REREAD THE REPLACEMENT TABLE
                            ;
000260                      50$:
                            .IF NE MMG$T
000260  010346                      MOV     R3,-(SP)                ;SAVE R3
000262                              .ADDR   #DLBBUF,R3              ;R3=PIC ADDRESS OF START OF DLBBUF
000270  016701                      MOV     DLCC,R1                 ;R1=START ADDRESS FOR THIS UNIT
        177660
000274  160301                      SUB     R3,R1                   ;R1=OFFSET INTO DLBBUF FOR THIS UNIT
000276  062701                      ADD     #2,R1                   ;POINT TO REPLACEMENT TABLE
        000002
000302  016702                      MOV     BUFADH,R2               ;GET HI ORDER DLBBUF ADDRESS
        001760
000306  016703                      MOV     BUFADL,R3               ;GET LOW ORDER DLBBUF ADDRESS
        001756
000312  060103                      ADD     R1,R3                   ;R3=THIS UNIT'S START ADDR IN UMR
000314  103002                      BCC     52$                     ;BRANCH IF NO CARRY
000316  062702                      ADD     #CSBA16,R2              ;ADD CARRY TO HI ORDER ADDR
        000020
000322  010267              52$:    MOV     R2,DLBPAR               ;PUT HI ORDER ADDR INTO PSEUDO QEL
        001550
                            ;       MOV     Q$MEM(R5),DLBMEM        ;PUT Q$MEM INTO PSEUDO QEL (NOT NEEDED)
                            .ENDC ;NE MMG$T
000326  012701                      MOV     #1,R1                   ;TABLE IS IN BLOCK 1
        000001
000332  012702                      MOV     #DLTSIZ/2,R2            ;WORDS TO READ (TABLE SIZE)
        000025
```

Build queue element to read table.

```
000336  004767                      CALL    DLSQUE                  ;SET UP REST OF PSEUDO QUEUE ELEMENT
        001470


                            .IF NE MMG$T
000342  012603                      MOV     (SP)+,R3                ;RESTORE R3 (ADDR FOR MOV'S)
                            .ENDC ;NE MMG$T
000344  012713                      MOV     #-1,@R3                 ;FLAG THAT THERE IS NO TABLE IN MEMORY
        177777
000350  011343                      MOV     @R3,-(R3)               ;VOID CURRENT CYLINDER, TOO
```

Read in the table.

At DLADDR, pseudoqueue element is processed to read in replacement table. I/O initiation will start transfer and return to the monitor. When transfer is complete, the .DRAST section is entered to dismiss the pseudoqueue element.

```
000352  000512                      BR      DLADDR                  ;COMPUTE DISK ADDRESS AND START THE
                                                                    ; TABLE READ
```

```
138 000354  105202  55$:    INCB    R2                      ;ABSOLUTE BLOCK READ?
139 000356                                                  .ASSUME FN$RED  EQ      377
140 000356  001510          BEQ     DLADDR                  ;YES, WE ARE ALL SET UP
141 000360  012767          MOV     #FNWRITE!CSIE,DLCODE    ;SET WRITE FUNCTION CODE
            000112
            000570
142 000366  000504          BR      DLADDR                  ;GO DO IT
143
144                         .IF NE ERL$G
145                 .ASSUME .-DLSTRT LT 1000
146                 SCSFLG: .WORD   0                       ; :SUCCESS LOGGING FLAG (DEFAULT=YES)
147                                                         ; =0 - LOG SUCCESSES
148                                                         ;<>0 - DON'T LOG SUCCESSES
149                         .ENDC ;NE ERL$G
150
151                         .DSABL  LSB

                    .
                    .
                    .
 74
 75 000370  000167  DLELNK: JMP     DLEROR                          ;LINK TO FATAL ERROR
            001212
```

Set up and perform I/O:

```
  1                         .SBTTL  INITIALIZE FOR TRANSFER, SET FUNCTION CODE, FIX WORD COUNT
  2
  3                 ;+
  4                 ; SET READ OR WRITE FUNCTION CODE
  5                 ; IF TRANSFER HAS REPLACED BLOCKS IN IT, BREAK IT INTO PIECES AND
  6                 ;    SEND EACH PIECE TO DLADDR SEPARATELY FOR I/O
  7                 ; NOTE:  ALL PIECES EXCEPT THE FIRST ARE BLOCK MULTIPLES
  8                 ;
  9                 ;       R4 -> CSR
 10                 ;       R5 -> USER QUEUE ELEMENT
 11                 ;-
 12
 13                         .ENABL  LSB
 14
 15 000374  005765  DLTRAN: TST     Q$WCNT(R5)              ;READ OR WRITE OPERATION?
            000006
 16 000400  100005          BPL     1$                      ;READ...
 17                                                         ; (NOTE: THIS FAILS 2ND TIME THROUGH)
 18 000402  005465          NEG     Q$WCNT(R5)              ;WRITE, MAKE WORD COUNT POSITIVE
            000006
 19 000406  012767          MOV     #FNWRITE!CSIE,DLCODE    ;SET WRITE FUNCTION CODE
            000112
            000542
 20 000414  016502  1$:     MOV     Q$WCNT(R5),R2           ;MAYBE, DETERMINE LENGTH OF
            000006
 21 000420  010203          MOV     R2,R3                   ;TRANSFER IN BLOCKS
 22 000422  062703          ADD     #255.,R3
            000377
 23 000426  105003          CLRB    R3
 24 000430  000303          SWAB    R3
 25 000432                                                  .ASSUME Q$BLKN EQ 0
 26 000432  061503          ADD     @R5,R3                  ;COMPUTE FIRST BLOCK AFTER TRANSFER
 27 000434  026703          CMP     DLUSIZ,R3               ;DOES OPEATION EXTEND INTO REPLACEMENT
            177524
```

Checking if bad-block replacement is needed:

```
                                                      ;BLOCKS ?
   000440  103753          BLO     DLELNK             ;YES, NOT ALLOWED W READ/WRITE
   000442  016700          MOV     DLCC,R0            ;POINT TO REPLACEMENT TABLE - 2
           177506
   000446  005760          TST     4(R0)              ;IS THE FIRST REPLACEMENT BLOCK = 0?
           000004
   000452  001452          BEQ     DLADDR             ;YES, THEN INVALID TABLE (FILES-11)
   000454  005720  2$:     TST     (R0)+              ;SKIP OVER REPLACEMENT BLOCK NUMBER
   000456  012001          MOV     (R0)+,R1           ;GET NEXT BLOCK NUMBER TO REPLACE
   000460  001447          BEQ     DLADDR             ;END OF TABLE, NO REPLACEMENT, DO IO
   000462                                             .ASSUME Q$BLKN EQ 0
   000462  020115          CMP     R1,@R5             ;THIS BAD BLOCK PART OF TRANSFER?
   000464  103773          BLO     2$                 ;NOPE, BELOW, IGNORE IT
   000466  020103          CMP     R1,R3              ;BAD BLOCK WITHIN TRANSFER?
   000470  103043          BHIS    DLADDR             ;NOPE, BEYOND, WHOLE TRANSFER GOOD
   000472  011001          MOV     @R0,R1             ;YES, PICK UP REPLACEMENT BLOCK NUMBER
   000474  014000          MOV     -(R0),R0           ;GET BAD BLOCK NUMBER
   000476                                             .ASSUME Q$BLKN EQ 0
   000476  161500          SUB     @R5,R0             ;COMPUTE DISTANCE OF BAD BLOCK
                                                      ; INTO TRANSFER
   000500  001004          BNE     3$                 ;NOT THE FIRST BLOCK,
                                                      ; GO DO GOOD FIRST PART
```

The replacement table is being used. Pseudoqueue elements are built to break-up the transfer.

```
                   ; FIRST BLOCK OF TRANSFER IS BAD
                   ; FILL IN PSEUDO QUEUE TO TRANSFER THE REPLACEMENT

   000502  005200          INC     R0                 ;SET BLOCK COUNT TO BE 1 BLOCK
   000504  000302          SWAB    R2                 ;IS THE REAL COUNT > 1 BLOCK
                                                      ; HI BYTE>0?
   000506  001403          BEQ     5$                 ;COUNT < 256. WORDS, FIX AND USE IT
   000510  000401          BR      4$                 ;COUNT >= 256. WORDS, GO USE 1 BLOCK

                   ; BAD BLOCK IS IN MIDDLE OF TRANSFER
                   ; FILL IN PSEUDO QUEUE FOR A TRANSFER UP TO BUT NOT INCLUDING THE BAD
                   ; BLOCK.

   000512                                             .ASSUME Q$BLKN EQ 0
   000512  011501  3$:     MOV     @R5,R1             ;START BLOCK OF PARTIAL=ORIGINAL BLOCK
   000514  010002  4$:     MOV     R0,R2              ;COPY BLOCK COUNT OF TRANSFER
   000516  000302  5$:     SWAB    R2                 ; MULTIPLY BY 256. TO GET WORD COUNT
   000520  016503          MOV     Q$BUFF(R5),R3      ;GET ORIGINAL BUFFER ADDRESS
           000004
                                                      ; FOR PSEUDO QUEUE
   000524                                             .ASSUME Q$BLKN EQ 0
   000524  060015          ADD     R0,@R5             ;UPDATE BLOCK NUMBER BY PARTIAL
                                                      ; BLOCK COUNT
   000526  160265          SUB     R2,Q$WCNT(R5)      ;FIX WORD COUNT IN USER QUEUE ELEMENT
           000006
   000532  010200          MOV     R2,R0              ;COPY THE WORD COUNT
   000534  006300          ASL     R0                 ;CHANGE WORD COUNT TO BYTE COUNT
   000536  060065          ADD     R0,Q$BUFF(R5)      ;UPDATE USER BUFFER ADDRESS
           000004

                           .IF NE  MMG$T
   000542  016567          MOV     Q$PAR(R5),DLBPAR   ;*C*SET HI ADDR BITS IN PSEUDO QUEUE
           000012
           001326
   000550  016567          MOV     Q$MEM(R5),DLBMEM   ;*C*SET HI ADDR BITS IN PSEUDO QUEUE
           000014
           001322
   000556  103006          BCC     6$                 ;NO OVERFLOW
   000560  062765          ADD     #CSBA16,Q$PAR(R5)  ;OVERFLOW ORIGINAL ADDRESS INTO
           000020
           000012
                                                      ; HIGH BITS
   000566  062765          ADD     #CSBA16,Q$MEM(R5)  ;OVERFLOW ORIGINAL ADDRESS INTO
           000020
           000014
                                                    ; HIGH BITS
   000574          6$:
                           .ENDC ;NE MMG$T

   000574  004767          CALL    DLSQUE             ;FILL IN REST OF PSEUDO QUEUE
```

```
        001232
000600                  .BR     DLADDR                  ;COMPUTE ADDRESS AND DO I/O

                        .DSABL  LSB


                        .SBTTL  COMPUTE DISK ADDRESS AND START TRANSFER

                ;+
                ;       R4 -> CSR
                ;       R5 -> QUEUE ELEMENT (USER OR PSEUDO)
                ;-

                        .ENABL  LSB

000600  010527  DLADDR: MOV     R5,(PC)+                ;SAVE POINTER TO QUEUE ELEMENT
                                                        ; WE ARE USING
000602  000000  DLQPTR:  .WORD  0
000604                                                  .ASSUME Q$BLKN EQ 0
000604  011502          MOV     @R5,R2                  ;GET BLOCK NUMBER
000606  100670          BMI     DLELNK                  ;NO NEGATIVE BLOCK NUMBERS!
000610  012701          MOV     #DLBPT,R1               ;GET NUMBER OF BLOCKS ON ONE TRACK
        000024
000614                                                  .ASSUME DLBPT   EQ      20.
000614  005000          CLR     R0                      ;INITIALIZE I/O DISK ADDRESS TO 0
000616  000410          BR      2$                      ;ENTER DIVIDE LOOP

000620  010203  1$:     MOV     R2,R3                   ;COPY DIVIDEND
000622  042702          BIC     #^C<17>,R2              ;COMPUTE DIV = 16Q + R
        177760
000626  040203          BIC     R2,R3                   ; AND GET 16Q TO WORK WITH
000630  060300          ADD     R3,R0                   ;RESULT <- RESULT + IOHS/4
000632                                                  .ASSUME IOHS/2/2 EQ     16.
000632  006203          ASR     R3                      ;COMPUTE 8Q
000634  006203          ASR     R3                      ; THEN 4Q
000636  160302          SUB     R3,R2                   ;NEW DIVIDEND = R - 4Q
000640  020201  2$:     CMP     R2,R1                   ;DONE? (NUMBER NOW < DLBPT)
000642  103366          BHIS    1$                      ;NOPE...
000644  006300          ASL     R0                      ;YES, QUOTIENT*IOHS/4 => QUO*IOHS/2
000646  050200          BIS     R2,R0                   ;MERGE BLOCK NUMBER WITH TRACK
000650  006300          ASL     R0                      ;*2 FOR TWO 128. WORD SECTORS/BLOCK
000652  103646          BCS     DLELNK                  ;OVERFLOW MEANS BEYOND END OF DEVICE
000654  100004          BPL     3$                      ;POSITIVE IS OK FOR EITHER RL01/02
000656  026727          CMP     DLUSIZ,#DLSIZ2          ;NEGATIVE IS OK FOR RL02 ONLY
        177302
        047742
000664  001241          BNE     DLELNK                  ; BUT NOT OK FOR RLO1
000666  010067  3$:     MOV     R0,DLDA                 ;SAVE STARTING DISK ADDRESS
        000256
000672  160201          SUB     R2,R1                   ;CALCULATE BLOCKS LEFT ON TRACK
000674  000301          SWAB    R1                      ;CONVERT TO WORDS LEFT ON TRACK
000676  010167          MOV     R1,DLWTRK               ;SAVE THAT NUMBER
        000224

                        .IF NE  ERL$G
                        MOV     Q$WCNT(R5),DLWC         ;SET WORD COUNT FOR EL
                        .ENDC ;NE ERL$G

000702  012727          MOV     #1,(PC)+                ;CLEAR RETRY COUNT
        000001
                                                        ; (THESE ARE FATAL ERRORS)
000706  000000  DLRTY:   .WORD  0
000710  004767          CALL    DLRST                   ;RESET DRIVE
```

```
        001062
000714  004767          CALL    DLGST                   ;AND GET STATUS
        001020
000720  100434          BMI     DLERJM                  ;ERROR HERE IS FATAL
000722  006200          ASR     R0                      ;IS THE DRIVE READY?
000724                                                  .ASSUME CSDRDY  EQ      1
000724  103032          BCC     DLERJM                  ;NO, FATAL UNRETRYABLE ERROR
000726  042701          BIC     #STWL!STHS!STDT,R1      ;IGNORE WRITE LOCK, HEAD SELECT,
        020300
                                                        ; DRIVE TYPE
000732  022701          CMP     #STHO!STBH!STSLM,R1     ;HEADS, BRUSHES AND STATE OK?
        000035
000736  001025          BNE     DLERJM                  ;NO, FATAL ERROR

        000742' DRETRY  = .+2
000740                                                  .ASSUME DRETRY-DLSTRT LT 1000
000740  012767          MOV     #DLRCNT,DLRTY           ;SET REAL RETRY COUNT
        000010
        177740
000746                  .BR     DLTRAK                  ;GET ON TRACK

                        .DSABL  LSB

                        .SBTTL  ENSURE THAT DISK IS ON TRACK BEFORE TRANSFER

                ;+
                ; CALCULATE THE DIFFERENCE WORD FOR THE SEEK.
                ; TRY 16 TIMES TO READ A HEADER.
                ; IF ALL FAIL, LOG AN ERROR AND ISSUE A REVERSE SEEK (SEEK -1 TRACK)
                ;  AND A READ HEADER TO CAUSE AN INTERRUPT.
                ;
                ;       R4 -> CSR
                ;       R5 -> QUEUE ELEMENT
                ;-

                        .ENABL  LSB

000746  005027  DLTRAK: CLR     (PC)+                   ;RESET REVERSE SEEK FLAG
000750  000000  DLREV:   .WORD  0
000752  017701          MOV     @DLCC,R1                ;GET CURRENT CYLINDER
        177176
000756  022701          CMP     #-1,R1                  ;IS IT VALID?
        177777
000762  001015          BNE     2$                      ;YES, USE IT TO START WITH
                ;***ACTION*** OLD CODE HAS ANOTHER RETRY VALUE
000764  016702          MOV     DRETRY,R2               ;SET READ HEADER RETRY COUNT
        177752
000770  006302          ASL     R2                      ; (DLRCNT*2)
000772  012701  1$:     MOV     #FNRDH,R1               ;SET CODE FOR READ HEADERS FUNCTION
        000010
000776  004767          CALL    DLXCT                   ;EXECUTE THE FUNCTION
        001006
001002  100005          BPL     2$                      ;FUNCTION EXECUTED OK
001004  077206          SOB     R2,1$                   ; any retries left?
001006  105267          INCB    DLREV                   ;SET REVERSE SEEK FLAG
        177736
001012  000167  DLERJM: JMP     DLERRH             ;RETRY OPERATION
        000452

001016  016700  2$:     MOV     DLDA,R0                 ;RETRIEVE STARTING DISK ADDRESS
        000126
001022  012702          MOV     #IOSA,R2                ;MASK OUT
        000077
001026  040200          BIC     R2,R0                   ;SECTOR BITS FROM DESIRED ADDRESS
001030  040201          BIC     R2,R1                   ; AND FROM CURRENT ADDRESS
001032  020001          CMP     R0,R1                   ;DO WE NEED TO DO A SEEK?
001034  001427          BEQ     DLXFER                  ;NOPE, ALREADY ON CYLINDER AND HEAD
001036  010003          MOV     R0,R3                   ;YES, SAVE DESIRED CYLINDER AND HEAD
001040  005202          INC     R2                      ;GET MASK FOR HEAD SELECT
001042                                                  .ASSUME IOHS    EQ      IOSA+1
001042  040200          BIC     R2,R0                   ;STRIP HEAD SELECT BIT FROM
                                                        ; DESIRED ADDRESS
001044  040201          BIC     R2,R1                   ; AND FROM CURRENT ADDRESS
001046  160001          SUB     R0,R1                   ;COMPUTE DISTANCE FROM DESIRED
                                                        ; TO ACTUAL CYLINDER
001050                                                  .ASSUME SKCADF  EQ      IOCA
001050  103003          BHIS    3$                      ;DESIRED <= ACTUAL, MOVE TOWARD EDGE
```

```
001052  005401          NEG     R1                     ;DESIRED > ACTUAL, MOVE TOWARD SPINDLE
001054  052701          BIS     #SKDIR,R1              ; (SET DIRECTION BIT)
        000004

001060  005201  3$:     INC     R1                     ;SET MARKER BIT
001062                                              .ASSUME    SKMARK  EQ      1
001062  030203          BIT     R2,R3                  ;DO WE WANT TO USE SURFACE 1?
001064  001402          BEQ     4$                     ;NO
001066  052701          BIS     #SKHS,R1               ;YES, SET SURFACE 1 BIT
        000020
001072  012777  4$:     MOV     #-1,@DLCC              ;VOID KNOWLEDGE OF CURRENT CYLINDER
        177777
        177054
001100  004767          CALL    DLSEEK                 ;EXECUTE THE SEEK
        000622
001104  100571          BMI     DLERRH                 ;OOPS, ERROR EXECUTING SEEK
001106  016777          MOV     DLDA,@DLCC             ;SET CURRENT CYLINDER
        000036
        177040
001114                  .BR     DLXFER                 ;NOW DO THE TRANSFER

                        .DSABL  LSB


                        .SBTTL  DLXFER - START AN I/O TRANSFER

                ;+
                ;       R4 -> CSR
                ;       R5 -> QUEUE ELEMENT
                ;-

                        .ENABL  LSB
001114          DLXFER:

001114  062704          ADD     #RLMP,R4               ;POINT TO RLMP IN CONTROLLER
        000006

001120  062705          ADD     #Q$WCNT,R5             ;POINT TO WORD COUNT IN QUEUE ELEMENT
        000006
001124  012703          MOV     (PC)+,R3               ;GET NUMBER OF WORDS LEFT ON TRACK
001126  000000  DLWTRK:  .WORD  0
001130  020315          CMP     R3,@R5                 ;COMPARE AGAINST TOTAL TRANSFER
001132  101401          BLOS    1$                     ;<=, USE REMAINDER OF TRACK
001134  011503          MOV     @R5,R3                 ;>, USE TOTAL TRANSFER COUNT
001136  010327  1$:     MOV     R3,(PC)+               ;SAVE TRANSFER COUNT FOR LATER
001140  000000  DLWC:    .WORD  0                      ; : TRANSFER COUNT
001142  005403          NEG     R3                     ;MUST BE 2'S COMPLEMENT

001144  010314          MOV     R3,@R4                 ;LOAD WORD COUNT INTO CONTROLLER
001146  012744          MOV     (PC)+,-(R4)            ;LOAD STARTING DISK ADDRESS
001150  000000  DLDA:    .WORD  0
001152  014544          MOV     -(R5),-(R4)            ;SET BUS ADDRESS
                        .
                        .
                        .
001154  012700          MOV     (PC)+,R0               ;GET FUNCTION CODE
001156  000000  DLCODE:  .WORD  0                      ;READ OR WRITE CODE
001160  052700          BIS     (PC)+,R0               ;ADD IN UNIT SELECT BITS
001162  000000  DLUNIT:  .WORD  0                      ;UNIT NUMBER IN BITS 8-9

                        .IF NE  MMG$T
001164  000416  $RLV1A: BR      10$                    ;IF NO RLV12...
                                                       ; (CHANGED TO 'NOP' IF USING RLV12)
001166  016546          MOV     Q$PAR-Q$BUFF(R5),-(SP) ;SAVE Q22 HIGH-ORDER BITS
        000006
001172  006216          ASR     (SP)                   ;SHIFT THEM TO THEIR CORRECT POSITIONS
001174  006216          ASR     (SP)
001176  006216          ASR     (SP)
001200  006216          ASR     (SP)
001202  012664          MOV     (SP)+,RLBAE-RLBA(R4)   ;SET THE HIGH-ORDER BITS
        000006
001206  016546          MOV     Q$PAR-Q$BUFF(R5),-(SP) ;SAVE HIGH-ORDER BUS ADDRESS
        000006
001212  042716          BIC     #<^C60>,(SP)           ;STRIP TO HIGH-ORDER BITS<17:16>
        177717
001216  052600          BIS     (SP)+,R0               ; AND MERGE WITH COMMAND WORD
```

```
001220  000410           BR      30$
 
001222  032765  10$:     BIT     #1700,Q$PAR-Q$BUFF(R5)  ;22-BIT ADDRESS SPECIFIED?
        001700
        000006
001230  001402           BEQ     20$                     ;NOPE, THEN ADDRESS IS OKAY TO USE
001232  000167           JMP     DLEROR                  ;YES, CAN'T BE USED ON NON RLV12
        000350

001236  056500  20$:     BIS     Q$PAR-Q$BUFF(R5),R0     ;MERGE EXTENDED ADDRESS BITS INTO
        000006
                                                         ; COMMAND WORD
001242          30$:
                         .ENDC ;NE MMG$T


001242  010044           MOV     R0,-(R4)                ;LOAD FUNCTION AND GO

001244  000207           RETURN                          ;WAIT FOR AN INTERRUPT


                         .DSABL  LSB
```

Interrupt Service Section

```
                         .SBTTL  DLINT - INTERRUPT ENTRY POINT

                ; INTERRUPTS ENTER THE HANDLER HERE

                         .ENABL  LSB
```

The .DRAST macro:

When a function is completed, the device interrupts, and the handler is entered here to dismiss the interrupt and the queue element.

```

001246                   .DRAST  DL,5
                         .
                         .
                         .
```

Drop to fork level rather than device priority because the routine is lengthy and it needs all the registers.

```
001256                   .FORK   DLFBLK                  ;GO TO FORK LEVEL
```

Load the registers.

```
001264  016704           MOV     DLCSR,R4                ;POINT TO CSR ADDRESS
        176542
001270  016705           MOV     DLQPTR,R5               ;POINT TO QUEUE ELEMENT
        177306
001274  105767           TSTB    DLREV                   ;REVERSE SEEK IN PROGRESS?
        177450
001300  001222           BNE     DLTRAK                  ;YES, GO RETRY THE REAL TRANSFER

001302  005714           TST     @R4                     ;CHECK RLCS

001304  100471           BMI     DLERRH                  ;IF ERROR, GO DIAGNOSE IT
001306                                                   .ASSUME CSERR   EQ      100000
001306  016703           MOV     DLWC,R3                 ;GET WORD COUNT OF THIS TRANSFER
        177626
001312  160365           SUB     R3,Q$WCNT(R5)           ;CALCULATE WORDS REMAINING TO TRANSFER
        000006
001316  001036           BNE     2$                      ;MORE TO DO, USE NEXT TRACK
001320  026727           CMP     DLCODE,#FNWRITE!CSIE    ;WAS THE LAST FUNCTION A WRITE?
        177632
        000112
001326  001030           BNE     11$                     ;NO, DONE WITH THIS (PARTIAL) ELEMENT

001330  032764           BIT     #1,RLDA(R4)             ;GOT A SECTOR TO WRITE YET?
        000001
        000004
```

```
   001336  001424          BEQ     11$                      ;NO, DON'T ZERO FILL
   001340  005265          INC     Q$WCNT(R5)               ;SET WORD COUNT TO 1
           000006
                                                            ; (CONTROLLER FILLS 127.)
                   .IF EQ MMG$T
                           .ADDR   #DLFILL,-(SP)            ;GET THE BUFFER ADDRESS
                   .IFF ;EQ MMG$T
   001344  016765          MOV     BUFADH,Q$PAR(R5)         ;GET HI ADDR OF UMR
           000716
           000012
   001352  016746     MOV  BUFADL,-(SP)            ;GET LO ADDR OF UMR

           000712
   001356  062716          ADD     #<BUFEND-DLBBUF>,@SP     ;POINT TO DLFILL
           000130
   001362  103003          BCC     100$                     ;IF NO OVERFLOW, BRANCH
   001364  062765          ADD     #CSBA16,Q$PAR(R5)        ;UPDATE HI ORDER ADDRESS BITS
           000020
           000012
   001372          100$:
                   .ENDC ;EQ MMG$T
   001372  012665          MOV     (SP)+,Q$BUFF(R5)         ;SET THE BUFFER ADDRESS
           000004

   001376  016467          MOV     RLDA(R4),DLDA            ; AND THE DISK ADDRESS
           000004
           177544


   001404  000167  1$:     JMP     DLTRAK                   ;GO DO IT (DLWTRK > 1 = Q$WCNT)
           177336

   001410  000167  11$:    JMP     DLEXFR                   ;GO FINISH TRANSFER
           000204

   001414  006303  2$:     ASL     R3                       ;CHANGE WORD COUNT TO BYTE COUNT
   001416  060365          ADD     R3,Q$BUFF(R5)            ;UPDATE USER BUFFER ADDRESS
           000004

                           .IF NE  MMG$T
   001422  103003          BCC     3$                       ;NO OVERFLOW
   001424  062765          ADD     #CSBA16,Q$PAR(R5)        ;UPDATE HIGH ORDER ADDRESS BITS
           000020
           000012
   001432          3$:
                           .ENDC ;EQ MMG$T

   001432  052767          BIS     #77,DLDA                 ;UPDATE SURFACE/CYLINDER ADDRESS
           000077
           177510
   001440  005267          INC     DLDA                     ; TO FIRST SECTOR, NEXT HEAD/CYLINDER
           177504
   001444  001460          BEQ     DLEROR                   ;OVERFLOWED DEVICE !!!
   001446  100004          BPL     301$                     ;OK FOR EITHER RL01/02
   001450  026727          CMP     DLUSIZ,#DLSIZ2           ;MINUS OK ONLY FOR RL02
           176510
           047742
   001456  001053          BNE     DLEROR                   ; VERY BAD IF RL01 !!!
   001460  012767  301$:   MOV     #DLWPT,DLWTRK            ;SAVE NUMBER OF WORDS ON A WHOLE TRACK
           012000
           177440
   001466  000746  4$:     BR      1$                       ;GO CONTINUE TRANSFER ON NEXT TRACK


                           .SBTTL  HANDLE THE ERRORS

   001470          DLERRH:
                           .IF EQ  ERL$G

   001470  011403          MOV     @R4,R3                   ;GET RLCS CONTENTS WITH ERROR BITS
```

```
                                .IFF
                                MOV     R4,R1                   ;GET CSR ADDRESS
                                .ADDR   #DLRBLK,R2              ; CALCULATE ADDRESS OF REGISTER BUFFER
                                MOV     R2,R3                   ;SAVE BUFFER ADDRESS
                                MOV     (R1)+,(R3)+             ;TRANSFER RLCS
                                MOV     (R1)+,(R3)+             ;TRANSFER RLBA
                                MOV     (R1)+,(R3)+             ;TRANSFER RLDA
                                MOV     (R1)+,(R3)+             ;TRANSFER RLMP
                                CALL    DLGST                   ;GET THE DRIVE STATUS INFO
                                MOV     R1,(R3)+                ; AND SAVE IT FOR ERROR LOGGER
                                COM     R1                      ;COMPLEMENT
                                BIT     #STWL,R1                ;Write lock error?
                                BEQ     5$                      ;Yes, don't log it
                                                                ; (reversed logic due to COM above)
                                MOV     DLDA,(R3)+              ;SAVE THE DISK ADDRESS THAT WE USED
                        $RLV1B: BR      10$                     ;IF NO RLV12...
                                                                ; (CHANGED TO 'NOP' IF USING RLV12)
                                MOV     RLBAE(R4),(R3)+         ;TRANSFER RLBAE

                        10$:    MOV     DRETRY,R3
                                SWAB    R3
                                ADD     #DLREG,R3               ;R3= MAX RETRIES/ NUMBER OF REGISTERS

                        $RLV1C: BR      20$                     ;IF NO RLV12...
                                                                ; (CHANGED TO 'NOP' IF USING RLV12)
                                INC     R3                      ;BUMP FOR EXTRA REGISTER ON RLV12

                        20$:    JSR     R4,FIXWC                ;GET Q$WCNT SET RIGHT, PUSH OLD VALUE
                                MOV     DLRTY,R4                ;GET NUMBER OF RETRIES LEFT
                        ADD     #DL$COD*400-1,R4        ;SET DEVICE ID FLAG, COUNT=COUNT-1
                                                                ; (report retries remaining, not
                                                                ;  current retry number)
                                CALL    @$ELPTR                 ;LOG THE ERROR
                                MOV     (SP)+,Q$WCNT(R5)        ;RESET WORD COUNT
                        5$:     MOV     DLCSR,R4                ;POINT TO CSR AGAIN
                                MOV     DLRBLK,R3               ;GET RLCS AT TIME OF FAILURE
                                .ENDC ;EQ ERL$G

   001472  012777               MOV     #-1,@DLCC               ;INVALIDATE CURRENT CYLINDER
           177777
           176454
                                                                ; (FORCE READ HEADER)
   001500  004767               CALL    DLRST                   ;RESET DRIVE
           000272
   001504  105767               TSTB    DLREV                   ;REVERSE SEEK REQUIRED?
           177240
   001510  001415               BEQ     6$                      ;NO, GO SEE IF WE CAN RETRY
   001512  105267       51$:    INCB    DLREV                   ;SET REVERSE SEEK FLAG IF RETRY

           177232
                                                                ; FROM DRIVE N;002
   001516  012701               MOV     #177600!SKMARK,R1       ;Reverse seek to cylinder zero
           177601
   001522  004767               CALL    DLSEEK                  ;EXECUTE THE SEEK
           000200
   001526  100427               BMI     DLEROR                  ;SEEK FAILED, CALL IT FATAL
   001530  016700               MOV     DLUNIT,R0               ;GET UNIT NUMBER TO USE
           177426
   001534  052700               BIS     #CSIE!FNRDH,R0          ;ADD CODE FOR READ HEADER
           000110

   001540  010014               MOV     R0,@R4                  ;LOAD FUNCTION AND GO

   001542  000207               RETURN                          ;WAIT FOR THE INTERRUPT

   001544  106203       6$:     ASRB    R3                      ;AT TIME OF FAILURE, WAS DRIVE READY?
   001546  103361               BCC     51$                     ;NO, REVERSE SEEK UNTIL IT IS
   001550                                                       .ASSUME CSDRDY  EQ      1
   001550  006303               ASL     R3                      ;SHIFT TO GET DRIVE ERROR BIT IN CARRY
   001552  006303               ASL     R3                      ; AND NXM BIT IN SIGN
   001554  100414               BMI     DLEROR                  ;FATAL IF NON-EXISTENT MEMORY
   001556                                                       .ASSUME CSNXM   EQ      020000
   001556  103010               BCC     7$                      ;GO RETRY IF NOT DRIVE ERROR
   001560                                                       .ASSUME CSDE    EQ      040000
   001560  004767               CALL    DLGST                   ;DRIVE ERROR, GO GET DRIVE STATUS
           000154
   001564  032701               BIT     #STWGE,R1               ;WRITE GATE ERROR?
           002000
   001570  001406               BEQ     DLEROR                  ;FATAL IF NOT
```

```
001572  032701          BIT     #STWL,R1                ;YES, WRITE GATE WITH WRITE LOCK?
        020000
001576  001003          BNE     DLEROR                  ;YES, FATAL
001600  005367  7$:     DEC     DLRTY           ;ANY RETRIES LEFT?
        177102
001604  003330          BGT     4$                      ;YES, GO DO ONE
001606  016705  DLEROR: MOV     DLCQE,R5                ;GET QUEUE ELEMENT POINTER
        176176
001612                                                  .ASSUME Q$BLKN-2 EQ Q$CSW
001612  052755          BIS     #HDERR$,@-(R5)          ;FLAG CHANNEL ERROR
        000001
001616  000416          BR      DLQCOM                  ;FINISH-UP

                        .DSABL  LSB

                        .SBTTL  FINISH SUCCESSFUL OPERATION

                        .ENABL  LSB

001620  016705  DLEXFR: MOV     DLCQE,R5                ;GET ORIGINAL QUEUE ELEMENT POINTER
        176164
001624  020567          CMP     R5,DLQPTR               ;PSEUDO QUEUE IN USE?
        176752
```

Test if we're dismissing a queue element for a replacement table reread or if we're doing a partial transfer using replacement. If a partial transfer, go back and get the rest before we dismiss the original queue element.

```
001630  001411          BEQ     1$                      ;NO, THIS IS THE END OF THE REQUEST
001632  126527          CMPB    Q$FUNC(R5),#FN$REP      ;WAS FUNCTION A FORCE TABLE RE-READ?
        000002
        000374
001640  001405          BEQ     1$                      ;YES, WE ARE NOW DONE
001642  005765          TST     Q$WCNT(R5)              ;IS THERE ANYTHING LEFT TO TRANSFER?
        000006
001646  001402          BEQ     1$                      ;NOPE, ALL DONE
001650  000167          JMP     DLTRAN                  ;GO DO NEXT PART OF BROKEN TRANSFER
        176520

001654          1$:
                        .IF NE  ERL$G
                        JSR     R4,FIXWC                ;FIX WORD COUNT FOR READ/WRITE
                        TST     (SP)+                   ;DUMP STACKED OLD VALUE
                        TST     SCSFLG                  ;LOGGING SUCCESSES?
                        BNE     DLQCOM                  ;NOPE...
                        MOV     #DL$COD*400+377,R4      ;FLAG SUCCESS FOR EL
                        CALL    @$ELPTR                 ;CALL THE ERROR LOG HANDLER
                        .ENDC ;NE ERL$G

```

I/O Completion Section

Dismiss the queue element.

```
001654          DLQCOM: .DRFIN  DL                      ;COMPLETE I/O OPERATION

                        .DSABL  LSB

                        .SBTTL  GET DEVICE SIZE

                ; SPECIAL FUNCTION TO GET VOLUME SIZE:
                ; READ THE DRIVE TYPE BIT FOR THE SELECTED DRIVE. THEN RETURN THE
                ; DRIVE'S SIZE, IN BLOCKS, IN THE FIRST WORD OF THE USER'S BUFFER.

001672          DLGSIZ:
                        .IF EQ  MMG$T
                        MOV     DLUSIZ,@Q$BUFF(R5)      ;PUT SIZE IN BUFFER
                        .IFF
001672  016746          MOV     DLUSIZ,-(SP)            ;SET SIZE ON STACK
        176266
001676  010504          MOV     R5,R4                   ;COPY QUEUE POINTER FOR PUTWORD
001700  004777          CALL    @$PTWRD                 ;PUT SIZE IN BUFFER
        000404
```

```
14                               .ENDC ;EQ MMG$T
15 001704  005700                TST     R0                      ;Was there an error (no drive?)
16                                                               ;R0 should be CSR from DLGST
17 001706                                                        .Assume CSERR EQ 100000
18 001706  100362                BPL     DLQCOM                  ;Branch if not
19 001710  032700                BIT     #CSERRC,R0              ;Is there an error code?
           036000
20 001714  001334                BNE     DLEROR                  ;Branch if yes

21 001716  032701                BIT     #STVC,R1                ;Is it a volume check error?
           001000
22 001722  001731                BEQ     DLEROR                  ;If not, report hard error
23 001724  000753                BR      DLQCOM

 1                               .SBTTL  DLXCT - FUNCTION EXECUTION ROUTINES
 2
 3                       ;+
 4                       ; EXECUTE A GET DRIVE STATUS OR ANY NON-INTERRUPT FUNCTION
 5                       ;       AND WAIT FOR COMPLETION
 6                       ;
 7                       ; INPUTS:
 8                       ;       R1 =  FUNCTION CODE             IF DLXCT
 9                       ;             SEEK DIFFERENCE WORD      IF DLSEEK
10                       ;       R4 -> CSR
11                       ;
12                       ; OUTPUTS:
13                       ;       FUNCTION EXECUTED
14                       ;
15                       ;       R0 = CSR CONTENTS
16                       ;       R1 = MP CONTENTS
17                       ;       N = 1 IF ERROR
18                       ;-
19
20                               .ENABL  LSB
21
22 001726                DLSEEK:

24 001726  010164                MOV     R1,RLDA(R4)             ;LOAD DIFFERENCE WORD IN CONTROLLER
           000004

28 001732  012701                MOV     #FNSEEK,R1              ;ISSUE SEEK COMMAND
           000006
29 001736  000424                BR      DLXCT
30
31 001740                DLGST:

33 001740  012764                MOV     #GSGS!GSMARK,RLDA(R4)   ;TELL DRIVE TO GET STATUS
           000003
           000004

37 001746  004767                CALL    1$                      ;EXECUTE THE GET STATUS
           000032
38 001752  100026                BPL     4$                      ;NO ERROR SO EXIT

40 001754  005764                TST     RLBA(R4)                ;ERROR -- IS IT AFTER BUS INIT?
           000002

45 001760  001023                BNE     4$                      ;NO -- LOG THE ERROR
46 001762  004767                CALL    DLRST                   ;YES -- DO A RESET
           000010

48 001766  012764                MOV     #GSGS!GSMARK,RLDA(R4)   ;AND TRY THE GET STATUS AGAIN
           000003
           000004

52 001774  000403                BR      1$                      ;BUT ONLY TRY IT ONCE!
53
54 001776                DLRST:

56 001776  012764                MOV     #GSRST!GSGS!GSMARK,RLDA(R4) ;GET DRIVE RESET COMMAND
           000013
           000004
60 002004  012701        1$:     MOV     #FNGSTS,R1              ;GET 'GET STATUS' FUNCTION CODE
           000004
61 002010  056701        DLXCT:  BIS     DLUNIT,R1               ;ADD IN UNIT SELECT BITS
           177146

63 002014  010114                MOV     R1,@R4                  ;GIVE IT TO DRIVER
64 002016  105714        2$:     TSTB    @R4                     ;WAIT FOR FUNCTION TO BE ACCEPTED
```

```
002020  100376          BPL     2$
002022          3$:

002022  016401          MOV     RLMP(R4),R1             ;GET RETURNED STATUS WORD
        000006
002026  011400          MOV     @R4,R0                  ; AND CSR VALUE (SET N-BIT IF ERROR)
                        .
                        .
                        .
002030  000207  4$:     RETURN

                        .DSABL  LSB

                        .
                        .
                        .
                        .DSABL  LSB

```

DLSQUE is used to read the bad-block replacement table into memory and to break up a transfer that uses the table.

```
                        .SBTTL  DLSQUE - SETUP PSEUDO QUEUE ELEMENT

                ;+
                ; SET UP THE PSEUDO QUEUE FOR BAD BLOCK TABLE READS OR PARTIAL TRANSFERS
                ;
                ; INPUTS:
                ;       R1 =  STARTING BLOCK NUMBER OF PARTIAL TRANSFER
                ;       R2 =  WORD COUNT
                ;       R3 -> BUFFER
                ;       R5 -> USER QUEUE ELEMENT
                ;
                ; OUTPUTS:
                ;       R0 =  RANDOM
                ;       R5 -> PSEUDO QUEUE ELEMENT
                ;-

002032          DLSQUE:
002032                  .ADDR   #DLBWCT,R0              ; POINT TO PSEUDO QUEUE ELEMENT
002040  010210          MOV     R2,@R0                  ;STORE WORD COUNT
002042  010340          MOV     R3,-(R0)                ;STORE BUFFER ADDRESS
002044  016540          MOV     Q$FUNC(R5),-(R0)        ;COPY UNIT NUMBER AND
        000002
                                                        ; SPECIAL FUNCTION BYTE
002050  010140          MOV     R1,-(R0)                ;STORE BLOCK NUMBER
002052                                                  .ASSUME Q$BLKN-2 EQ Q$CSW
002052  014560          MOV     -(R5),-2(R0)            ;STORE POINTER TO CSW
        177776
002056  010005          MOV     R0,R5                   ;POINT R5 AT PSEUDO QUEUE
002060  000207          RETURN

                        .IF NE  ERL$G
```

```
                        .SBTTL  FIXWC - FIX WORD COUNT FOR LOGGER

                ;+
                ; FIX WORD COUNT IN QUEUE ELEMENT FOR ERROR LOGGER
                ;
                ; INPUTS:
                ;       R5 -> QUEUE ELEMENT
                ;       DLWC = WORD COUNT USED FOR I/O
                ;
                ; OUTPUTS:
                ;       R4 =  RANDOM
                ;       @SP = OLD VALUE OF Q$WCNT TO RESTORE
                ;       Q$WCNT(R5) = DLWC (NEGATED IF WRITE)
                ;-

                FIXWC:  MOV     Q$WCNT(R5),@SP          ;SAVE OLD COUNT ON STACK
                        MOV     DLWC,Q$WCNT(R5)         ;SET THE CORRECT VALUE
                        CMP     DLCODE,#FNWRITE!CSIE    ;WAS IT A WRITE?
                        BNE     1$                      ;NO
                        NEG     Q$WCNT(R5)              ;YES, FIX ELEMENT VALUE
                1$:     JMP     @R4                     ;RETURN

                        .ENDC ;NE ERL$G
```

```
                         .SBTTL  DATA AREAS

                 ; PSEUDO QUEUE ELEMENT

002062  177777           .WORD   -1                      ;ADDRESS OF CSW
002064  177777           .WORD   -1                      ;BLOCK NUMBER
002066     000           .BYTE   0                       ;SPECIAL FUNCTION BYTE
002067     377           .BYTE   -1                      ;UNIT NUMBER
002070  177777  DLBADD: .WORD   -1                      ;BUFFER ADDRESS
002072  177777  DLBWCT: .WORD   -1                      ;WORD COUNT

                         .IF NE  MMG$T
002074  000000           .WORD   0                       ;COMPLETION ADDRESS
002076  177777  DLBPAR: .WORD   -1                      ;PAR VALUE
002100  177777  DLBMEM: .WORD   -1                      ;MEM VALUE
002102  000000           .WORD   0                       ;(RESERVED)
                         .ENDC ;NE MMG$T


                 ; BAD BLOCK REPLACEMENT TABLE BUFFER AND CURRENT CYLINDER WORD
                 ;
                 ;       CONSISTS OF ONE WORD AND ONE TABLE FOR EACH UNIT.
                 ;       EACH TABLE CONSISTS OF TWO WORD ENTRIES. WORD 1
                 ;       IS BAD BLOCK AND WORD 2 IS IT'S REPLACEMENT. A
                 ;       TABLE IS ENDED BY A ZERO ENTRY.
                 ;
                 ;       THIS TABLE WILL BE MAPPED INTO HIGH MEMORY WITH UB SUPPORT
                 ;
```

This is the bad-block replacement table:

```
002104          DLBBUF:
        000002           .REPT   DL$UN                   ;ONE TABLE PER UNIT
                         .WORD   -1                      ;CURRENT CYLINDER NUMBER (-1=UNKNOWN)
                         .WORD   -1                      ;INDICATES TABLE NOT READ YET
                         .BLKB   DLTSIZ-2                ;THE TABLE
                         .ENDR
002234          BUFEND:

                 ; DLFILL ALSO USES THE PERMANENT UMR

002234  000000  DLFILL:  .WORD  0                       ;MUST BE 0 TO ZERO-FILL BUFFER

002236  000000  DLFBLK: .WORD   0,0,0,0                 ;FORK QUEUE BLOCK
002240  000000
002242  000000
002244  000000

                         .IF NE  ERL$G
                DLRBLK: .BLKW   DLREG+1                 ;DL STATUS REGISTERS FOR CALL
                                                         ; TO ERROR LOGGER (+1 FOR RLBAE)
                         .ENDC ;NE ERL$G

                 .IF NE MMG$T

                 ;+
                 ;       DL INTERNAL VARIABLE DEFINITIONS.
                 ;-

002246  000000  $ENTPT: .WORD   0       ; POINTER TO $ENTRY TABLE
002250  000000  $PNMPT: .WORD   0       ; POINTER TO $PNAME TABLE
002252  000000  H2UB:   .WORD   0       ; POINTER TO UBVECT
002254  000000  DLSLOT: .WORD   0       ; DL'S OFFSET IN DEVICE TABLES
002256  000000  DLENT:  .WORD   0       ; DL'S $ENTRY TABLE ENTRY POINTER
002260  000000  DLPNA:  .WORD   0       ; DL'S $PNAME TABLE ENTRY POINTER
002262  000000  DLILQE: .WORD   0       ; DL INTERNAL QUEUE LAST QEL POINTER
002264  000000  DLICQE: .WORD   0       ; DL INTERNAL QUEUE FIRST QEL POINTER

                 ;+
                 ;  DEFINITION OF THE HANDLER INTERNAL BUFFER AND THE WORDS THAT ARE
                 ;  USED TO PROGRAM DMA DEVICES THAT TRANSFER DATA TO AND FROM IT.
                 ;-

002266  000000  BUFADH: .WORD   0       ; BITS 0-15 OF UNIBUS VIRTUAL POINTER TO DLBBUF
002270  000000  BUFADL: .WORD   0       ; BITS 16-21 OF UNIBUS VIRTUAL POINTER TO DLBBUF
```

```
                 ; TABLE OF STANDARD DMA SPFUNS THAT DO NOT HAVE A PERMANENT UMR
                 ; ALLOCATED TO THEM

   002272        UBTAB:  .DRSPF  -,<FN$WRT>      ;ABSOLUTE WRITE, NO BAD BLOCK
   002274                .DRSPF  -,<FN$RED>      ;ABSOLUTE READ (REPLACEMENT)
   002276  000000        .WORD   0               ;TABLE TERMINATOR

                 .ENDC ;NE MMG$T
```

Bootstrap Driver

```
                         .SBTTL  BOOTSTRAP DRIVER

```

The .DRBOT macro:

```
   002300                .DRBOT  DL,BOOT1,B.READ
           177777  ...V7=-1
                   .IIF IDN NO,YES,...V7=0
```

Termination Section

The .DREND macro generated by .DRBOT (the macro expansion):

```
   001770                .DREND  DL,0,
                 .IF B <>
   001770        .PSECT  DLDVR
                 .IFF
                 .PSECT
                 .ENDC
                 .IIF NDF DL$END,DL$END::
                 .IF EQ  .-DL$END
                 .IF NE MMG$T!<0&2.>
                 $RLPTR::.WORD   0
                 $MPPTR::.WORD   0
                 $GTBYT::.WORD   0
                 $PTBYT::.WORD   0
                 $PTWRD::.WORD   0
                 .ENDC
                 .IF NE ERL$G!<0&1>
                 $ELPTR::.WORD   0
                 .ENDC
                 .IF NE TIM$IT!<0&4.>
                 $TIMIT::.WORD   0
                 .ENDC
   001770  000000  $INPTR::.WORD   0
   001772  000000  $FKPTR::.WORD   0
                 .IIF NDF ...V22 ...V22=0
                 .IF NE  ...V22&^o40000
                 DL$X64 =:.
                 .REPT   16.
                         .WORD   0
                 .ENDR
                 .ENDC
                 .GLOBL  DLSTRT
```

The following line marks the end of the loadable portion of the handler. It is used to determine the handler's length in memory.

```
           001774' DLEND==.
                   .IFF
                   .PSECT  DLBOOT
                   .IIF LT <DLBOOT-.+^o664>,.ERROR;?SYSMAC-E-Primary boot too large;
                   .=DLBOOT+^o664
                   BIOERR: JSR     R1,REPORT
                            .WORD  IOERR-DLBOOT
                   REPORT: MOV     #BOOTF-DLBOOT,R0
                           MOV     #30002$-DLBOOT,R2
                           CALL    @R2
                           MOV     @R1,R0
                           CALL    @R2
                           MOV     #CRLFLF-DLBOOT,R0
                           CALL    @R2
                   30001$: HALT
```

```
                        BR      30001$
                30002$: TSTB    @#TPS
                        BPL     30002$
                        MOVB    (R0)+,@#TPB
                        BNE     30002$
                        RETURN
                BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"
                IOERR:  .ASCII  "I/O error"
                CRLFLF: .ASCIZ  <CR><LF><LF>
                        .EVEN
                .IIF NDF ...V7,...V7=-1
                .REPT   4.
                        .WORD   ...V7
                .ENDR
                DLBEND::
                .ENDC
                .IIF NDF TPS,TPS=:^o177564
                .IIF NDF TPB,TPB=:^o177566
        000012  LF=:^o12
        000015  CR=:^o15
        001000  B$BOOT=:^o1000
        004716  B$DEVN=:^o4716
        004722  B$DEVU=:^o4722
        004730  B$READ=:^o4730
                .IF NDF B$DNAM
                .IF EQ  MMG$T
                B$DNAM=:^RDL
                .IFF
                B$DNAM=:^RDLX
                .ENDC ; EQ MMG$T
                .ENDC ; NDF B$DNAM
000062          .ASECT
        000062  .=^o62
000062  000000'         .WORD   DLBOOT,DLBEND-DLBOOT,B.READ-DLBOOT
000064  001000
000066  000210
000000          .PSECT  DLBOOT
000000  000240  DLBOOT::NOP
000002  000415          BR      BOOT1-2.
        000100  ...V2=^o100
                .IRP    X       <UBUS,QBUS>
                ...V3=0
                .IIF    IDN     <X>     <UBUS>  ...V3=1.
                .IIF    IDN     <X>     <QBUS>  ...V3=2.
                .IIF    IDN     <X>     <CBUS>  ...V3=4.
                .IIF    IDN     <X>     <UMSCP> ...V3=^o10
                .IIF    IDN     <X>     <QMSCP> ...V3=^o20
                .IIF    IDN     <X>     <CMSCP> ...V3=^o40
                .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
                ...V2=...V2!...V3
                .ENDR
        000000  ...V3=0
        000001  .IIF    IDN     <UBUS>   <UBUS>  ...V3=1.
                .IIF    IDN     <UBUS>   <QBUS>  ...V3=2.
                .IIF    IDN     <UBUS>   <CBUS>  ...V3=4.
                .IIF    IDN     <UBUS>   <UMSCP> ...V3=^o10
                .IIF    IDN     <UBUS>   <QMSCP> ...V3=^o20
                .IIF    IDN     <UBUS>   <CMSCP> ...V3=^o40
                .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
        000101  ...V2=...V2!...V3
        000000  ...V3=0
                .IIF    IDN     <QBUS>   <UBUS>  ...V3=1.
        000002  .IIF    IDN     <QBUS>   <QBUS>  ...V3=2.
                .IIF    IDN     <QBUS>   <CBUS>  ...V3=4.
                .IIF    IDN     <QBUS>   <UMSCP> ...V3=^o10
                .IIF    IDN     <QBUS>   <QMSCP> ...V3=^o20
                .IIF    IDN     <QBUS>   <CMSCP> ...V3=^o40
                .IIF    EQ      ...V3   .ERROR;?SYSMAC-E-Invalid C O N T R O L, found - UBUS,QBUS;
        000103  ...V2=...V2!...V3
        000032' .=BOOT1-6.
000032     020          .BYTE   ^o20,...V2,^o20,^o^C<20+...V2+20>
000033     103
000034     020
000035     234
                .IF     EQ      <1-1>
000036  000400          BR      BOOT1
                .IFF
                .IF     EQ      <1-2.>
```

```
                          BMI     BOOT1
                  .IFF
                          .ERROR;?SYSMAC-E-Invalid S I D E S, expecting 1/2, found - 1;
                  .ENDC
                  .ENDC
 4
 5         000040'        . = DLBOOT+40                ;PUT THE JUMP BOOT INTO SYSCOM AREA
 6 000040  000137  BOOT1: JMP     @#BOOT-DLBOOT        ;START THE BOOTSTRAP
           000600


 1                        .SBTTL  BOOTSTRAP READ ROUTINE
 2
 3                        .ENABL  LSB
 4
 5         000210'        . = DLBOOT+210
 6 000210  005004  B.READ: CLR    R4                   ;CLEAR TRACK COUNTER
 7 000212  162700  1$:    SUB     #DLBPT,R0            ;COUNT DOWN ANOTHER WHOLE TRACK
           000024
 8 000216  103403         BLO     2$                   ;IF OVERFLOW, DONE
 9 000220  062704         ADD     #IOHS,R4             ;ADD IN ANOTHER TRACK
           000100
10 000224  000772         BR      1$                   ;LOOP FOR MORE
11
12 000226  062700  2$:    ADD     #DLBPT,R0            ;CORRECT TRACK COUNTER
           000024
13 000232  006300         ASL     R0                   ;CONVERT REMAINDER TO SECTOR IN TRACK
14 000234  050400         BIS     R4,R0                ;MERGE SECTOR WITH TRACK/CYL

16 000236  016705         MOV     BOTCSR,R5            ;GET ADDRESS OF CONTROLLER
           000344
17 000242  062705         ADD     #RLDA,R5             ;POINT TO DISK ADDRESS REGISTER
           000004
18 000246  016567         MOV     RLCS-RLDA(R5),B.DLCS ;GET CURRENT CSR VALUE
           177774
           000174
19 000254  042767         BIC     #^C<CSDS01>,B.DLCS   ;ISOLATE CURRENT UNIT NUMBER
           176377
           000166

21 000262  004767         CALL    B.SEEK               ;SEEK TO PROPER TRACK
           000066
22 000266  005401         NEG     R1                   ;NEGATE WORD COUNT

24 000270  010265         MOV     R2,RLBA-RLDA(R5)     ;SET BUS ADDRESS
           177776
                  .
                  .
                  .
28 000274          DLREAD:

30 000274  010165         MOV     R1,RLMP-RLDA(R5)     ;SET WORD COUNT
           000002
31 000300  010015         MOV     R0,@R5               ;SET DISK ADDRESS
                          .
                          .
                          .
36 000302  004067         JSR     R0,B.XCT             ;EXECUTE THE READ
           000136
37 000306  000014          .WORD  FNREAD               ;READ FUNCTION CODE
38 000310  000241         CLC                          ;ENSURE CARRY=0 BEFORE RETURN
39 000312  100053         BPL     5$                   ;SUCCESS, EXIT

41 000314  011503         MOV     @R5,R3               ;GET LAST DISK ADDRESS
                          .
                          .
                          .
46 000316  042703         BIC     #^C<IOSA>,R3         ;CLEAR ALL BUT SECTOR ADDRESS
           177700
47 000322  022703         CMP     #DLBPT*2,R3          ;TRACK OVERRUN?
           000050
48 000326  001156         BNE     BIOERR               ;IF NOT, REAL ERROR, EXIT
```

```
   000330  011503           MOV     @R5,R3                 ;GET DISK ADDRESS
                            .
                            .
                            .
   000332  160003           SUB     R0,R3                  ;COMPUTE SECTORS TRANSFERRED
   000334  000303           SWAB    R3                     ;CONVERT SECTORS TO WORD COUNT
   000336  006203           ASR     R3
   000340  060301           ADD     R3,R1                  ;REMOVE WORDS TRANSFERRED

   000342  011500           MOV     @R5,R0                 ;GET DISK ADDRESS
                            .
                            .
                            .
   000344  062700           ADD     #IOHS-<DLBPT*2>,R0     ;INCREMENT SURFACE/TRACK
           000030
   000350  012746           MOV     #DLREAD-DLBOOT,-(SP)   ;CALL TO SEEK NEXT TRACK, THEN READ IT
           000274
   000354                   .BR     B.SEEK                 ;SEEK NOW

   000354  004067  B.SEEK:  JSR     R0,B.XCT               ;EXECUTE READ HEADERS
           000064
   000360  000010            .WORD  FNRDH                  ;READ HEADER FUNCTION CODE

   000362  016503           MOV     RLMP-RLDA(R5),R3       ;GET CURRENT DISK TRACK AND SURFACE
           000002
                            .
                            .
                            .
   000366  042703           BIC     #IOHS!IOSA,R3          ;CLEAR SURFACE/SECTOR TO GET
           000177
                                                           ; CURRENT TRACK
   000372  010004           MOV     R0,R4                  ;COPY DESIRED DISK ADDRESS
   000374  042704           BIC     #IOHS!IOSA,R4          ;CLEAR SURFACE/SECTOR TO GET
   000177
                                                           ; DESIRED TRACK
   000400  160403           SUB     R4,R3                  ;SUBTRACT DESIRED FROM CURRENT TRACK
   000402  103003           BCC     3$                     ;IF CURRENT >= DESIRED,
                                                           ; SEEK OUTWARD BY DIFF
   000404  005403           NEG     R3                     ;MAKE POSITIVE DIFFERENCE OF
                                                           ; DELTA POSITION
   000406  052703           BIS     #SKDIR,R3              ;INDICATE MOVE TOWARD SPINDLE
           000004
   000412  032700  3$:      BIT     #IOHS,R0               ;DO WE DESIRE SURFACE 1?
           000100
   000416  001402           BEQ     4$                     ;NO, LEAVE SURFACE SELECT 0
   000420  052703           BIS     #SKHS,R3               ;SET BIT TO SELECT SURFACE 1
           000020
   000424  005203  4$:      INC     R3                     ;SET MARKER BIT

   000426  010315           MOV     R3,@R5                 ;LOAD DIFFERENCE WORD

                            .
                            .
                            .
   000430  004067           JSR     R0,B.XCT               ;EXECUTE A SEEK
           000010
   000434  000006            .WORD  FNSEEK                 ;SEEK FUNCTION CODE
   000436  100512           BMI     BIOERR                 ;IF PL, OK

   000440  010015           MOV     R0,@R5                 ;SET ACTUAL DISK ADDRESS
                            .
                            .
                            .
   000442  000207  5$:      RETURN                         ;RETURN


                   ; EXECUTE THE FUNCTION IN R3 AND RETURN ERROR STATUS

   000444  012003  B.XCT:   MOV     (R0)+,R3               ;GET FUNCTION CODE
```

```
109 000446  052703          BIS     (PC)+,R3                ;ADD UNIT BITS TO FUNCTION CODE
110 000450  000000  B.DLCS:  .WORD  0                       ;BOOTED UNIT NUMBER
111 000452  010365          MOV     R3,RLCS-RLDA(R5)        ;EXECUTE FUNCTION
            177774
112 000456  032765  6$:     BIT     #CSERR!CSCRDY,RLCS-RLDA(R5) ;WAIT FOR COMPLETION OR ERROR
            100200
            177774
                            .
                            .
                            .
118 000464  001774          BEQ     6$                      ;NEITHER, LOOP
119 000466  000200          RTS     R0                      ;RETURN WITH N=1 IF ERROR
                            .
                            .
                            .
  6         000600'         . = DLBOOT+600

  8 000600  012706  BOOT:   MOV     #10000,SP               ;SET STACK POINT
            010000

 10 000604  013746          MOV     @(PC)+,-(SP)
 11 000606  174400  BOTCSR:  .WORD  DL$CSR
 12 000610  042716          BIC     #^C1400,@SP             ;STRIP TO UNIT NUMBER
            176377
 13 000614  000316          SWAB    @SP                     ;MOVE TO BITS 0-1

 15 000616  012700          MOV     #2,R0                   ;READ IN SECOND PART OF BOOT
            000002
 16 000622  012701          MOV     #4*256.,R1              ;FOUR BLOCKS TO READ
            002000
 17 000626  012702          MOV     #1000,R2                ;INTO LOCATION 1000
            001000
 18 000632  004767          CALL    B.READ                  ;READ THE REST OF THE BOOT
            177352
 19 000636  012737          MOV     #B.READ-DLBOOT,@#B$READ ;STORE START LOCATION OF READ ROUTINE
            000210
            004730
 20 000644  012737          MOV     #B$DNAM,@#B$DEVN        ;STORE RAD50 DEVICE NAME
            015370
            004716

 22 000652  012637          MOV     (SP)+,@#B$DEVU          ;SET THE UNIT NUMBER IN THE BOOT
            004722
                            .
                            .
                            .
 26 000656  000137          JMP     @#B$BOOT                ;GO DO THE BOOT WORK
    001000
 27
 28 000662                  .DREND  DL
 29
                    .IF B <>
    001774          .PSECT  DLDVR
                    .IFF
                    .PSECT
                    .ENDC
                    .IIF NDF DL$END,DL$END::
                    .IF EQ  .-DL$END
                    .IF NE MMG$T!<0&2.>
                    $RLPTR::.WORD   0
                    $MPPTR::.WORD   0
                    $GTBYT::.WORD   0
                    $PTBYT::.WORD   0
                    $PTWRD::.WORD   0
                    .ENDC
                    .IF NE ERL$G!<0&1>
                    $ELPTR::.WORD   0
                    .ENDC
                    .IF NE TIM$IT!<0&4.>
                    $TIMIT::.WORD   0
                    .ENDC
                    $INPTR::.WORD   0
                    $FKPTR::.WORD   0
                    .IIF NDF ...V22 ...V22=0
                    .IF NE  ...V22&^o40000
                    DL$X64 =:.
                    .REPT   16.
                            .WORD   0
                    .ENDR
```

```
                .ENDC
                .GLOBL  DLSTRT
                DLEND==.
                .IFF
000662          .PSECT  DLBOOT
                .IIF LT <DLBOOT-.+^o664>,.ERROR;?SYSMAC-E-Primary boot too large;
        000664' .=DLBOOT+^o664
000664  004167  BIOERR: JSR     R1,REPORT
        000002
000670  000753           .WORD  IOERR-DLBOOT
000672  012700  REPORT: MOV     #BOOTF-DLBOOT,R0
        000740
000676  012702          MOV     #30004$-DLBOOT,R2
        000722
000702  004712          CALL    @R2
000704  011100          MOV     @R1,R0
000706  004712          CALL    @R2
000710  012700          MOV     #CRLFLF-DLBOOT,R0
        000764
000714  004712          CALL    @R2
000716  000000  30003$: HALT
000720  000776          BR      30003$
000722  105737  30004$: TSTB    @#TPS
        177564
000726  100375          BPL     30004$
000730  112037          MOVB    (R0)+,@#TPB
        177566
000734  001372          BNE     30004$
000736  000207          RETURN
000740     015  BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"
000741     012
000742     077
000743     102
000744     117
000745     117
000746     124
000747     055
000750     125
000751     055
000752     000
000753     111  IOERR:  .ASCII  "I/O error"
000754     057
000755     117
000756     040
000757     145
000760     162
000761     162
000762     157
000763     162
000764     015  CRLFLF: .ASCIZ   <CR><LF><LF>
000765     012
000766     012
000767     000
                        .EVEN
                .IIF NDF ...V7,...V7=-1
        000004  .REPT   4.
                        .WORD   ...V7
                .ENDR
000770  177777          .WORD   ...V7
000772  177777          .WORD   ...V7
000774  177777          .WORD   ...V7
000776  177777          .WORD   ...V7
001000          DLBEND::
                .ENDC
31                      .IF NE MMG$T
```

```
                     .SBTTL  FETCH/LOAD CODE
                     ;+
                     ;       FETCH
                     ;
                     ;       ENTRY:  R0  = STARTING ADDRESS OF THIS HANDLER SERVICE ROUTINE.
                     ;               R1  = ADRESS OF GETVEC ROUTINE.
                     ;               R2  = VALUE $SLOT*2. (LENGTH OF THE $PNAME TABLE IN BYTES.)
                     ;               R3  = TYPE OF ENTRY.
                     ;               R4  = ADDRESS OF SY READ ROUTINE.
                     ;               R5 -> $ENTRY SLOT FOR THIS HANDLER.
                     ;
                     ;-


   001000  010501  FETCH:  MOV     R5,R1                   ; SAVE PTR TO DL'S $ENTRY SLOT
   001002  011505          MOV     @R5,R5                  ; GET ADDRESS OF DLLQE
   001004  016504          MOV     DLCSR-DLLQE(R5),R4      ; GET CSR FOR DL
           000024
   001010  005046          CLR     -(SP)                   ; SPACE FOR RETURN VALUE
   001012                  .MFPS                           ; GET PROCESSOR STATUS
   001024                  .MTPS   #340                    ; RAISE PROCESSOR PRIORITY LEVEL TO 7
   001044  013746          MOV     @#NXM.V+2,-(SP)         ;;;SAVE CURRENT NXM TRAP PSW
           000006
   001050  013746          MOV     @#NXM.V,-(SP)           ;;;SAVE CURRENT NXM TRAP VECTOR
           000004
   001054                  .ADDR   #$DLNXM,-(SP)           ;;;BUILD ADDRESS TO OUR TRAP ROUTINE
   001062  012637          MOV     (SP)+,@#NXM.V           ;;;SET UP THE NXM VECTOR
           000004
   001066  012737          MOV     #340,@#NXM.V+2          ;;;SET UP THE NXM PSW
           000340
           000006
   001074  005764          TST     RLBAE(R4)               ;;;BAE REGISTER EXIST?
           000010
   001100  012637          MOV     (SP)+,@#NXM.V           ;;;MAYBE, FIRST RESTORE NXM VECTOR
           000004
   001104  012637          MOV     (SP)+,@#NXM.V+2         ;;;       AND NXM PSW
           000006
   001110  006166          ROL     2(SP)                   ;;;SAVE THE CARRY BIT
           000002
   001114                  .MTPS                           ; RESTORE PREVIOUS PRIORITY LEVEL
   001126  006026          ROR     (SP)+                   ; RESTORE CARRY
   001130  103403          BCS     20$                     ; NOT AN RLV12... BR IS NOT AN ERROR

                            .IF EQ ERL$G
   001132  012765          MOV     #NOP,$RLV1A-DLLQE(R5)   ; Change BR to NOP for RLV12
           000240
           001156
                            .IFF
                           MOV     #NOP,R0                 ; R0="NOP"
                           MOV     R0,$RLV1A-DLLQE(R5)     ; PATCH 'BR' TO 'NOP' FOR RLV12
                           MOV     R0,$RLV1B-DLLQE(R5)     ; PATCH ERROR LOGGING CODE SO IT KNOWS
                           MOV     R0,$RLV1C-DLLQE(R5)     ;   ABOUT EXTRA REGISTER (RLBAE)
                            .ENDC ;EQ ERL$G

   001140  000241  20$:    CLC

                   ;+
                   ;       LOAD UP LOCAL VARIABLES WITHIN DL
                   ;-

   001142  010105          MOV     R1,R5                   ; RESTORE PTR TO DLLQUE TO R5
   001144  011100          MOV     @R1,R0                  ; GET ADDRESS OF DLLQE
   001146  013704          MOV     @#SYSPTR,R4             ; GET START OF RMON
           000054
   001152  016403          MOV     $H2UB(R4),R3            ; R3 = UBVECT POINTER
           000460
   001156  010360          MOV     R3,<H2UB-DLBASE>(R0)    ; H2UB = ADDRESS OF UBVECT
           002244
   001162  016403          MOV     $PNPTR(R4),R3           ; R3 = RMON OFFSET TO PNAME TABLE
           000404
   001166  060403       ADD        R4,R3                   ; R3 -> PNAME TABLE ADDRESS
   001170  010360          MOV     R3,<$PNMPT-DLBASE>(R0)  ; $PNMPT -> PNAME TABLE ADDRESS
           002242
   001174  060203          ADD     R2,R3                   ; R3 -> $ENTRY TABLE
   001176  010360          MOV     R3,<$ENTPT-DLBASE>(R0)  ; $ENTPT -> $ENTRY TABLE
           002240
   001202  010560          MOV     R5,<DLENT-DLBASE>(R0)   ; DLENT -> DL'S $ENTRY TABLE ENTRY
```

```
        002250
001206  160205          SUB     R2,R5                   ; R5 -> DL'S $PNAME TABLE ENTRY
001210  010560          MOV     R5,<DLPNA-DLBASE>(R0)   ; DLPNA -> DL'S $PNAME TABLE ENTRY
        002252

```

Allocate permanent UMRs if using UNIBUS mapping registers:

```
                ;+
                ;       ALLOCATE PERMANENT UMRS TO POINT INTO DL'S INTERNAL DMA BUFFERS,
                ;       DLBBUF AND DLFILL, AND GET THE UNIBUS VIRTUAL ADDRESS.
                ;-

001214  012701          MOV     #<DLBBUF-DLBASE>,R1     ; R1 = LOW 16 BITS OF DMABUF ADDRESS
        002076
001220  060001          ADD     R0,R1                   ;
001222  005002          CLR     R2                      ; R2 = HIGH 6 BITS OF DMABUF ADDRESS
001224  016004          MOV     <DLPNA-DLBASE>(R0),R4   ; R4 -> PNAME ENTRY FOR DL
        002252
001230  016005          MOV     <H2UB-DLBASE>(R0),R5    ; GET UB ENTRY ADDRESS
        002244
001234  010046          MOV     R0,-(SP)                ; SAVE DL STARTING ADDRESS
001236  012700          MOV     #NOUMRS,R0              ; R0 = NUMBER OF UMRS REQUIRED
        000001
001242  004765          CALL    UB.ALL(R5)              ; CALL ALLUMR
        000002
001246  012600          MOV     (SP)+,R0                ; RESTORE DL STARTING ADDRESS
001250  103411          BCS     30$                     ; COULDN'T GET UMR, FAIL THE LOAD
001252  010160          MOV     R1,<BUFADL-DLBASE>(R0)  ; STORE UNIBUS VIRTUAL ADDRESS LOW
        002262
001256  006302          ASL     R2                      ; SHIFT
001260  006302          ASL     R2                      ; HI BITS LEFT 4
001262  006302          ASL     R2                      ; TO GET THEM INTO THE
001264  006302          ASL     R2                      ; CORRECT PLACE
001266  010260          MOV     R2,<BUFADH-DLBASE>(R0)  ; STORE UNIBUS VIRTUAL ADDRESS HIGH
        002260
001272  000241          CLC                             ; LOAD SUCCEEDED
001274  000207  30$:    RETURN



                ;

001276  052766  $DLNXM: BIS     #1,2(SP)                ;SET THE CARRY BIT
        000001
        000002
001304  000002          RTI

```

Routine to unload DL and release any UMRs:

```
                ;+
                ;       RELEAS
                ;
                ;       ROUTINE TO UNLOAD DL
                ;
                ;       ENTRY:  SAME AS FOR LOAD.
                ;
                ;-

                .ENABL  LSB
001306          RELEAS::
001306  010501          MOV     R5,R1                   ; R1 = $ENTRY SLOT FOR DM
001310  160201          SUB     R2,R1                   ; R2 -> $PNAME SLOT FOR DM
001312  013704          MOV     @#SYSPTR,R4             ; GET START OF RMON
        000054
001316  016405          MOV     $H2UB(R4),R5            ; R5 = UB ENTRY VECTOR
        000460
001322  004765          CALL    UB.RLS(R5)              ; RELEASE UMRS
        000004
001326  000207          RETURN                          ; AND EXIT

                        .ENDC ;NE MMG$T

        000001          .END
```

Symbol Table From Assembly

```
ABTIO$  001000          DLBWCT  002072R     002 DOC$UN= 000000
BAREA   000606          DLCC    000154R     002 DRETRY= 000742R     002
BIOERR  000664R     003 DLCODE  001156R     002 DVC.CT  000006
BOOT    000600R     003 DLCQE   000010RG    002 DVC.DE  000010
BOOTF   000740R     003 DLCSR   000032R     002 DVC.DK  000004
BOOT1   000040R     003 DLDA    001150R     002 DVC.DL  000012
BOTCSR  000606R     003 DLDSIZ  023742          DVC.DP  000011
BTCSR = 004124          DLELNK  000370R     002 DVC.LP  000007
BUFADH  002266R     002 DLEND = 002316RG    002 DVC.MT  000005
BUFADL  002270R     002 DLENT   002256R     002 DVC.NI  000013
BUFEND  002234R     002 DLERJM  001012R     002 DVC.NL  000001
BUFSIZ  000054          DLEROR  001606R     002 DVC.PS  000014
BUS$    000100          DLERRH  001470R     002 DVC.SB  000020
BUS$C   020000          DLEXFR  001620R     002 DVC.SI  000016
BUS$M   020100          DLFBLK  002236R     002 DVC.SO  000017
BUS$Q   000100          DLFILL  002234R     002 DVC.TP  000003
BUS$U   000000          DLFLNK  000250R     002 DVC.TT  000002
BUS$X   020100          DLGSIZ  001672R     002 DVC.UK  000000
B$BOOT  001000          DLGST   001740R     002 DVC.VT  000015
B$DEVN  004716          DLICQE  002264R     002 DVM.DM  000002
B$DEVU  004722          DLILQE  002262R     002 DVM.DX  000001
B$DNAM  015370          DLINT   001250RG    002 DVM.NF  000200
B$READ  004730          DLLQE   000006RG    002 DVM.NS  000001
B.DLCS  000450R     003 DLNBAD  000012          DV2.V2  040000
B.READ  000210R     003 DLPNA   002260R     002 EIS$I = 000001
B.SEEK  000354R     003 DLQCOM  001654R     002 EOF$    020000
B.XCT   000444R     003 DLQPTR  000602R     002 ERL$G = 000000
CONFG2  000370          DLRCNT  000010          FETCH   001000R     003
CR      000015          DLREAD  000274R     003 FILST$  100000
CRLFLF  000764R     003 DLREG   000006          FIX$ED= 000001
CSBA16  000020          DLREV   000750R     002 FNGSTS  000004
CSBA17  000040          DLRST   001776R     002 FNNOP   000000
CSCRDY  000200          DLRTY   000706R     002 FNRDH   000010
CSDCRC  004000          DLSEEK  001726R     002 FNRDNH  000016
CSDE    040000          DLSIZE  023742          FNREAD  000014
CSDLT   010000          DLSIZ2  047742          FNSEEK  000006
CSDRDY  000001          DLSLOT  002254R     002 FNWCHK  000002
CSDS0   000400          DLSQUE  002032R     002 FNWRIT  000012
CSDS01  001400          DLSTRT  000000RG    002 FN$RED  000377
CSERR   100000          DLSTS   102405          FN$REP  000374
CSERRC  036000          DLSYS   000006RG    002 FN$SIZ  000373
CSFUN   000016          DLTRAK  000746R     002 FN$WRT  000376
CSHCRC  004000          DLTRAN  000374R     002 GSGS    000002
CSHNF   010000          DLTSIZ  000052          GSMARK  000001
CSIE    000100          DLUNIT  001162R     002 GSRST   000010
CSNXM   020000          DLUSIZ  000164R     002 HDERR$  000001
CSOPI   002000          DLWC    001140R     002 HNDLR$  004000
DISCSR  000174          DLWPT   012000          HS2.BI  000001
DLADDR  000600R     002 DLWTRK  001126R     002 HS2.KI  000002
DLBADD  002070R     002 DLXCT   002010R     002 HS2.KL  000004
DLBASE= 000006R     002 DLXFER  001114R     002 HS2.KU  000010
DLBBUF  002104R     002 DL$COD  000005          HS2.MO  000020
DLBEND  001000RG    003 DL$CSR= 174400 G        H2UB    002252R     002
DLBMEM  002100R     002 DL$END  002300RG    002 INSCSR  000176
DLBOOT  000000RG    003 DL$NAM= 015340          INSDAT  000200
DLBPAR  002076R     002 DL$UN = 000002 G        INSSYS  000202
DLBPT   000024          DL$VEC= 000160 G        IOCA    077600

IOCA0   000200          RELEAS  001306RG    003 $ENTPT  002246R     002
IOERR   000753R     003 REPORT  000672R     003 $FKPTR  002314RG
```

Figure A–3: XL Communications Handler

```
XL - Communications Driver     MACRO V05.05  Thursday 18-Apr-91 13:00
Table of contents
```



```
                    .MCALL  .MODULE
  000000            .MODULE XL,VERSION=36,COMMENT=<Communications Driver>

                    ;                       COPYRIGHT 1989, 1990, 1991 BY
                    ;           DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
                    ;                         ALL RIGHTS RESERVED
                    ;
                    ;THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
                    ;ONLY  IN  ACCORDANCE  WITH  THE TERMS  OF  SUCH  LICENSE AND WITH THE
                    ;INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY OTHER
                    ;COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
                    ;OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS HEREBY
                    ;TRANSFERRED.
                    ;
                    ;THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT NOTICE
                    ;AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT
                    ;CORPORATION.
                    ;
                    ;DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF ITS
                    ;SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.

                            .ENABL  LC

                    ;+
                    ;
                    ; FACILITY:     RT-11 Device driver
                    ;
```

```
                ; FUNCTIONAL DESCRIPTION:
                ;
                ;       This driver aids in the writing of virtual terminal software. It
                ;       supports the XON/XOFF protocol in that if receives too many chars
                ;       it will transmit a CTRL/S and send a CTRL/Q when it again has room.
                ;       It will also stop transmitting if it receives a CTRL/S and resume
                ;       on a CTRL/Q. Normal RT-11 READ/WRITE commands can be done to the
                ;       plus various special functions. On any data transfer, chars are
                ;       striped to seven bits and chars of value zero are ignored. On output
                ;       the character following a carriage return is not output.
                ;
```

CONDITIONAL ASSEMBLY SUMMARY

```
                        .SBTTL  Conditional assembly summary


        000001          XL$LUN  = 1
        000001          XL$MTY  = 1
        000001          XL$PDP  = 1
                        ;
                        ;
        000001          MMG$T   = 1
        000001          TIM$IT  = 1
                ;+
                ;COND
                ;
                ;       XL$DVE  (0)             support for DLV11E
                ;                0              no support
                ;                1              support
                ;
                ;       XL$PC   (0)             support for PRO300 series
                ;                0              no support
                ;                1              support
                ;
                ;       XL$SBC  (0)             support for SBC-11/21[+] and MXV SLUs
                ;                0              no support
                ;                1              support
                ;
                ;       Exactly one of XL$PC, XL$DVE and XL$SBC
                ;               may be specified.
                ;
                ;       XL$PDT  (0)             support PDT lights
                ;                0              no support
                ;                1              support
                ;
                ;       XL$PDT is ignored if XL$PC is 1
                ;
                ;       XL$PRI  (4)             interrupt priority
                ;               (5)             if XL$SBC is 1
                ;                4-7            range
                ;
                ;       XL$CSR  (176500)        CSR address
                ;               (173300)        if XL$PC is 1
                ;
                ;       XL$VEC  (300)           Vector address
                ;               (210)           if XL$PC is 1
                ;
                ;       XL$MTY  (0)             No support multiterminal handler hooks
                ;                1              Support for multiterminal handler hooks
                ;
                ;       XL$MTY may be 1 only when XL$PC is 0.
                ;
                ;       XL$LUN  (1)             Line number to use in multiterminal
                ;
                ;       MMG$T                   std conditional
                ;       TIM$IT                  std conditional (no code effect)
                ;       ERL$G                   std conditional (no code effect)
                ;
                ;-
```

MACROS AND DEFINITIONS

```
                          .SBTTL  MACROS AND DEFINITIONS

                          .LIBRARY        "SRC:SYSTEM.MLB"

```

Prepare for using standard definitions:

```
                  ; Declare the RT system macros we'll be using

                          .MCALL  .DRDEF  .MTPS   .INTEN
                          .MCALL  .ASSUM  .ADDR   .BR
                          .MCALL  .MTSTA

                  ; Define and verify some conditionals

                          .IIF NDF XL$DVE XL$DVE  = 0     ;Default to non DLV11-E interface

                          .IIF NDF XL$PC  XL$PC   = 0     ;Default to non PRO-3xx support

                          .IIF NDF XL$PDT XL$PDT  = 0     ;Default to no PDT lights display

                          .IIF NDF XL$SBC XL$SBC  = 0     ;Default to non SBC-11 interface

                          .IIF NDF XL$MTY XL$MTY  = 0     ;Default to no support for MTY hooks
                          .IIF NDF XL$LUN XL$LUN  = 1     ;Default to LUN 1

 000000           .Assume <XL$PC & <XL$DVE ! XL$SBC>> EQ 0 MESSAGE=<Conflicting options>
 000000           .Assume <XL$DVE + XL$SBC> LE 1 MESSAGE=<Conflicting options>
 000000           .Assume <XL$PC & XL$MTY> EQ 0 MESSAGE=<Conflicting options>

                  ; Set the audit trail

         000000           .XLGEN  = XL$PC ! <XL$DVE * 2> !
<XL$SBC * 4> ! <XL$PDT * 10>
         000020           .XLGEN  = .XLGEN ! <XL$MTY * 20>

 000000                   .AUDIT  .XL                     ;The handler
 000110  107123           .WORD   .AUDIT
 000112  000044           .WORD   .XL
 000114  177777           .WORD   -1
 000000                   .AUDIT  .XLGEN                  ; and the conditionals
 000114  000020           .WORD   .XLGEN
 000116  177777           .WORD   -1

                  ; Define the device
                  ;       o Entered on all aborts
                  ;       o handles .SPFUN system call

                          .IF NE XL$PC
                          XL$CSR  = 173300                ;Force these for a PRO
                          XL$VEC  = 210
                          .ENDC ;NE XL$PC

                          .IIF NDF XL$PRI XL$PRI  = 4     ;Interrupt processing level

                          .IF NE XL$SBC
                          XL$PRI  = 5                     ;Force this for SBC-11/12[+] and MXV
                          .ENDC ;NE XL$SBC

```

The .DRDEF macro with expansion:

```
 000000                   .DRDEF  XL,57,<ABTIO$!HNDLR$!SPFUN$>,0,176500,300,DMA=NO
 000100  000040           .WORD   40
 000176  176500           .WORD   XL$CSR

                          .IF EQ XL$PC
```

The .DRPTR macro with expansion:

```
   000200                .DRPTR  FETCH=FETCH,LOAD=LOAD,UNLOAD=UNLOAD,RELEAS=RELEAS
   000000  031066        .RAD50  "HAN"
   000002  002704'       .WORD   FETCH
   000004  003312'       .WORD   RELEAS
   000006  002704'       .WORD   LOAD
   000010  003256'       .WORD   UNLOAD
   000021     000        .BYTE   0
                         .IFF ;EQ XL$PC
                         .DRPTR  UNLOAD=UNLOAD,RELEAS=RELEAS
                         .ENDC ;EQ XL$PC

```

The .DREST macro with expansion:

```
   000022                .DREST  CLASS=DVC.VT
   000000  031066        .RAD50  "HAN"
   000020     015        .BYTE   DVC.VT
   000021     000        .BYTE   0
   000032  000000        .WORD   0
   000036  000000        .WORD   0
   000072  000000        .WORD   0
   000074  000000        .WORD   0

```

Support the following special functions (.DRSPF):

```
   000076                .DRSPF  <201>                    ;Reset 'received XOFF from host' flag
   000022     176                .BYTE   176
   000023     200                .BYTE   200
   000024     000                .BYTE   0
   000025     000                .BYTE   0
   000026     000                .BYTE   0
   000027     000                .BYTE   0
   000030  000000                .WORD   000000
                                                          ; and send XON to host
   000032                .DRSPF  <202>                    ;Set/clear BREAK
   000022     176                .BYTE   176
   000023     200                .BYTE   200
   000024     000                .BYTE   0
   000025     000                .BYTE   0
   000026     000                .BYTE   0
   000027     000                .BYTE   0
   000030  000000                .WORD   000000
                                                          ; word count <> 0, BREAK
                                                          ; word count =  0, end BREAK
   000032                .DRSPF  <203>                    ;Special read. Word count is maximum
   000022     176                .BYTE   176
   000023     200                .BYTE   200
   000024     000                .BYTE   0
   000025     000                .BYTE   0
   000026     000                .BYTE   0
   000027     000                .BYTE   0
   000030  000000                .WORD   000000
                                                          ; number of bytes to read. Terminates
                                                          ; when number of bytes specified have
                                                          ; been read or when the input buffer
                                                          ; is empty. Always reads at least one
                                                          ; byte even if buffer is empty when
                                                          ; the read is issued.
   000032                .DRSPF  <204>                    ;Returns driver status in first word
   000022     176                .BYTE   176
   000023     200                .BYTE   200
   000024     000                .BYTE   0
   000025     000                .BYTE   0
   000026     000                .BYTE   0
   000027     000                .BYTE   0
   000030  000000                .WORD   000000
                                                          ; of buffer. High byte = driver edit
                                                          ; level. Low byte = XOFF status and
                                                          ; some modem signals.
   000032                .DRSPF  <205>                    ;Sets a flag which will cause
   000022     176                .BYTE   176
   000023     200                .BYTE   200
   000024     000                .BYTE   0
   000025     000                .BYTE   0
   000026     000                .BYTE   0
```

```
     000027     000                 .BYTE   0
     000030  000000                 .WORD   000000
                                                              ; interrupts to be turned off on
                                                              ; program exit
     000032                 .DRSPF  <206>                     ;Sets/Resets DTR
     000022     176                 .BYTE   176
     000023     200                 .BYTE   200
     000024     000                 .BYTE   0
     000025     000                 .BYTE   0
     000026     000                 .BYTE   0
     000027     000                 .BYTE   0
     000030  000000                 .WORD   000000
                                                              ; word count
<> 0, set DTR
                                                              ; word count = 0, reset DTR

                    ; Handler version number given to VTCOM in INIT message

             000022         $$$VER  == 18.                    ;VTCOM and XL must be a matched set

                    ; RT-11 System communications area

```

The following macros (through .TSTDF) use the standard definitions from SYSTEM.MLB:

```
                            .MCALL  .SYCDF
     000032                 .SYCDF                            ;Define system communications area

                    ; RMON Fixed offset area

                            .MCALL  .FIXDF  .CF1DF  .CF2DF
                            .MCALL  .SGNDF

     000032                 .FIXDF                            ;Define RMON fixed offsets

     000032                 .CF1DF                            ;Define config word 1 bits
     000032                 .CF2DF                            ;Define config word 2 bits
     000032                 .SGNDF                            ;Define SYSGEN features word bits

                    ; Multiterminal status block

                            .MCALL  .MSTDF

     000032                 .MSTDF                            ;Define .MTSTA status block

                    ; Handler header definitions

                            .MCALL  .HBGDF
     000032                 .HBGDF                            ;Define handler header

                    ; Handler hooks related definitions

                            .MCALL  .THKDF  .TCBDF  .TSTDF
     000032                 .TCBDF                            ;Define TCB offsets
     000032                 .THKDF                            ;Define handler hooks data structure
     000032                 .TSTDF                            ;Define T.STAT word bits

                    ; Input buffer definitions

             000100         BUFSIZ  = 64.                     ;Size of input buffer (in bytes)
             000020         STPSIZ  = BUFSIZ/4                ;Low-water mark (when to send XOFF)
             000060         RSTSIZ  = BUFSIZ*3/4              ;High-water mark (when to send XON)

                    ; Control Characters

             000012         C.LF    = 12                      ;Line feed
             000015         C.CR    = 15                      ;Carriage return
             000021         C.CTLQ  = 21                      ;XON (^Q)
             000023         C.CTLS  = 23                      ;XOFF (^S)
             000032         C.CTLZ  = 32                      ;End-of-file (^Z)

                    ; .SPFUN codes supported by driver

             000201         CLRDRV  = 201                     ;Reset 'received XOFF from host' flag
                                                              ; and send XON to host
             000202         BRKDRV  = 202                     ;Set/clear BREAK
```

```
                                                   ; word count <> 0, BREAK
                                                   ; word count =  0, end BREAK
       000203        SRDDRV  = 203                 ;Special read. Word count is maximum
                                                   ; number of bytes to read. Terminates
                                                   ; when number of bytes specified have
                                                   ; been read or when the input buffer
                                                   ; is empty. Always reads at least one
                                                   ; byte even if buffer is empty when
                                                   ; the read is issued.
       000204        STSDRV  = 204                 ;Returns driver status in first word
                                                   ; of buffer. High byte = driver edit
                                                   ; level. Low byte =
       000001                ST.XFH  = 000001      ;XOFF sent to host
       000002                ST.XOF  = 000002      ;XOFF received from host
       000004                ST.CTS  = 000004      ;Dataset: Clear To Send asserted
       000010                ST.CD   = 000010      ;Dataset: Carrier Detect asserted
       000020                ST.RI   = 000020      ;Dataset: Ring Indicate asserted

       000205        OFFDRV  = 205                 ;Sets a flag which will cause
                                                   ; interrupts to be turned off on
                                                   ; program exit
       000206        DTRDRV  = 206                 ;Sets/Resets DTR
                                                   ; word count <> 0, set DTR
                                                   ; word count = 0, reset DTR
             ;NOTE: if you add special function code, add them to .DRSPF too!

             ; Interface bit definitions

       040000        RC.RI   = 040000              ;Ring indicator
       020000        RC.CTS  = 020000              ;Clear to send
       010000        RC.CD   = 010000              ;Carrier detect
       000100        RC.IE   = 000100              ;Interrupt enable
       000004        RC.RTS  = 000004              ;Request to send
       000002        RC.DTR  = 000002              ;Data terminal ready

       000100        XC.IE   = 000100              ;Transmitter: interrupt enable

                     .IF NE XL$DVE
                     XC.SMK  = 170000              ;Speed mask
                     XC.SCE  = 004000              ;Speed change enable
                     .ENDC ;NE XL$DVE

                     .IF NE XL$SBC
                     XC.SMK  = 000070              ;Speed mask
                     XC.SCE  = 000002              ;Speed change enable
                     .ENDC ;NE XL$SBC

       000001        XC.BRK  = 000001              ; BREAK

                     .IF NE XL$PC

             ; PRO-3xx Interrupt controller registers

                     IC0DR   = 173200              ;Interrupt controller 0 data register
                     IC0CR   = IC0DR+2             ;Interrupt controller 0 csr register

             ; PRO-3xx Communications port registers

                     XL$BUF  = XL$CSR              ;Recv/Xmit buffer register
                     XL$CSA  = XL$CSR+2            ;CSR register A
                     XL$CSB  = XL$CSR+6            ;CSR register B
                     XL$MC0  = XL$CSR+10           ;Modem control register 0
                     XL$MC1  = XL$CSR+12           ;Modem control register 1
                     XL$BAU  = XL$CSR+14           ;Baud rate control register

             ; CSRA Write/Read register bit definitions

                     RPT.R0  = 000          ;Write/Read register 0
                             CRC.TR  = 300  ; Reset transmit underrun/end of message latch
                             CMD.RE  = 020  ; Reset external/status interrupts
                             CMD.CR  = 030  ; Channel reset
                             CMD.RT  = 050  ; Reset transmitter interrupt pending
                             CMD.ER  = 060  ; Reset error latches
                             CMD.EI  = 070  ; End of interrupt
                     RPT.R1  = 001          ;Write/Read register 1
                             W1.RIE  = 030  ; Receiver interrupt enable
                                            ;  (Int. on rec. char or special (no parity))
                             W1.TIE  = 002  ; Transmitter interrupt enable
```

```
                RPT.R2  = 002           ;Write/Read register 2
                RPT.R3  = 003           ;Write register 3
                        RCL.8   = 300   ; Receiver character length (8 bits)
                        W3.RXE  = 001   ; Receiver enable
                RPT.R4  = 004           ;Write register 4
                        CLK.16  = 100   ; 16x rate multiplier
                        STP.1   = 004   ; 1 stop bit
                        W4.EVN  = 002   ; Even parity
                        W4.PEN  = 001   ; Parity enable
                RPT.R5  = 005           ;Write register 5
                        TCL.8   = 140   ; Transmit character length (8 bits)
                        W5.SB   = 020   ; Send break
                        W5.TXE  = 010   ; Transmitter enable

        ; CSRB Write/Read register bit definitions

                RPT.R1  = 001           ;Write/Read register 1
                        W1.REQ  = 004   ; MUST be loaded with 004
                RPT.R2  = 002           ;Write/Read register 2
                        W2.REQ  = 000   ; MUST be loaded with 000
                        R2.IMK  = 034   ; Interrupt vector mask
                        IMK.BE  = 020   ; Transmit buffer empty
                        IMK.ES  = 024   ; External/Status change
                        IMK.CA  = 030   ; Received character available
                        IMK.SR  = 034   ; Special receiver condition

        ; Modem control Register bit definitions

                CLK.BG  = 000           ; Rx = RBRG,    Tx = TBRG       ->MD = none
                M0.DTR  = 020           ; Data terminal ready
                M0.RTS  = 010           ; Request to send
                M1.RI   = 100           ; Ring indicator
                M1.CTS  = 040           ; Clear to send
                M1.CD   = 020           ; Carrier detect
                .ENDC ;NE XL$PC

        ; Baud rate mask definitions (PRO-3xx, DLV11-E,F and MXV11-B)

                .IF NE <XL$PC ! XL$DVE>
                B.50    = 000                   ;    50 baud
                B.75    = 001                   ;    75 baud
                B.110   = 002                   ;   110 baud
                B.134   = 003                   ; 134.5 baud
                B.150   = 004                   ;   150 baud
                B.300   = 005                   ;   300 baud
                B.600   = 006                   ;   600 baud
                B.1200  = 007                   ;  1200 baud
                B.1800  = 010                   ;  1800 baud
                B.2000  = 011                   ;  2000 baud
                B.2400  = 012                   ;  2400 baud
                B.3600  = 013                   ;  3600 baud
                B.4800  = 014                   ;  4800 baud
                B.7200  = 015                   ;  7200 baud
                B.9600  = 016                   ;  9600 baud
                B.192K  = 017                   ; 19.2k baud
                .ENDC ;NE <XL$PC ! XL$DVE>

        ; Baud rate mask definitions [SBC-11 only]

                .IF NE XL$SBC
                B.300   = 000                   ;   300 baud
                B.600   = 001                   ;   600 baud
                B.1200  = 002                   ;  1200 baud
                B.2400  = 003                   ;  2400 baud
                B.4800  = 004                   ;  4800 baud
                B.9600  = 005                   ;  9600 baud
                B.192K  = 006                   ; 19.2K baud
                B.384K  = 007                   ; 38.4k baud
                .ENDC ;NE XL$SBC

        ; Miscellaneous definitions

177776          PS      =: 177776               ; Processor status word
000007          UNITMK  =: 007                  ;Q$UNIT unit number mask
000370          JOBMK   =: 370                  ;Q$JNUM job number mask

        ; Macro to define LSB of bit field

```

```
                         .MACRO  LSBDF   SYMBOL,VALUE
                                 SYMBOL = VALUE & <-VALUE>
                         .ENDM ;LSBDF
```

Block 0 of handler file

```
                         .SBTTL  Block 0 of handler file

```

The SPEED table is placed low in block 0 without conflicting with audit trail:

```
000032                   .ASECT
        000120           . = 120

                         .IF NE <XL$PC ! XL$DVE ! XL$SBC>
                 ; SPEED table. Mask for given speed is same as word offset into table.
                 ;       To select 134.5 bps, specify 134 in the SET command.

                 SPEEDT:
                          .IF NE <XL$DVE ! XL$PC>
                         .WORD   50.,    75.,    110.,   134.,   150.,   300.
                         .WORD   600.,   1200.,  1800.,  2000.,  2400.,  3600.
                         .WORD   4800.,  7200.,  9600.,  19200.
                          .ENDC ;NE <XL$DVE ! XL$PC>

                          .IF NE XL$SBC
                         .WORD   300.,   600.,   1200.,  2400.,  4800.,  9600.
                         .WORD   19200., 38400.
                          .ENDC ;NE XL$SBC

                         .WORD   0                       ;Table fence

                         .ENDC ;NE <XL$PC ! XL$DVE ! XL$SBC>

```

We must ensure that 0 fence for display CSRs is not overwritten:

```
000120           .Assume . LE DISCSR-2 MESSAGE=<Code before installation code too large>
```

INSTALLATION CODE

```
                         .SBTTL  INSTALLATION CODE

                         .ENABL  LSB

                         .IF EQ XL$MTY
                         .DRINS  XL
                         .IFF ;EQ XL$MTY
```

Ensure that install-time CSR is zero when defaulting to MTTY, so the handler always installs:

```
000120                   .DRINS  -XL
000172  000000           .WORD   0
000174  176500  DISCSR:  .WORD   -<-XL$CSR>
000176  000000  INSCSR:  .WORD   0
                         .ENDC ;EQ XL$MTY

000200  000401           BR      10$                     ;Install as a data device
000202  000416           BR      40$                     ; never as a system device

000204  013700  10$:     MOV     @#$SYPTR,R0             ;R0->$RMON
        000054
000210  032760           BIT     #PROS$,$CNFG2(R0)       ;Installing on a PRO-3xx?
        020000
        000370

                         .IF EQ XL$PC
000216  001010           BNE     40$                     ;Yes, then reject the installation
                          .IF NE XL$MTY
000220  105767           TSTB    I$MTTY                  ;Are handler hooks needed?
        000020
```

```
000224  001404          BEQ     20$                     ;Nope...
000226  005760          TST     $THKPT(R0)              ;Yes, is the support available?
        000000G
000232  001402          BEQ     40$                     ;Nope, reject the installation
```

Hooks cannot be established until handler is in memory, which doesn't happen until Fetch/Load:

```
000234  000400          BR      30$                     ;Yes, nothing to do until fetch/load
000236          20$:
                         .ENDC ;NE XL$MTY
                        .IFF ;EQ XL$PC
                        BEQ     40$                     ;Nope, then reject the installation
                        .ENDC ;EQ XL$PC
                        .IF EQ XL$PC
                         .IF NE <XL$DVE ! XL$SBC>
                        MOV     INSCSR,R0               ;R0->Receiver CSR
```

Speed set at install-time:

```
                        MOV     ISPEED,4(R0)            ;Set the speed (in transmitter CSR)
                         .ENDC ;NE <XL$DVE ! XL$SBC>
                        .IFF ;EQ XL$PC
                        MOVB    ISPEED,@#XL$BAU         ;Set the XMIT/RECV baud rate
                ; Things to do through csr A
                        MOV     #XL$CSA,R0              ;R0->csr A
                        MOVB    #CMD.CR,@R0             ;Reset channel A
                        MOVB    #CRC.TR,@R0             ;Reset transmitter underrun latch
                        MOVB    #RPT.R4,@R0             ;Select csr A, write register 4
                        MOVB    #<CLK.16!STP.1>,@R0     ; set clock rate x16, 1 stop bit
                        MOVB    #RPT.R3,@R0             ;Select csr A, write register 3
                        MOVB    #<W3.RXE!RCL.8>,@R0     ; set receiver enable, 8-bit chars
                        MOVB    #RPT.R5,@R0             ;Select csr A, write register 5
                        MOVB    #<W5.TXE!TCL.8>,@R0     ; set transmitter enable, 8-bit chars
                        MOVB    #RPT.R2,@R0             ;Select csr A, write register 2
                        MOVB    #0,@R0                  ; *** must be loaded with 0 ***
                        MOVB    #CMD.RE,@R0             ;Reset external/status interrupts
                ; Things to do through csr B
                        MOV     #XL$CSB,R0              ;R0->csr B
                        MOVB    #CMD.CR,@R0             ;Reset channel B
                        MOVB    #RPT.R2,@R0             ;Select csr B, write register 2
                        MOVB    #W2.REQ,@R0             ; *** ensure base vector of 0 ***
                        MOVB    #RPT.R1,@R0             ;Select csr B, write register 1
                        MOVB    #W1.REQ,@R0             ; *** ensure correct vector info ***
                ; Now we play with the interrupt controller
                        MOVB    #<30!3>,@#IC0CR         ;Enable comm port interrupts
                ; And finally, the modem
                        MOVB    #CLK.BG,@#XL$MC0        ;Set modem clock
                        .ENDC ;EQ XL$PC
000236  005727  30$:    TST     (PC)+                   ;Accept the installation (carry=0)
000240  000261  40$:    SEC                             ;Reject the installation (carry=1)
000242  000207          RETURN
                        .DSABL  LSB
                        .IF NE <XL$PC ! XL$DVE ! XL$SBC>
                         .IF NE <XL$DVE ! XL$SBC>
                        LSBDF   ...,XC.SMK              ;Determine lowest bit of speed mask
                         .ENDC ;NE <XL$DVE ! XL$SBC>
```

```
                   ISPEED:
                            .IF NE XL$PC
                           .WORD   <B.1200 * 20> + B.1200  ;Default to 1200 baud RECV and XMIT
                            .ENDC ;NE XL$PC

                            .IF NE <XL$DVE ! XL$SBC>
                           .WORD   <B.1200 * ...> ! XC.SCE ;Default to 1200 baud RECV and XMIT
                            .ENDC ;NE <XL$DVE ! XL$SBC>
                           .ENDC ;NE <XL$PC ! XL$DVE ! XL$SBC>

                           .IF NE XL$MTY
```

Default flag to MTTY if built for hooks support:

```
    000244     377  I$MTTY: .BYTE   -1                      ; : Install-time 'hooks required' flag
    000245     000          .BYTE                           ;reserved
```

Duplicate code from .DRBEG to restore pointer to vector table when SET XL NOMTTY is issued:

```
    000246  000000C VECSAV: .WORD   100000+<<XL$VTB-H1.VEC>/2-1> ; : Vector info for SET NOMTTY
    000250  176500  CSRSAV: .WORD   XL$CSR                  ; : CSR info for SET NOMTTY
                            .ENDC ;NE XL$MTY

    000252                  .Assume . LE 400 MESSAGE=<Installation code too large>
```

SET OPTION PARAMETER TABLE

```
                            .SBTTL  SET OPTION PARAMETER TABLE

                    ;       Option  Data                    Routine Syntax
                    ;       ------  ----                    ------- ------

                            .IF EQ 1
                    .DRSET  BIT8    <^c177>                 O.BIT8  NO      ;[NO]BIT8
                            .ENDC ;EQ 1

                            .IF EQ XL$PC
    000252          .DRSET  CSR     160012                  O.CSR   OCT     ;CSR=n
    000400  160012          160012
    000402  012712          .RAD50  \CSR\
    000406     021          .BYTE   <O.CSR-^o400>/2.
    000407     140          .BYTE   ...V2
    000410  000000          .WORD   0
    000412          .DRSET  VECTOR  477                     O.VEC   OCT     ;VECTOR=n
    000410  000477          477
    000412  105113          .RAD50  \VECTOR\
    000414  077552
    000416     046          .BYTE   <O.VEC-^o400>/2.
    000417     140          .BYTE   ...V2
    000420  000000          .WORD   0
                            .ENDC ;EQ XL$PC

                            .IF EQ XL$PC
                             .IF NE XL$PDT
                    .DRSET  LIGHTS  -1                      O.LGHT  NO      ;[NO]LIGHTS
                             .ENDC ;NE XL$PDT
                            .ENDC ;EQ XL$PC

                            .IF NE XL$MTY
    000422          .DRSET  LINE    16.                     O.LINE  NUM     ;LINE=n
    000420  000020          16.
    000422  046166          .RAD50  \LINE\
    000424  017500
    000426     056          .BYTE   <O.LINE-^o400>/2.
    000427     100          .BYTE   ...V2
    000430  000000          .WORD   0
    000432          .DRSET  MTTY    -1                      O.MTTY  NO      ;[NO]MTTY
    000430  177777          -1
    000432  052164          .RAD50  \MTTY\
    000434  116100
    000436     063          .BYTE   <O.MTTY-^o400>/2.
    000437     200          .BYTE   ...V2
    000440  000000          .WORD   0
```

```
                        .ENDC ;NE XL$MTY

                        .IF NE <XL$PC ! XL$DVE ! XL$SBC>
                .DRSET  SPEED   NOP                     O.SPEE  NUM     ;SPEED=n
                        .ENDC ;NE <XL$PC ! XL$DVE ! XL$SBC>
```

SET OPTION PROCESSING ROUTINES

```
                        .SBTTL  SET OPTION PROCESSING ROUTINES

                        .IF EQ 1
                ; SET XL [NO]BIT8

                O.BIT8: CLRB    R3                      ;Ensure high bit is left alone
                        NOP                     ;placekeeper
                        MOV     R3,CHMASK               ;Save character alteration mask
                        RETURN
                        .ENDC ;EQ 1

                        .IF EQ XL$PC

                ; SET XL CSR=octal_address

```

When SET XL MTTY in effect, cannot alter install-time CSR (176); must save it for restore when SET XL NOMTTY issued:

```
000442          O.CSR:
                         .IF NE XL$MTY
000442  010067          MOV     R0,CSRSAV               ;Yes, update saved CSR for SET NOMTTY
        177602
000446  105767          TSTB    I$MTTY                  ;Are we set MTTY?
        177572
000452  001002          BNE     20$                     ;Yep, don't set install-time word
                         .ENDC ;NE XL$MTY

000454  010067  10$:    MOV     R0,INSCSR               ;Let installation code know
        177516
000460  010067  20$:    MOV     R0,DISCSR               ;Fill in display CSR
        177510
000464                  .ADDR   #XIS,R1                 ;R1 -> Where to put CSR info
000464  010701          MOV     PC,R1
000466  062701          ADD     #XIS-.,R1
        177444'
000472  012702          MOV     #4,R2                   ;R2 = Count of words to set
        000004
000476  010021  30$:    MOV     R0,(R1)+                ;Set a table entry
000500  062700          ADD     #2,R0                   ;Prepare for next entry
        000002
000504  005302          DEC     R2                      ;More to do?
000506  003373          BGT     30$                     ;Yep...
000510  020003          CMP     R0,R3                   ;Was address specified in range?
000512  000207          RETURN                          ; c-bit=0 if so, =1 if not

                ; SET XL VECTOR=octal_address

000514  010067  O.VEC:  MOV     R0,XL$VTB               ;Save the new input interrupt vector
        000142'
000520  062700          ADD     #4,R0
        000004
000524  010067          MOV     R0,XL$VTB+6             ; and output interrupt vector
        000150'
000530  020300          CMP     R3,R0                   ;Was address specified in range?
000532  000207          RETURN                          ; c-bit=0 if so, =1 if not

                         .IF NE XL$PDT

                ; SET XL [NO]LIGHTS

                O.LGHT: CLR     R3                      ;LIGHTS entry point
                        NOP                             ; (padding)
                        COM     R3                      ;NOLIGHTS entry point
                        MOV     R3,LitFlg               ;Set/Reset lights flag
                        BR      O.NOR
                         .ENDC ;NE XL$PDT
```

```
                        .IF NE XL$MTY

               ; SET XL LINE=line_number

000534  120003  O.LINE: CMPB    R0,R3                   ;Is line number valid?
000536  101027          BHI     O.ERR                   ;Nope...
000540  110067          MOVB    R0,O$LINE               ;Yes, set line number to use
        000511'
000544  000423          BR      O.NOR

               ; SET XL [NO]MTTY

000546  000411  O.MTTY: BR      10$                     ;Entry point for MTTY
000550  000240          NOP                     ;placekeeper
000552  005000          CLR     R0                      ;Entry point for NOMTTY
000554  016767          MOV     CSRSAV,INSCSR           ;Nope, restore install-time CSR
        177470
        177414
000562  016767          MOV     VECSAV,H1.VEC           ; and vector information
        177460
        000210
000570  000404          BR      20$

000572  005067  10$:    CLR     INSCSR                  ;Reset install-time CSR and
        177400
000576  005067          CLR     H1.VEC                  ; vector so handler installs
        000176
000602  110067  20$:    MOVB    R0,O$MTTY               ;Set/Reset MTTY hooks use flag
        000510'
000606  110067          MOVB    R0,I$MTTY               ; and inform install code of setting
        177432
000612  000400          BR      O.NOR
                         .ENDC ;NE XL$MTY
                        .ENDC ;EQ XL$PC

                        .IF NE <XL$PC ! XL$DVE ! XL$SBC>

               ; SET XL SPEED=decimal_speed
```

Setting speed alters the on-disk image, but also takes immediate effect:

```

                O.SPEE:
```

Can’t use when MTTY is in effect because not all lines have programmable baud rate:

```
                         .IF NE XL$MTY
                        TSTB    I$MTTY                  ;Handler hooks in use?
                        BNE     O.ERR                   ;Yes, can’t touch the CSR
                         .ENDC ;NE XL$MTY

                        .ADDR   #SPEEDT,R1              ;R1 -> Baud rate table
                10$:    TST     @R1                     ;End of table?
                        BEQ     O.ERR                   ;Yes, speed requested is invalid
                        CMP     R0,(R1)+                ;Nope, request match this entry?
                        BNE     10$                     ;Nope, try another speed entry
                        SUB     PC,R1                   ;Yes, determine speed mask
                        SUB     #<SPEEDT+2-.>,R1        ; ...

                         .IF NE XL$PC
                        ASR     R1                      ;Convert from byte to word offset
                        MOVB    R1,-(SP)                ;Save the receive speed mask
                        ASL     R1                      ;And make transmit speed match
                        ASL     R1                      ; by shifting
                        ASL     R1                      ;  it to the
                        ASL     R1                      ;   high nibble
                        BISB    (SP)+,R1                ;OR in the receive speed mask
                        MOVB    R1,@#XL$BAU             ; and change the speed now
                         .ENDC ;NE XL$PC

                         .IF NE XL$DVE
                        SWAB    R1                      ;Move to high byte
                        ASL     R1                      ; then shift mask to where
                        ASL     R1                      ; it should be for
```

```
                          ASL     R1                      ; a DLV11-E
                           .ENDC ;NE XL$PC
                           .IF NE XL$SBC
                          ASL     R1                      ;Shift mask to where it
                          ASL     R1                      ; should be for SBC or MXV SLU
                           .ENDC ;NE XL$SBC
                           .IF NE <XL$DVE ! XL$SBC>
                          BIS     #XC.SCE,R1              ;Set the 'speed change enable' bit
                          MOV     INSCSR,R0               ;R0->Receiver CSR
                           .ENDC ;NE <XL$DVE ! XL$SBC>
                          MOV     R1,ISPEED               ;Save new speed for installation
                           .IF NE <XL$DVE ! XL$SBC>
                          MOV     ISPEED,4(R0)            ;Set the speed (in transmitter CSR)
                           .ENDC ;NE <XL$DVE ! XL$SBC>
                          .BR     O.NOR
                          .ENDC ;NE <XL$PC ! XL$DVE ! XL$SBC>
 000614  005727  O.NOR:   TST     (PC)+                   ;Success (carry=0)
 000616  000261  O.ERR:   SEC                             ;Failure (carry=1)
 000620  000207           RETURN
 000622                   .Assume . LE 1000 MESSAGE=<Set code too large>
```

DRIVER ENTRY

```
                          .SBTTL  DRIVER ENTRY
                  ; The handler gets entered here each time the monitor places a new
                  ; request on the device queue.  The handler either processes the
                  ; request immediately and returns it to the monitor or the request
                  ; is removed from the device queue and placed on one of the internal
                  ; queues.  There is one internal queue for input and one for output.
                  ;
                  ; Because of the separate queues, simultaneous input and output may
                  ; be performed.
                          .ENABL  LSB
                          .IF EQ XL$MTY
                          .DRBEG  XL
```

Following code is for hooks support. Ensures vector word is zero so handler loads without affecting any vectors when XL is SET MTTY. Restored with SET XL NOMTTY.

```
                          .IFF ;EQ XL$MTY
 000622                   .DRBEG  XL,0                    ;Default to use handler hooks
 000052  002704           .WORD   <XLEND-XLSTRT>
 000054  000000           .WORD   XLDSIZE
 000056  007057           .WORD   XLSTS
 000060  000006           .WORD   ^o<ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>>
 000000  000000           .WORD   0&^C3.
 000002  001120           .WORD   XLINT-.,^o340
 000004  000340
 000006  000000  XLLQE:: .WORD    0
 000010  000000  XLCQE:: .WORD    0
 000012  000257                   .WORD   257
                          .ENDC ;EQ XL$MTY
 000014  016704           MOV     XLCQE,R4                ;R4->Current queue element
         177770
         000022' STATFG = <. + 2>
 000020  006227           ASR     #1                      ;First call since .FETCH/LOAD or
         000001
                                                          ; last shutdown?
 000024  103013           BCC     40$                     ;Nope...
                          .IF EQ XL$PC
 000026  004767           CALL    ENAINI                  ;Turn on receiver interrupts
         002166
```

```
000032  012700          MOV     #<RC.RTS!RC.DTR>,R0       ;Assert DTR
        000006
000036  004767          CALL    SETSTT                    ; ...
        002412
000042  012767          MOV     #-2,SNDS                  ;Indicate we must send an XON
        177776
        001070
000050  004767          CALL    ENAOUI                    ;Enable output interrupts
        002216
                         .IF NE XL$PDT
                        CALL    SETLIT                    ;Set the lights to indicate state
                         .ENDC ;NE XL$PDT
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R1,@CSRA             ;Select csr A, write register 1
                        BIS     #<W1.RIE!W1.TIE>,SSRAW1 ;Turn on RECV and XMIT interrupts
                        MOV     SSRAW1,@CSRA              ; (update from software register)
                        BIS     #<M0.DTR!M0.RTS>,@MCR0  ;Force DTR and RTS
                        MOVB    #C.CTLQ,@DBUF             ;First thing we send is an XON
                        .ENDC ;EQ XL$PC

000054  116405  40$:    MOVB    Q$FUNC(R4),R5             ;Get the function code
        000002
000060  001040          BNE     SPFUN                     ;If non-zero, we have a .SPFUN
000062  006364          ASL     Q$WCNT(R4)                ;Convert word count to byte count
        000006
000066  103406          BCS     WRITE                     ;If negative, write request
                                                          ; otherwise, read
000070  004567  READ:   JSR     R5,XLENQ                  ;Queue the read request
        002002
```

Internal input queue:

```
000074  000000  XICQE:   .WORD  0                         ; : address of first element on queue
000076  000000  XILQE:   .WORD  0                         ; : address of last element on queue
000100  000167          CALLR   XIIN                      ;Process any input already received,
        001436
                                                          ; read will be completed via
                                                          ; interrupts

000104  005267  WRITE:  INC     QCHG                      ;Set 'queue being modified' flag
        001066
000110  004567          JSR     R5,XLENQ                  ;Queue the write request
        001762
```

Internal output queue:

```
000114  000000  XOCQE:   .WORD  0                         ; : address of first element on queue
000116  000000  XOLQE:   .WORD  0                         ; : address of last element on queue
000120  005067          CLR     QCHG                      ;Reset 'queue being modified' flag
        001052

                        .IF EQ XL$PC
000124  004767          CALL    ENAOUI                    ;Enable output interrupts
        002142
                        .IFF ;EQ XL$PC
                        CALL    GNXTCH                    ;Get a character for output
                        BEQ     50$                       ;None available...
                        MOVB    R5,@DBUF                  ;Now prime the interrupt pump
                        .ENDC ;EQ XL$PC

000130  000207  50$:    RETURN

                        .DSABL  LSB
```

REGISTERS AND VECTOR TABLES

```
                                .SBTTL  REGISTERS AND VECTOR TABLES

                                .IF EQ XL$PC
                        ; *** Begin Critical Ordering ***
000132  176500  XIS:    .WORD   XL$CSR                  ; : Receiver status register
000134  176502  XIB:    .WORD   XL$CSR+2                ; : Receiver buffer register
000136  176504  XOS:    .WORD   XL$CSR+4                ; : Transmitter status register
000140  176506  XOB:    .WORD   XL$CSR+6                ; : Transmitter buffer register
                        ; *** End Critical Ordering ***
                                .IFF ;EQ XL$PC
                DBUF:   .WORD   XL$BUF                  ; : Input/Output buffer register
                CSRA:   .WORD   XL$CSA                  ; : Control/Status register A
                CSRB:   .WORD   XL$CSB                  ; : Control/Status register B
                MCR0:   .WORD   XL$MC0                  ; : Modem control/status register 0
                MCR1:   .WORD   XL$MC1                  ; : Modem control/status register 1
                BAUD:   .WORD   XL$BAU                  ; : Baud rate control register
                        .ENDC ;EQ XL$PC

                ; Now for some software registers

                        .IF NE XL$PC
                SSRAW1: .WORD   0                       ;Software status A, write register 1
                SSRAW5: .WORD   <W5.TXE!TCL.8>          ;Software status A, write register 5
                        .ENDC ;NE XL$PC

                ; Define the interrupt vectors

                        .IF EQ XL$PC
000142                  .DRVTB  XL,XL$VEC,XIINT         ;Input interrupt servicer
000142  000300          .WORD   XL$VEC&^C3.,XIINT-.,^o340!0,^o100000
000144  001214
000146  000340
000150  100000
000152                  .DRVTB  ,XL$VEC+4,XLINT         ;Output interrupt servicer
000150  000304          .WORD   XL$VEC+4&^C3.,XLINT-.,^o340!0,^o100000
000152  000750
000154  000340
000156  100000
                        .IFF ;EQ XL$PC
                        .DRVTB  XL,XL$VEC,XLINT         ;Input/Output interrupt servicer
                        .DRVTB  ,XL$VEC+4,XLINT
                        .ENDC ;EQ XL$PC

000160  177600  CHMASK: .WORD   ^C177                   ;Character mask

                        .IF EQ XL$PC
                         .IF NE XL$PDT
```

LIGHTS ROUTINE FOR PDT-11'S

```
                        .SBTTL  LIGHTS ROUTINE FOR PDT-11'S

                ;+
                ;
                ; Sets PDT lights to indicate XON/XOFF state.
                ;
                ;       LED 1 on if PDT has sent XOFF
                ;       LED 2 on if PDT has received XOFF
                ;
                ;-

                SETLIT: TST     (PC)+                   ;SET XL LIGHTS in effect?
                LITFLG:  .WORD  0                       ; : lights flag (0 = no, <>0 = yes)
                        BEQ     30$                     ;Nope...
                        MOV     #040000,R5              ;Default to lights off
                        TST     SNDS                    ;XOFF sent to host?
                        BLE     10$                     ;Nope...
                        BIS     #000100,R5              ;Yes, turn on LED 1
                10$:    TST     RECS                    ;XOFF received from host?
                        BEQ     20$                     ;Nope...
                        BIS     #000200,R5              ;Yes, turn on LED 2
                20$:    MOV     R5,@#177420             ;Force the new lights setting
                30$:    RETURN

                         .ENDC ;NE XL$PDT
                        .ENDC ;EQ XL$PC
```

SPFUN PROCESSING

```
                        .SBTTL  SPFUN PROCESSING

                ; This section of code gets jumped to. It expects that the address of the
                ; queue element is is R4 and the address of the special function code to
                ; be executed is in R5.

```

Special read may require post-interrupt processing, so it must be internally queued:

```
000162  120527  SPFUN:  CMPB    R5,#SRDDRV              ;Special read request?
        000203
000166  001740          BEQ     READ                    ; Yes, go queue it
000170  120527          CMPB    R5,#BRKDRV              ;[end]BREAK request?
        000202
000174  001423          BEQ     20$                     ; Yes...
000176  120527          CMPB    R5,#CLRDRV              ;Clear driver flags request?
        000201
000202  001440          BEQ     40$                     ; Yes...
000204  120527          CMPB    R5,#STSDRV              ;Status request?
        000204
000210  001445          BEQ     50$                     ; Yes...
000212  120527          CMPB    R5,#OFFDRV              ;Shutting us down?
        000205
000216  001502          BEQ     100$                    ;Yes...
000220  120527          CMPB    R5,#DTRDRV              ;DTR set/reset?
        000206
000224  001514          BEQ     110$                    ;Yes...
                                                        ;Unknown .SPFUN, ignore
000226          10$:    .DRFIN  XL                      ;Inform monitor of completion
```

SPFUN routines can be processed without post-interrupt processing, so they are handled without being moved to internal queue and returned to RT–11:

```
000226  010704          MOV     PC,R4
000230  062704          ADD     #XLCQE-.,R4
        177560
000234  013705          MOV     @#^o54,R5
        000054
000240  000175          JMP     @^o270(R5)
        000270

                ; [end]BREAK processing
                ;       Word count indicates operation
                ;       (0 = end break, non-zero = break)

000244  005764  20$:    TST     Q$WCNT(R4)              ;Break or end-break?
        000006
000250  001406          BEQ     30$                     ;If zero, end-break...
000252  012767          MOV     #1,BRKFLG               ;Break, set 'break in progress' flag
        000001
        000652

                        .IF EQ XL$PC
000260  004767          CALL    SETBRK                  ;Turn on break
        002064
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R5,@CSRA           ;Select csr A, write register 5
                        BIS     #W5.SB,SSRAW5           ;Turn on break
                        MOV     SSRAW5,@CSRA            ; (update from software register)
                        .ENDC ;EQ XL$PC

000264  000760          BR      10$

000266          30$:
                        .IF EQ XL$PC
000266  004767          CALL    RESBRK                  ;Turn off break
        002030
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R5,@CSRA           ;Select csr A, write register 5
                        BIC     #W5.SB,SSRAW5           ;Turn off break
                        MOV     SSRAW5,@CSRA            ; (update from software register)
                        .ENDC ;EQ XL$PC
```

```
   000272  005067          CLR     BRKFLG                  ;Reset the 'break in progress' flag
           000634
                           .IF EQ XL$PC
   000276  004767          CALL    ENAOUI                  ;Make sure output is running
           001770
                           .ENDC ;EQ XL$PC
   000302  000751          BR      10$
                   ; Clear driver flags request
                   ;       resets received XOFF flag
                   ;       sends XON to host
   000304  005067  40$:    CLR     RECS                    ;Reset the 'received XOFF' flag
           000660
                           .IF EQ XL$PC
   000310  012767          MOV     #-2,SNDS                ;Indicate we want an XON sent
           177776
           000622
   000316  004767          CALL    ENAOUI                  ;Make sure output is running
           001750
                            .IF NE XL$PDT
                           CALL    SETLIT                  ;Update lights display
                            .ENDC ;NE XL$PDT
                           .IFF ;EQ XL$PC
                           CLR     SNDS                    ;Indicate that an XON has been
                           MOVB    #C.CTLQ,@DBUF           ; sent
                           .ENDC ;EQ XL$PC
   000322  000741          BR      10$
                   ; Get Status request
                   ;       returns handler version in high byte
                   ;       returns XON/XOFF state in low byte
                   ;         bit 0 on if host has been XOFF'd
                   ;         bit 1 on if host has XOFF'd us
                   ;         bit 2 on if CTS is asserted
                   ;         bit 3 on if CD is asserted
                   ;         bit 4 on if RI is asserted
   000324  012705  50$:    MOV     #$$$VER*400,R5          ;High byte = handler version
           011000
   000330  005767          TST     SNDS                    ;Have we XOFF'd host?
           000604
   000334  003401          BLE     60$                     ;Nope...
   000336                  .ASSUME ST.XFH EQ 1
   000336  005205          INC     R5                      ;Yes, set the indicator
   000340  005767  60$:    TST     RECS                    ;Have we been XOFF'd?
           000624
   000344  001402          BEQ     70$                     ;Nope...
   000346  052705          BIS     #ST.XOF,R5              ;Yes, set the indicator
           000002
   000352          70$:
                           .IF EQ XL$PC
   000352  004767          CALL    GETSTT                  ;Get current status
           002022
   000356  032700          BIT     #RC.CTS,R0              ;Is 'Clear To Send' asserted?
           020000
                           .IFF ;EQ XL$PC
                           BIT     #M1.CTS,@MCR1           ;Is 'Clear To Send' asserted?
                           .ENDC ;EQ XL$PC
   000362  001402          BEQ     80$                     ;Nope...
   000364  052705          BIS     #ST.CTS,R5              ;Yes, set an indicator
           000004
   000370          80$:
                           .IF EQ XL$PC
   000370  032700          BIT     #RC.CD,R0               ;Is 'Carrier Detect' asserted?
           010000
                           .IFF ;EQ XL$PC
                           BIT     #M1.CD,@MCR1            ;Is 'Carrier Detect' asserted?
                           .ENDC ;EQ XL$PC
```

```
000374  001402          BEQ     82$                     ;Nope...
000376  052705          BIS     #ST.CD,R5               ;Yes, set an indicator
        000010

000402          82$:
                        .IF EQ XL$PC
000402  032700          BIT     #RC.RI,R0               ;Is 'Ring Indicator' asserted?
        040000
                        .IFF ;EQ XL$PC
                        BIT     #M1.RI,@MCR1            ;Is 'Ring Indicator' asserted?
                        .ENDC ;EQ XL$PC

000406  001402          BEQ     84$                     ;Nope...
000410  052705          BIS     #ST.RI,R5               ;Yes, set an indicator
        000020

000414          84$:
                        .IF EQ MMG$T
                        MOV     R5,@Q$BUFF(R4)          ;Return the status word
                        .IFF ;EQ MMG$T
000414  010546          MOV     R5,-(SP)                ;Return the status word
000416  004777          CALL    @$PTWRD                 ; ...
        002252
                        .ENDC ;EQ MMG$T

000422  000701          BR      10$

                ; Shut down driver request (OFFDRV)
                ;       Sets a flag such that when VTCOM exits, interrupts will
                ;       not be re-enabled. STATFG is used as the once-only,
                ;       interrupt startup flag.

000424  116446  100$:   MOVB    Q$JNUM(R4),-(SP)        ;Save Q$JNUM
        000003
000430  042716          BIC     #^C<JOBMK>,@SP          ;Isolate job number issuing request
        177407
000434  006216          ASR     @SP                     ;Shift for abort code check
000436  006216          ASR     @SP
000440  006216          ASR     @SP
000442  112667          MOVB    (SP)+,JNUM              ;Save it for later check
        000040
000446  012767          MOV     #1,STATFG               ;Reset us to pre-start state
        000001
        177346
000454  000664          BR      10$

                ; Set/Reset DTR (DTRDRV)
                ;       Sets or resets DTR based on word count
                ;       (0 = DTR off, <>0 = DTR on)

000456          110$:
                        .IF EQ XL$PC
000456  004767          CALL    GETSTT                  ;Get current state
        001716
000462  042700          BIC     #<RC.RTS!RC.DTR>,R0     ;Assume DTR is desired off
        000006
                        .IFF ;EQ XL$PC
                        MOVB    @MCR0,R0                ;Get current state
                        BIC     #<M0.DTR!M0.RTS>,R0     ;Assume DTR is desired off
                        .ENDC ;EQ XL$PC

000466  005764          TST     Q$WCNT(R4)              ;Correct assumption?
        000006
000472  001402          BEQ     115$                    ;Yep...

                        .IF EQ XL$PC
000474  052700          BIS     #<RC.RTS!RC.DTR>,R0     ;Nope, turn it on
        000006
                        .IFF ;EQ XL$PC
                        BIS     #<M0.DTR!M0.RTS>,R0     ;Nope, turn it on
                        .ENDC ;EQ XL$PC

000500          115$:
                        .IF EQ XL$PC
000500  004767          CALL    SETSTT                  ;Assert desired bits
        001750
                        .IFF ;EQ XL$PC
                        MOVB    R0,@MCR0                ;Set desired state
```

```
                            .ENDC ;EQ XL$PC

    000504  000650          BR      10$

    000506          JNUM:   .BLKW                           ; :Job number which issued OFFDRV

                            .IF NE XL$PC
```

INTERRUPT SERVICE/DISPATCHER

```
                            .SBTTL  INTERRUPT SERVICE/DISPATCHER

                    ;+
                    ;
                    ; Interrupt entry point for input and output interrupts. The interrupt
                    ; type is determined by bits <04:02> in RR2 of CSR B. The four defined
                    ; types of interrupts are:
                    ;
                    ;       1) Transmitter buffer empty     (^B100xx)
                    ;       2) External/status change       (^B101xx)
                    ;       3) Received character available (^B110xx)
                    ;       4) Special receiver condition   (^B111xx)
                    ;
                    ;-

                            .DRAST  XL,4,XLDONE

                            MOV     #RPT.R2,@CSRB           ;Select csr B, read register 2
                            MOV     @CSRB,-(SP)             ;Get the interrupt type
                            BIC     #^C<R2.IMK>,@SP         ;Strip the uninteresting stuff
                            ASR     @SP                     ;Shift for word table offset
                            .ADDR   #INTTAB,@SP,ADD         ;Add address of start of table
                            MOV     @(SP),@SP               ;Get the table entry
                            ADD     PC,@SP                  ;Convert to address
                    INTDSP: JMP     @(SP)+                  ;Dispatch the interrupt

                    ESINT:  MOV     #CMD.RE,@CSRA           ;Reset external/status interrupts
                    IECOM:  MOV     #CMD.EI,@CSRA           ;Declare end of interrupt
                            RETURN

                    SRINT:  MOV     #CMD.ER,@CSRA           ;Reset error latches
                            JMP     XIINT                   ; then handle as received character

                    INTTAB: .WORD   IECOM-INTDSP            ;unknown interrupt
                            .WORD   IECOM-INTDSP            ;unknown interrupt
                            .WORD   IECOM-INTDSP            ;unknown interrupt
                            .WORD   IECOM-INTDSP            ;unknown interrupt
                            .WORD   XOINT-INTDSP            ;Transmitter buffer empty
                            .WORD   ESINT-INTDSP            ;External/Status change
                            .WORD   XIINT-INTDSP            ;Received character available
                            .WORD   SRINT-INTDSP            ;Special receiver interrupt
                            .ENDC ;NE XL$PC

                            .IF NE XL$MTY
```

MULTITERMINAL HANDLER HOOKS SUPPORT DATA

```
                            .SBTTL  Multiterminal Handler Hooks Support Data

                    ; The following byte indicates whether the handler should make use
                    ; of the multiterminal hooks during FETCH/LOAD and during operation.

                    ; *** SET ***
```

Set/reset by SET XL [NO]MTTY:

```
    000510     377  O$MTTY: .BYTE   -1                      ;Default to hooks used
    000511          .Assume <O$MTTY-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>

                    ; *** SET ***
```

Set/reset by SET XL LINE=n:

```
   000511     001  O$LINE: .BYTE   XL$LUN                 ;Default line to use
   000512          .Assume <O$LINE-XLSTRT> LE 1000 MESSAGE=<Code to set not in block 1>

   000512     377  ISPND:  .BYTE   -1                     ; : Input suspend flag
   000513     377  OSPND:  .BYTE   -1                     ; : Output suspend flag
                           .EVEN
```

XLHOOK - Multiterminal Handler Hooks Hook Routine

```
                           .SBTTL  XLHOOK  - Multiterminal Handler Hooks Hook Routine

                   ;+
                   ;
                   ; XLHOOK
                   ;       Entered from multiterminal input or output interrupt service.
                   ;
                   ; Call (TH.GOC):
                   ;       R0 = Function code
                   ;
                   ; Return (TH.GOC):
                   ;       PSW<c> = 0, R5 = character
                   ;       PSW<c> = 1, no character available
                   ;
                   ; Call (TH.PIC):
                   ;       R0 = Function code
                   ;       R5 = character
                   ;
                   ;-

                           .ENABL  LSB

```

The following line must reside before hook entry point:

```
   000514  113740  XLPNAM: .Rad50  /XL/                   ; : Rad50 physical name
                                                          ; loaded by FETCH/LOAD code

   000516          XLHOOK:
   000516                  .Assume <XLHOOK-XLPNAM> EQ 2 MESSAGE=<XLPNAM must preceed XLHOOK>

   000516  010446          MOV     R4,-(SP)               ;Save register for awhile

                   ; Function code = 1 = TH.GOC
                   ;       (Get Output Character)

   000520  020027          CMP     R0,#TH.GOC             ;'Get Output Character' request?
           000001
   000524  001006          BNE     10$                    ;Nope...
   000526  105767          TSTB    OSPND                  ;Is output suspended?
           177761
   000532  001014          BNE     20$                    ;Yep...
   000534  004767          CALL    HOINT                  ;Yes, hook handler output service
           000370
   000540  000412          BR      30$

                   ; Function code = 2 = TH.PIC
                   ;       (Put Input Character)

   000542  020027  10$:    CMP     R0,#TH.PIC             ;'Put Input Character' request?
           000002
   000546  001006          BNE     20$                    ;Nope...
   000550  105767          TSTB    ISPND                  ;Is input suspended?
           177736
   000554  001003          BNE     20$                    ;Yep...
   000556  004767          CALL    HIINT                  ;Yes, hook handler input service
           000604
   000562  000401          BR      30$

   000564  000261  20$:    SEC                            ;Return failure
   000566  012604  30$:    MOV     (SP)+,R4               ;*C* Restore previously saved register
   000570  000207          RETURN

                           .DSABL  LSB
```

PREMTY - Prepare for multiterminal hook

```
                           .SBTTL  PREMTY  - Prepare for multiterminal hook

                   ;+
                   ;
                   ; PREMTY
                   ;       Prepares for use of a multiterminal hook.
                   ;
                   ; Return:
                   ;       R3 -> TCB
                   ;
                   ; Note:
                   ;       *** Co-routine ***
                   ;       Saves R3
                   ;
                   ;-

000572  105767  PREMTY: TSTB    O$MTTY                  ;Terminal hooks in use?
        177712
000576  001410          BEQ     10$                     ;Nope...
000600  010346          MOV     R3,-(SP)                ;Save some registers for awhile
000602  016703          MOV     TCBADX,R3               ;R3 -> TCB hooked to us
        002046
000606  016646          MOV     2(SP),-(SP)             ;Restack the return address
        000002
000612  004736          CALL    @(SP)+                  ;Co-routine back to caller
000614  012603          MOV     (SP)+,R3                ;Restore previously saved register
000616  005726          TST     (SP)+                   ;Discard old return address
                                                        ; to return to callers caller
000620  000207  10$:    RETURN

                        .ENDC ;NE XL$MTY
```

DRIVER RESET ENTRY

```
                        .SBTTL  DRIVER RESET ENTRY

                ;+
                ;
                ; This routine is entered on the abort of a job or an HRESET. It
                ; deques and tells RT that all I/O requests by a job are done. It
                ; expects to be entered with the number of the aborting job in R4.
                ;
                ; Entered with:
                ;       R4 =  Job number {aborting | issuing .ABTIO}
                ;       R5 =  0 if abort by job
                ;          -> Channel Control Block (CCB) if abort by channel (.ABTIO)
                ;
                ;-

                        .ENABL  LSB

000622  010046  XLDONE: MOV     R0,-(SP)                ;Save R0 for awhile

                        .IF EQ XL$PC
000624  004767          CALL    DISINI                  ;Turn off input interrupts
        001342
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R1,@CSRA           ;Select csr A, write register 1
                        BIC     #W1.RIE,SSRAW1          ;Turn off input interrupts
                        MOV     SSRAW1,@CSRA            ; (update from software register)
                        .ENDC ;EQ XL$PC

000630  004467          JSR     R4,50$                  ; while we remove entries from the
        000110
000634  177124           .WORD  XICQE-60$-Q$LINK        ; input queue

000636  120467          CMPB    R4,JNUM                 ;Is aborting job same as one which
        177644
                                                        ; issued OFFDRV call?
000642  001003          BNE     5$                      ;No, so interrupts should still be on
000644  005767          TST     STATFG                  ;Should we turn interrupts back on?
        177152
000650  001002          BNE     10$                     ;Nope...
```

```
000652          5$:
                        .IF EQ XL$PC
000652  004767          CALL    ENAINI                  ;Turn input interrupts back on
        001342
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R1,@CSRA           ;Select csr A, write register 1
                        BIS     #W1.RIE,SSRAW1          ;Turn input interrupts back on
                        MOV     SSRAW1,@CSRA            ; (update from software register)
                        .ENDC ;EQ XL$PC
000656  005267  10$:    INC     QCHG                    ;Set the 'queue being modified' flag
        000314
000662  004467          JSR     R4,50$                  ; while we remove entries from the
        000056
000666  177144           .WORD  XOCQE-60$-Q$LINK        ; output queue
000670  005067          CLR     QCHG                    ;Reset the 'queue being modified' flag
        000302
000674  120467          CMPB    R4,JNUM                 ;Is aborting job same as one which
        177606
                                                        ;issued OFFDRV call?
000700  001003          BNE     15$                     ;No, so interrupts should still be on
000702  005767          TST     STATFG                  ;Again, interrupts back on?
        177114
000706  001002          BNE     30$                     ;Nope...
000710          15$:
                        .IF EQ XL$PC
000710  004767          CALL    ENAOUI                  ;Turn output interrupts back on
        001356
                        .IFF ;EQ XL$PC
                        MOV     R5,-(SP)                ;Save R5 for awhile
                        CALL    GNXTCH                  ;Get a character for output
                        BEQ     20$                     ;None available...
                        MOVB    R5,@DBUF                ;Now prime the interrupt pump
                20$:    MOV     (SP)+,R5                ;Restore R5
                        .ENDC ;EQ XL$PC
000714  012600  30$:    MOV     (SP)+,R0                ;Restore R0
000716  005767          TST     XLCQE                   ;Anything to return to RT?
        177066
000722  001001          BNE     40$                     ;Yes...
```

Use RETURN if no internally-queued elements are being aborted:

```
000724  000207          RETURN                          ;Nope, just return
```

Use .DRFIN if any abortable queue elements have been placed on the device queue.

NOTE

Only abortable queue elements should be placed on the device queue.

```
000726          40$:    .DRFIN  XL
000726  010704          MOV     PC,R4
000730  062704          ADD     #XLCQE-.,R4
        177060
000734  013705          MOV     @#^o54,R5
        000054
000740  000175          JMP     @^o270(R5)
        000270

                ; The following code scans the internal queue for queue elements which
                ; match the abort criteria (job number for job abort, channel if abort
                ; by channel).  It then dequeues them from the internal queue, returning
                ; them to the device queue.
```

Internal queuing code. Used to remove abortable queue elements from internal queues:

```
          000004          SP.CCB  = 4                     ;Stacked CCB pointer
          000006          SP.JOB  = 6                     ;Stacked job number
   000744 010546  50$:    MOV     R5,-(SP)                ;Save CCB pointer
   000746 012405          MOV     (R4)+,R5                ;Pick up the displacement and
   000750 010446          MOV     R4,-(SP)                ; store the return address
   000752 060705          ADD     PC,R5                   ;Calculate actual address
                                                          ; (60$ must follow this)
   000754 010546  60$:    MOV     R5,-(SP)                ;Save the Q header address
   000756 016504  70$:    MOV     Q$LINK(R5),R4           ;Link to the next entry
          177774
   000762 001450          BEQ     120$                    ;If zero, no more
   000764 005766          TST     SP.CCB(SP)              ;Abort by channel (.ABTIO) ?
          000004
   000770 001405          BEQ     80$                     ;Nope, aborting job...
   000772 026466          CMP     Q$CSW(R4),SP.CCB(SP)    ;Yes, this qelement for that channel?
          177776
          000004
   001000 001037          BNE     110$                    ;Nope...
   001002 000412          BR      90$                     ;Yes, go remove it
   001004 116400  80$:    MOVB    Q$JNUM(R4),R0           ;Get number of job being aborted
          000003
   001010 006200          ASR     R0                      ; and
   001012 006200          ASR     R0                      ;  shift
   001014 006200          ASR     R0                      ;   to
   001016 042700          BIC     #^C<37>,R0              ;    isolate job bits
          177740
   001022 020066          CMP     R0,SP.JOB(SP)           ;Job own this queue element?
          000006
   001026 001024          BNE     110$                    ;Nope...
   001030 016465  90$:    MOV     Q$LINK(R4),Q$LINK(R5)   ;Yes, unlink it from the list
          177774
          177774
   001036 005064          CLR     Q$LINK(R4)              ;Make sure it doesn't link anywhere
          177774
   001042 005767          TST     XLCQE                   ;Anything on the queue?
          176742
   001046 001005          BNE     100$                    ;Yes, then link it in at the end
   001050 010467          MOV     R4,XLCQE                ;Otherwise, make it the first
          176734
   001054 010467          MOV     R4,XLLQE                ; and only
          176726
   001060 000736          BR      70$                     ;Check for more elements to abort
   001062 016700  100$:   MOV     XLLQE,R0                ;R0->element at end of queue
          176720
   001066 010460          MOV     R4,Q$LINK(R0)           ;Link it to this new one
          177774
   001072 010467          MOV     R4,XLLQE                ; and make the new one last
          176710
   001076 000727          BR      70$                     ;Check for more elements to abort

                  ; Here if element is not part of the aborting job

   001100 010405  110$:   MOV     R4,R5                   ;Skip this element
   001102 000725          BR      70$                     ;Check for more elements to abort

                  ; DeQueue is done, record the new end of the queue

   001104 012604  120$:   MOV     (SP)+,R4                ;R4->Queue header
   001106 010564          MOV     R5,Q$LINK+2(R4)         ;Set the new end of queue
          177776
   001112 012604          MOV     (SP)+,R4                ;Recover the return address
   001114 012605          MOV     (SP)+,R5                ;Restore CCB pointer
   001116 000204          RTS     R4

                          .DSABL  LSB
```

OUTPUT INTERRUPT SERVICER

```
                        .SBTTL  OUTPUT INTERRUPT SERVICER

                        .IF EQ XL$PC
  001120                .DRAST  XL,XL$PRI,XLDONE
  001120  000640        BR      XLDONE
  001122  004577  XLINT::JSR    R5,@$INPTR
          001552
  001126  000140        .WORD   ^C<XL$PRI*^o40>&^o340
                        .IFF ;EQ XL$PC
                  XOINT:
                        .ENDC ;EQ XL$PC

                        .ENABL  LSB

```

Hook output interrupt entry point:

```
  001130          HOINT:        ;Output interrupt hook point

  001130  005727        TST     (PC)+                   ;Is break in progress?
  001132  000000  BRKFLG:  .WORD 0                      ; : 'break in progress' flag (0=no)
  001134  001030        BNE     30$                     ;Yes, then don't do any output
  001136  005727        TST     (PC)+                   ;Need to send an XON or XOFF?
  001140  000000  SNDS:    .WORD 0                      ; : send XON/XOFF flag
                                                        ; -2 = XON should be sent
                                                        ; -1 = XOFF should be sent
                                                        ; 0  = XON has been sent
                                                        ; 1  = XOFF has been sent
  001142  100011        BPL     10$                     ;Neither...
  001144  112705        MOVB    #C.CTLQ,R5              ;Assume we are to send an XON
          000021
  001150  062767        ADD     #2,SNDS                 ;Are we correct? (SNDS = 0 if yes)
          000002
          177762
  001156  001414        BEQ     20$                     ;Yes, go send it
  001160  112705        MOVB    #C.CTLS,R5              ;No, we must send an XOFF
          000023
  001164  000411        BR      20$                     ;Now go send it

  001166  005727  10$:  TST     (PC)+                   ;Have we been XOFF'd?
  001170  000000  RECS:    .WORD 0                      ; : received XOFF flag
  001172  001011        BNE     30$                     ;Yes, then don't do any output
  001174  005727        TST     (PC)+                   ;No, are output queues being modified?
  001176  000000  QCHG:    .WORD 0                      ; : 'queues being modified' flag
  001200  001006        BNE     30$                     ;Yes, then don't do any output
  001202  004767        CALL    GNXTCH                  ;Go get a character to output
          000024
  001206  001403        BEQ     30$                     ;None available...

  001210          20$:
  001210  004767        CALL    PUTC                    ;Output the character
          001276

                        .IF EQ XL$PC
                         .IF NE XL$PDT
                        CALL    SETLIT                  ;Update the PDT lights display
                         .ENDC ;NE XL$PDT
  001214  000403        BR      40$
                        .ENDC ;EQ XL$PC

  001216          30$:
                        .IF EQ XL$PC
  001216  004767        CALL    DISOUI                  ;Turn off output interrupts
          001022
                        .IFF ;EQ XL$PC
                        MOV     #CMD.RT,@CSRA           ;Reset transmitter interrupt pending
                        MOV     #CMD.EI,@CSRA           ;Declare end of interrupt
                        .ENDC ;EQ XL$PC

  001222  000401        BR      50$

  001224  005727  40$:  TST     (PC)+
  001226  000261  50$:  SEC
  001230  000207  60$:  RETURN

                        .DSABL  LSB
```

GNXTCH - Get next output character

```
                        .SBTTL  GNXTCH  - Get next output character

                ;+
                ;
                ; GNXTCH
                ;       Obtains the next character from the output queue and returns
                ;       it in R5.
                ;
                ; CALL:
                ;       CALL    GNXTCH
                ;
                ; RETURNS:
                ;       z-bit = 0, R5 contains character to be output
                ;       z-bit = 1, no characters available to output
                ;
                ; NOTES:
                ;       As requests are completed, the associated queue elements are
                ;       returned to RT-11.
                ;
                ;-

                        .ENABL  LSB

001232  016704  GNXTCH: MOV     XOCQE,R4                ;R4->current output queue element
        176656
001236  001426          BEQ     10$                     ;None available...

                        .IF EQ MMG$T
                        ADD     #Q$WCNT,R4              ;R4->word count
                        TST     @R4                     ;Any characters left to output?
                        BEQ     20$                     ;Nope, this request is complete
                        INC     @R4                     ;Yes, now there is one less to do
                        MOVB    @-(R4),R5               ;Get the byte to output
                        INC     @R4                     ;bump pointer to next byte
                        .IFF ;EQ MMG$T
001240  005764          TST     Q$WCNT(R4)              ;Any characters left to output?
        000006
001244  001424          BEQ     20$                     ;Nope, this request is complete
001246  005264          INC     Q$WCNT(R4)              ;Yes, now there is one less to do
        000006
001252  004777          CALL    @$GTBYT                 ;Get the byte to output
        001412
001256  112605          MOVB    (SP)+,R5
                        .ENDC ;EQ MMG$T

001260  046705          BIC     CHMASK,R5               ;Strip the undesired bits
        176674
001264  001762          BEQ     GNXTCH                  ; and nulls are not to be suffered
001266  006227          ASR     (PC)+                   ;Was last character a <CR>?
001270  000000  CRFLG:    .WORD 0                       ; : <CR> flag
001272  103003          BCC     5$                      ;Nope...
001274  120527          CMPB    R5,#C.LF                ;Yes, is this character a <LF>?
        000012
001300  001754          BEQ     GNXTCH                  ;Yes, then suppress it...
001302  120527  5$:     CMPB    R5,#C.CR                ;Is this character a <CR>?
        000015
001306  001002          BNE     10$                     ;Nope...
001310  005267          INC     CRFLG                   ;Yes, set the flag
        177754
001314  000207  10$:    RETURN

001316  005267  20$:    INC     QCHG                    ;Set the 'queue being modified' flag
        177654

                        .IF EQ XL$PC
001322  004767          CALL    DISOUI                  ;Shut off the output
        000716
                        .ENDC ;EQ XL$PC

001326  016704          MOV     XOCQE,R4                ;R4->Current output queue element
        176562
001332  016467          MOV     Q$LINK(R4),XOCQE        ;Replace top of output queue with
        177774
        176554
```

```
                                                    ; next element
   001340  004767          CALL    XLFIN            ;Return the element to RT
           000572
   001344  005067          CLR     QCHG             ;Reset the 'queue being modified' flag
           177626

                           .IF EQ XL$PC
   001350  004767          CALL    ENAOUI           ;Restart the output
           000716
                           .ENDC ;EQ XL$PC

   001354  000726          BR      GNXTCH

                           .DSABL  LSB
```

INPUT INTERRUPT SERVICER

```
                           .SBTTL  INPUT INTERRUPT SERVICER

                   ; This is the input interrupt servicer. Input interrupts are always enabled
                   ; once this driver is called for the first time. Only a "RstDrv" SPFUN
                   ; request will shut off its interrupt enable.

                           .IF EQ XL$PC
   001356                  .DRAST  XI,XL$PRI
   001356  000207          RETURN
   001360  004577  XIINT::JSR      R5,@$INPTR
           001314
   001364  000140          .WORD   ^C<XL$PRI*^o40>&^o340
                           .IFF ;EQ XL$PC
                   XIINT:
                           .ENDC ;EQ XL$PC

                           .ENABL  LSB

```

Hook input interrupt entry point:

```
   001366          HIINT:          ;Input interrupt hook point

   001366  004767          CALL    GETC             ;Get an input character
           001104
   001372  046705          BIC     CHMASK,R5        ;Strip the undesired bits
           176562
   001376  001406          BEQ     10$              ; and nulls are not to be suffered
   001400  120527          CMPB    R5,#C.CTLS       ;Are we being XOFF'd?
           000023
   001404  001004          BNE     20$              ;Nope...
   001406  012767          MOV     #1,RECS          ;Yes, set the 'received XOFF' flag
           000001
           177554
   001414          10$:
                           .IF EQ XL$PC
                            .IF NE XL$PDT
                           CALL    SETLIT           ;Update the PDT lights display
                            .ENDC ;NE XL$PDT
                           .IFF ;EQ XL$PC
                           MOV     #CMD.EI,@CSRA    ;Declare end of interrupt
                           .ENDC ;EQ XL$PC

   001414  000207          RETURN

   001416  120527  20$:    CMPB    R5,#C.CTLQ       ;Are we being XON'd?
           000021
   001422  001005          BNE     30$              ;Nope...
   001424  005067          CLR     RECS             ;Yes, reset the 'received XOFF' flag
           177540

                           .IF EQ XL$PC
   001430  004767          CALL    ENAOUI           ;Get the output going again
           000636
                           .IFF ;EQ XL$PC
                           CLR     SNDS             ;Indicate that an XON has been
                           MOVB    #C.CTLQ,@DBUF    ; sent
                           .ENDC ;EQ XL$PC

```

```
   001434  000767           BR      10$
                   ; Here for characters other than XON (^Q) and XOFF (^S)
   001436  005767  30$:     TST     XIBFRE                  ;Any room in the input buffer?
           001170
   001442  001427           BEQ     50$                     ;Nope, go force an XOFF to the host
                   ; We have room, so store the character in the ring buffer. It will
                   ; be processed at FORK level.
   001444  016704           MOV     XIBIN,R4                ;Yes, R4=offset into buffer
           001156
   001450                   .ADDR   #XIBUF,R4,ADD           ;Add address of start of buffer
   001450  060704           ADD     PC,R4
   001452  062704           ADD     #XIBUF-.,R4
           001054
   001456  110514           MOVB    R5,@R4                  ;Store the character
   001460  005367           DEC     XIBFRE                  ;Buffer has one less free byte now
           001146
   001464  005267           INC     XIBIN                   ;Bump the offset for next time
           001136
   001470  026727           CMP     XIBIN,#BUFSIZ           ;Time to wrap?
           001132
           000100
   001476  103402           BLO     40$                     ;Nope...
   001500  005067           CLR     XIBIN                   ;Reset the buffer offset
           001122
                   ; Here to check for 'low-water' mark (running out of buffer space)
   001504  026727  40$:     CMP     XIBFRE,#STPSIZ          ;Crossed the 'low-water' mark yet?
           001122
           000020
   001512  101010           BHI     60$                     ;Nope, then go process some input
   001514  005767           TST     SNDS                    ;Yes, have we already sent an XOFF?
           177420
   001520  003005           BGT     60$                     ;Yes, so go process some input
                   ; Here to send an XOFF to the host
   001522          50$:
                            .IF EQ XL$PC
   001522  012767           MOV     #-1,SNDS                ;Request an XOFF to be sent
           177777
           177410
   001530  004767           CALL    ENAOUI                  ;Turn on output to make sure
           000536
                            .IFF ;EQ XL$PC
                            MOV     #1,SNDS                 ;Indicate that an XOFF has been
                            MOVB    #C.CTLS,@DBUF           ; sent
                            .ENDC ;EQ XL$PC
                   ; Here to process some input
   001534  005767  60$:     TST     XICQE                   ;Any requests to satisfy?
           176334
   001540  001725           BEQ     10$                     ;No, so just return
                            .IF NE XL$PC
                            MOV     #CMD.EI,@CSRA           ;Declare end of interrupt
                            .ENDC ;NE XL$PC
   001542                   .BR     XIIN
                            .DSABL  LSB
```

PROCESS INPUT RECEIVED FROM INTERRUPT SERVICER

```
                                .SBTTL  PROCESS INPUT RECEIVED FROM INTERRUPT SERVICER

                                .ENABL  LSB

                        ; This routine runs at fork level. It's purpose is to remove characters
                        ; from the ring buffer and use them to satisfy input requests.

   001542  005267  XIIN:   INC     INPRC                   ;Did someone beat us to this routine?
           000254
   001546  001124          BNE     110$                    ;Yes..

   001550  004767          CALL    SAV30
           000250

                        ; We have the routine. Now we loop to process as much of the input as we can.
                        ; Clear flag to say we own routine and no others can come in. This can be
                        ; done because we are going to check to see if anything is in the input buffer
                        ; after clearing the flag.

   001554          5$:;;;  CLR     INPRC                   ;We're now the owner of this routine

.
.
.

   001554  026727          CMP     XIBFRE,#RSTSIZ          ;Crossed the 'high-water' mark yet?
           001052
           000060
   001562  103410          BLO     10$                     ;Nope...
   001564  005767          TST     SNDS                    ;Yes, have we already sent an XON?
           177350
   001570  001405          BEQ     10$                     ;Yes...

                                .IF EQ XL$PC
   001572  012767          MOV     #-2,SNDS                ;No, then request an XON to be sent
           177776
           177340
   001600  004767          CALL    ENAOUI                  ;Turn on output to make sure
           000466
                                .IFF ;EQ XL$PC
                                CLR     SNDS            ;Now indicate that an XON has been
                                MOVB    #C.CTLQ,@DBUF   ; sent
                                .ENDC ;EQ XL$PC

   001604  016704  10$:    MOV     XICQE,R4                ;Any input requests to satisfy?
           176264
   001610  001500          BEQ     100$                    ;Nope...
   001612  006227          ASR     (PC)+                   ;Time to return an EOF?
   001614  000000  CTZFLG:  .WORD  0                       ; : EOF flag (^Z)
   001616  103472          BCS     90$                     ;Yes...
   001620  026727          CMP     XIBFRE,#BUFSIZ          ;Anything in the buffer?
           001006
           000100
   001626  001471          BEQ     100$                    ;Nope...

                        ; Here to remove a character from the input ring buffer

   001630  016705          MOV     XIBOUT,R5               ;R5=Offset into buffer for next char.
           000774
   001634                  .ADDR   #XIBUF,R5,ADD           ;Add address of start of buffer
   001634  060705          ADD     PC,R5
   001636  062705          ADD     #XIBUF-.,R5
           000670
   001642  111505          MOVB    @R5,R5                  ;Get a character from the ring buffer
   001644  005267          INC     XIBFRE                  ;Buffer has one more free byte
           000762
   001650  005267          INC     XIBOUT                  ;Bump offset for next time
           000754
   001654  026727          CMP     XIBOUT,#BUFSIZ          ;Time to wrap?
           000750
           000100
   001662  103402          BLO     20$                     ;Nope...
   001664  005067          CLR     XIBOUT                  ;Yes, reset the buffer offset
           000740
   001670  105764  20$:    TSTB    Q$FUNC(R4)              ;Special function read?
           000002
   001674  001003          BNE     30$                     ;Yes..
```

```
001676  120527          CMPB    R5,#C.CTLZ              ;No, is character a ^Z?
        000032
001702  001420          BEQ     40$                     ;Yes, handle it specially

001704          30$:
                        .IF EQ MMG$T
                        ADD     #Q$WCNT,R4              ;R4->Word count
                        MOVB    R5,@-(R4)               ;Return the character
                        INC     (R4)+                   ;Bump the buffer pointer
                        DEC     @R4                     ;Is transfer complete? (z-bit=1 if so)
                        .IFF ;EQ MMG$T
001704  110546          MOVB    R5,-(SP)                ;Return the character
001706  004777          CALL    @$PTBYT                 ; ...
        000760
001712  005364          DEC     Q$WCNT(R4)              ;Is transfer complete? (z-bit=1 if so)
        000006
                        .ENDC ;EQ MMG$T

001716  001422          BEQ     70$                     ;Yes...
001720  026727          CMP     XIBFRE,#BUFSIZ          ;Anything left in buffer?
        000706
        000100
001726  001312          BNE     5$                      ;Yes, go process it
001730  016704          MOV     XICQE,R4                ;R4->Input request queue element
        176140
001734  105764          TSTB    Q$FUNC(R4)              ;Special request?
        000002
001740  001705          BEQ     5$                      ;Nope, process some more input
001742  000402          BR      50$                     ;Yes, then request is done

001744  005267  40$:    INC     CTZFLG                  ;Set the EOF flag
        177644

001750          50$:
                        .IF EQ MMG$T
                        ADD     #Q$WCNT,R4              ;R4->word count
                60$:    CLRB    @-(R4)                  ;Return a zero byte
                        INC     (R4)+                   ;Bump the buffer pointer
                        DEC     @R4                     ;Is the transfer complete?
                        BNE     60$                     ;Nope...
                        .IFF ;EQ MMG$T
001750  105046          CLRB    -(SP)                   ;Return a zero byte
001752  004777          CALL    @$PTBYT                 ; ...
        000714
001756  005364          DEC     Q$WCNT(R4)              ;Is the transfer complete?
        000006
001762  001372          BNE     50$                     ;Nope...
                        .ENDC ;EQ MMG$T

001764  016704  70$:    MOV     XICQE,R4                ;R4->Current input queue element
        176104
001770  016467          MOV     Q$LINK(R4),XICQE        ;Replace top of input queue with
        177774
        176076
                                                        ; next queue element
001776  004767  80$:    CALL    XLFIN                   ;Return the element to RT
        000134
002002  000664          BR      5$                      ;And check for more input

002004  052754  90$:    BIS     #EOF$,@-(R4)            ;Indicate EOF
        020000
002010  000765          BR      70$                     ;And declare queue element done

002012          100$:
                ;;;     DEC     INPRC                   ;Did anything else come in while
                ;;;                                     ; we were otherwise occupied?
                ;;;     BPL     5$                      ;Yes, then go process it
002012  012767          MOV     #-1,INPRC               ;Release the input processing routine
        177777
        000002
002020  000207  110$:   RETURN                          ;Nope, then we'll retire for awhile

                ; This flag is -1 when no one is executing the XIIN routine.
                ; It is zero when someone is executing in the XIIN routine.
                ; It becomes greater than zero to indicate that more input has come
                ; in while someone was executing the XIIN routine.

002022  177777  INPRC:  .WORD   -1
```

```

                       .DSABL  LSB

                ; The following routine is used by XIIN to simulate the effects of a
                ; FORK (saving of registers 0-3 and lowering of priority)

002024  010046  SAV30:  MOV     R0,-(SP)                ;Save some registers
002026  010146          MOV     R1,-(SP)                ; ...
002030  010246          MOV     R2,-(SP)                ; ...
002032  010346          MOV     R3,-(SP)                ; ...
002034  016646          MOV     10(SP),-(SP)            ;Restack the return address
        000010
002040                  .MTPS   #0                      ;Lower our priority
002040  005046          CLR     -(SP)
002042  112716          MOVB    #0,(SP)
        000000
002046  013746          MOV     @#^o54,-(SP)
        000054
002052  062716          ADD     #^o360,(SP)
        000360
002056  004736          CALL    @(SP)+
002060  004736          CALL    @(SP)+                  ;Co-routine back to caller
002062  012603          MOV     (SP)+,R3                ;Restore the registers
002064  012602          MOV     (SP)+,R2                ; ...
002066  012601          MOV     (SP)+,R1                ; ...
002070  012600          MOV     (SP)+,R0                ; ...
002072  005726          TST     (SP)+                   ;Discard old return address
002074  000207          RETURN                          ; and return to caller's caller
```

XLENQ - Place Qelement on internal queue

```
                        .SBTTL  XLENQ   - Place Qelement on internal queue

                ;+
                ;
                ; XLENQ
                ;       Removes the current Qelement from the device queue and places
                ;       it on an internal queue.  It is presumed (by virtue of the way
                ;       RT works) that there will be only one Qelement in the device
                ;       queue.
                ;
                ; Call:
                ;       R4 -> Qelement to be queued
                ;
                ;       JSR     R5,Q
                ;        .BLKW          ;CQE pointer of internal queue
                ;        .BLKW          ;LQE pointer of internal queue
                ;
                ; Return:
                ;       Qelement has been removed from the device queue and placed on
                ;       the specified internnal queue.
                ;
                ;-

```

Internal queuing code; moves queue element to an internal queue:

```
002076  005067  XLENQ:  CLR     XLCQE                   ;Ensure there are no Qelements
        175706
002102  005067          CLR     XLLQE                   ; on the device queue
        175700
002106  005715          TST     @R5                     ;Is our internal queue empty?
002110  001003          BNE     10$                     ;Nope...
002112  010425          MOV     R4,(R5)+                ;Yes, so make it the first
002114  010425          MOV     R4,(R5)+                ; and last element
002116  000205          RTS     R5

002120  005725  10$:    TST     (R5)+                   ;Bump to last element pointer
002122  010446          MOV     R4,-(SP)                ;Save address of new element
002124  011504          MOV     @R5,R4                  ;R4->Last queue element
002126  011664          MOV     @SP,Q$LINK(R4)          ;Link it to the new element
        177774
002132  012625          MOV     (SP)+,(R5)+             ; and make the new element the last
002134  000205          RTS     R5
```

XLFIN - Internal Queue Element Completion

```
                          .SBTTL  XLFIN   - Internal Queue Element Completion

                  ;+
                  ;
                  ; XLFIN
                  ;       Used to inform RT-11 of a Qelement which has completed.
                  ;
                  ; Call:
                  ;       R4 -> Completed Qelement
                  ;
                  ; Return:
                  ;       Qelement has been returned to RT-11
                  ;
                  ; Note:
                  ;       o All registers except R4 are preserved
                  ;       o Fake device queue is used to return the Qelement to
                  ;         RT-11 to avoid race conditions with the real device
                  ;         queue.
                  ;       o A CALL to monitor completion is used because there may
                  ;         be more to do at this time, we don't want to lose control
                  ;         to the monitor yet.
                  ;
                  ;-

```

Internal queuing code; returns queue element, using fake device queue:

```
 002136  010467  XLFIN:  MOV     R4,XLFCQE               ;Queue element we are returning will
         000520
 002142  010467          MOV     R4,XLFLQE               ; become first and last element
         000512
 002146  005064          CLR     Q$LINK(R4)              ;Unlink it from everything else
         177774
 002152                  .ADDR   #XLFCQE,R4              ;R4 -> Fake device queue for passing
 002152  010704          MOV     PC,R4
 002154  062704          ADD     #XLFCQE-.,R4
         000506
                                                         ; to DRFIN
 002160  013705          MOV     @#$SYPTR,R5             ;R5->$RMON
         000054
```

Modified form of .DRFIN, used to return queue element to monitor and gain control when monitor is done. Required for hooks support if queue element completes as a result of call from multiterminal service:

```
 002164  004775          CALL    @$QCOMP(R5)             ;Inform monitor of I/O completion
         000270
 002170  000207          RETURN

                         .IF EQ XL$PC
```

DISINI - Disable input interrupts
ENAINI - Enable input interrupts

```
                         .SBTTL  DISINI  - Disable input interrupts
                         .SBTTL  ENAINI  - Enable input interrupts

 002172          DISINI:
                          .IF NE XL$MTY
 002172  105767          TSTB    O$MTTY                  ;Terminal hooks in use?
         176312
 002176  001404          BEQ     10$                     ;Nope...
 002200  112767          MOVB    #-1,ISPND               ;Disable input interrupt processing
         177777
         176304
 002206  000403          BR      20$
                          .ENDC ;NE XL$MTY

 002210  042777  10$:    BIC     #RC.IE,@XIS             ;Turn off input interrupts
         000100
```

```
        175714
002216  000207  20$:    RETURN

002220          ENAINI:
                         .IF NE XL$MTY
002220  105767          TSTB    O$MTTY                  ;Terminal hooks in use?
        176264
002224  001403          BEQ     10$                     ;Nope...
002226  105067          CLRB    ISPND                   ;Enable input interrupt processing
        176260
002232  000403          BR      20$
                         .ENDC ;NE XL$MTY

002234  052777  10$:    BIS     #RC.IE,@XIS             ;Turn input interrupts back on
        000100
        175670
002242  000207  20$:    RETURN
```

DISOUI - Disable output interrupts
ENAOUI - Enable output interrupts

```
                        .SBTTL  DISOUI  - Disable output interrupts
                        .SBTTL  ENAOUI  - Enable output interrupts

002244          DISOUI:
                         .IF NE XL$MTY
002244  105767          TSTB    O$MTTY                  ;Terminal hooks in use?
        176240
002250  001404          BEQ     10$                     ;Nope...
002252  112767          MOVB    #-1,OSPND               ;Disable output interrupt processing
        177777
        176233
002260  000403          BR      20$
                         .ENDC ;NE XL$MTY

002262  042777  10$:    BIC     #XC.IE,@XOS             ;Disable output interrupts
        000100
        175646
002270  000207  20$:    RETURN

002272          ENAOUI:
                         .IF NE XL$MTY
002272  004767          CALL    PREMTY                  ;Prepare for hook
        176274
002276  001405          BEQ     10$                     ;Terminal hooks not active...
002300  105067          CLRB    OSPND                   ;Enable output interrupt processing
        176207
002304  004777          CALL    @MTOENX                 ; and then enable output interrupts
        000334
002310  000403          BR      20$
                         .ENDC ;NE XL$MTY

002312  052777  10$:    BIS     #XC.IE,@XOS             ;Enable output interrupts
        000100
        175616
002320  000207  20$:    RETURN
```

RESBRK - Turn off BREAK
SETBRK - Turn on BREAK

```
                        .SBTTL  RESBRK  - Turn off BREAK
                        .SBTTL  SETBRK  - Turn on BREAK

002322          RESBRK:
                         .IF NE XL$MTY
002322  004767          CALL    PREMTY                  ;Prepare for hook
        176244
002326  001404          BEQ     10$                     ;Terminal hooks not active...
002330  005000          CLR     R0                      ;Deassert BREAK
002332  004777          CALL    @MTYBRX                 ; ...
        000310
002336  000403          BR      20$
                         .ENDC ;NE XL$MTY

002340  042777  10$:    BIC     #XC.BRK,@XOS            ;Deassert BREAK
        000001
        175570
002346  000207  20$:    RETURN

002350          SETBRK:
                         .IF NE XL$MTY
002350  004767          CALL    PREMTY                  ;Prepare for hook
        176216
002354  001405          BEQ     10$                     ;Terminal hooks not active...
002356  012700          MOV     #XC.BRK,R0              ;Assert BREAK
        000001
002362  004777          CALL    @MTYBRX                 ; ...
        000260
002366  000403          BR      20$
                         .ENDC ;NE XL$MTY

002370  052777  10$:    BIS     #XC.BRK,@XOS            ;Assert BREAK
        000001
        175540
002376  000207  20$:    RETURN
```

GETSTT - Get line status
RESSTT - Reset line state bits
SETSTT - Set line state bits

```
                        .SBTTL  GETSTT  - Get line status
                        .SBTTL  RESSTT  - Reset line state bits
                        .SBTTL  SETSTT  - Set line state bits

                ;+
                ;
                ; GETSTT
                ;       Returns the current line status
                ;
                ; Call:
                ;       none
                ;
                ; Return:
                ;       R0 = Line status
                ;
                ; Note:
                ;       R3 is altered
                ;
                ;-

002400          GETSTT:
                         .IF NE XL$MTY
002400  004767          CALL    PREMTY                  ;Prepare for hook
        176166
002404  001403          BEQ     10$                     ;Terminal hooks not active...
002406  004777          CALL    @MTYSTX                 ;Get current line status
        000240
002412  000402          BR      20$
                         .ENDC ;NE XL$MTY

002414  017700  10$:    MOV     @XIS,R0                 ;R0 = Current line status
        175512
002420  000207  20$:    RETURN

                ;+
```

```
                ;
                ; RESSTT
                ;       Deasserts line state bits
                ;
                ; Call:
                ;       R0 = Bits to deassert
                ;
                ; Return:
                ;       R0 = Updated line status
                ;
                ; Note:
                ;       o R3 is altered
                ;
                ;       o Unlike SETSTT, which sets the bits as specified,
                ;         this routine first reads the status and then
                ;         deasserts the undesired bits.
                ;
                ;-

002422  010046  RESSTT: MOV     R0,-(SP)                ;Save bits to deassert
002424  004767          CALL    GETSTT                  ;Get current status
        177750
002430  042600          BIC     (SP)+,R0                ;deassert the desired bits

                         .IF NE XL$MTY
002432  004767          CALL    PREMTY                  ;Prepare for hook
        176134
002436  001403          BEQ     10$                     ;Terminal hooks not active...
002440  004777          CALL    @MTYCTX                 ;Yes, set new line state
        000204
002444  000402          BR      20$
                         .ENDC ;NE XL$MTY

002446  010077  10$:    MOV     R0,@XIS                 ;Set new line status
        175460
002452  000207  20$:    RETURN

                ;+
                ;
                ; SETSTT
                ;       Asserts line state bits
                ;
                ; Call:
                ;       R0 = Bits to assert
                ;
                ; Return:
                ;       R0 = Updated line status
                ;
                ; Note:
                ;       o R3 is altered
                ;
                ;       o Unlike RESSTT, which first reads the status and
                ;         deasserts the undesired bits, this routine simply
                ;         asserts the desired bits.
                ;
                ;-

002454          SETSTT:
                         .IF NE XL$MTY
002454  004767          CALL    PREMTY                  ;Prepare for hook
        176112
002460  001403          BEQ     10$                     ;Terminal hooks not active...
002462  004777          CALL    @MTYCTX                 ;Yes, set desired bits
        000162
002466  000402          BR      20$
                         .ENDC ;NE XL$MTY

002470  050077  10$:    BIS     R0,@XIS                 ;Set new line status
        175436
002474  000207  20$:    RETURN

                        .ENDC ;EQ XL$PC
```

GETC - Input a character
PUTC - Output a character

```
                                 .SBTTL  GETC    - Input a character
                                 .SBTTL  PUTC    - Output a character

                         ;+
                         ;
                         ; GETC
                         ;       Gets a character from the interface.
                         ;
                         ; Return:
                         ;       R5 = Character
                         ;
                         ; Note:
                         ;       In the case of call during multiterminal hook operation,
                         ;       the character is already in R5 due to the multiterminal
                         ;       input interrupt service code.
                         ;
                         ;-

       002476            GETC:
                                  .IF NE XL$MTY
       002476  105767            TSTB    O$MTTY                  ;Terminal hooks in use?
               176006
       002502  001002            BNE     10$                     ;Yep, bypass normal DL input
                                  .ENDC ;NE XL$MTY

                                 .IF EQ XL$PC
       002504  117705            MOVB    @XIB,R5                 ;R5 = Character
               175424
                                 .IFF ;EQ XL$PC
                                 MOVB    @DBUF,R5                ;Get a character from input
                                 .ENDC ;EQ XL$PC

       002510  000207    10$:    RETURN

                         ;+
                         ;
                         ; PUTC
                         ;       Puts a character to the interface.
                         ;
                         ; Call:
                         ;       R5 = Character
                         ;
                         ; Note:
                         ;       In the case of call during multiterminal hook operation,
                         ;       the character is already in R5 due to the multiterminal
                         ;       input interrupt service code.
                         ;
                         ;-

       002512            PUTC:
                                 .IF NE XL$MTY
       002512  105767            TSTB    O$MTTY                  ;Terminal hooks in use?
               175772
       002516  001002            BNE     10$                     ;Yep, bypass normal DL output
                                 .ENDC ;NE XL$MTY

                                 .IF EQ XL$PC
       002520  110577            MOVB    R5,@XOB                 ;Output the character
               175414
                                 .IFF ;EQ XL$PC
                                 MOVB    R5,@DBUF                ;Output the character
                                 .ENDC ;EQ XL$PC

       002524  000207    10$:    RETURN
```

INPUT BUFFER AREA

```
                                 .SBTTL  INPUT BUFFER AREA

```

Internal receive buffer:

```
                ; Reserve space for the input buffer and data to manage the input buffer

   002526         XIBUF:  .BLKB   BUFSIZ                  ;Input buffer
   002626 000000  XIBIN:  .WORD   0                       ;'Next Character In' offset
   002630 000000  XIBOUT: .WORD   0                       ;'Next Character Out' offset
   002632 000100  XIBFRE: .WORD   BUFSIZ                  ;Number of free bytes in buffer

                ; Define areas for fork blocks used by the interrupt servicers

   002634 000000  DQFBLK: .WORD   0,0,0,0
   002636 000000
   002640 000000
   002642 000000

                          .IF NE XL$MTY

```

Handler hooks code; pointers loaded by LOAD code, used to reach hooks routines in multiterminal monitor:

```
                ; Multiterminal handler hooks pointers

   002644         MTOENX: .BLKW                           ; : -> Output enable routine
   002646         MTYBRX: .BLKW                           ; : -> Break control routine
   002650         MTYCTX: .BLKW                           ; : -> Line control routine
   002652         MTYSTX: .BLKW                           ; : -> Line status routine
   002654         TCBADX: .BLKW                           ; : -> TCB we're attached to
                          .ENDC ;NE XL$MTY

                ; Fake queue header for returning completed Qelements

```

Internal queuing—fake device queue. Zero word required to simulate non-held handler:

```
   002656 000000          .WORD   0
   002660         XLFLQE: .BLKW
   002662         XLFCQE: .BLKW

   002664                 .DREND  XL
   002664 000000  $RLPTR::.WORD   0
   002666 000000  $MPPTR::.WORD   0
   002670 000000  $GTBYT::.WORD   0
   002672 000000  $PTBYT::.WORD   0
   002674 000000  $PTWRD::.WORD   0
   002676 000000  $TIMIT::.WORD   0
   002700 000000  $INPTR::.WORD   0
   002702 000000  $FKPTR::.WORD   0

                          .IF EQ XL$PC
```

LOAD - Handler FETCH/LOAD code

```
                          .SBTTL  LOAD    - Handler FETCH/LOAD code

                  ;+
                  ;
                  ; LOAD
                  ;       This routine is entered on FETCH or LOAD of the XL handler
                  ;       and is used 1) to verify use of the handler in the specific
                  ;       configuration and, if needed, 2) to establish the required
                  ;       connections between the handler and the interrupt service of
                  ;       a monitor with support for multiterminal handler hooks.
                  ;
                  ;-

                          .ENABL  LSB

   002704         FETCH::
   002704         LOAD::
   002704 010567          MOV     R5,ENTRY$               ;Save entry point
          000314
   002710 010267          MOV     R2,SLOT$                ; and table size
          000312
```

```
   002714  011505          MOV     @R5,R5                  ;R5 -> Base of handler (in memory)
   002716  013700          MOV     @#$SYPTR,R0             ;R0 -> Base of RMON
           000054

```

Hooks code. Establishes linkages between handler and TCB:

```
                            .IF NE XL$MTY
   002722  105765          TSTB    <O$MTTY-XLLQE>(R5)      ;Terminal hooks to be used?
           000502
   002726  001463          BEQ     20$                     ;Then use normal DL
   002730  016001          MOV     $THKPT(R0),R1           ;R1 -> Multiterminal handler hooks
           000000G
                                                           ; data structure in RMON
   002734  001531          BEQ     60$                     ;Monitor doesn't have the support...
   002736  105721          TSTB    (R1)+                   ;Bypass structure size byte
   002740  112102          MOVB    (R1)+,R2                ;R2 = Number of LUNs on system
   002742  012103          MOV     (R1)+,R3                ;R3 -> TCB list
   002744  012165          MOV     (R1)+,<MTOENX-XLLQE>(R5) ;Set pointer to output enable routine
           002636
   002750  012165          MOV     (R1)+,<MTYBRX-XLLQE>(R5) ;Set pointer to Break control routine
           002640
   002754  012165          MOV     (R1)+,<MTYCTX-XLLQE>(R5) ;Set pointer to Control routine
           002642
   002760  012165          MOV     (R1)+,<MTYSTX-XLLQE>(R5) ;Set pointer to Status routine
           002644
   002764  116500          MOVB    <O$LINE-XLLQE>(R5),R0   ;R0 = Line to attach to
           000503
   002770  100513          BMI     60$                     ;Must be a positive number
   002772  120002          CMPB    R0,R2                   ;Is line in this configuration?
   002774  002111          BGE     60$                     ;Nope, invalid line number
   002776  006300          ASL     R0                      ;Shift for word offset into TCB list
   003000  060003          ADD     R0,R3                   ;R3 -> TCB list entry
   003002  011303          MOV     @R3,R3                  ;R3 -> TCB for LUN
   003004  005763          TST     T.CSR(R3)               ;Is the line present in hardware?
           000016
   003010  001503          BEQ     60$                     ;Nope...
   003012  005763          TST     T.STAT(R3)              ;Is the line a console?
           000014

   003016                  .Assume CONSL$ EQ 100000
   003016  100500          BMI     60$                     ;Yes...
   003020  010500          MOV     R5,R0                   ;R0 -> Handler hook routine
   003022  062700          ADD     #<XLHOOK-XLLQE>,R0      ; ...
           000510
   003026  005763          TST     T.OWNR(R3)              ;Is the line already attached?
           000012
   003032  001403          BEQ     10$                     ;Nope...
   003034  020063          CMP     R0,T.OWNR(R3)           ;Yes, to this handler?
           000012
   003040  001067          BNE     60$                     ;Nope...
   003042  016701  10$:    MOV     ENTRY$,R1               ;R1 -> $ENTRY entry
           000156
   003046  166701          SUB     SLOT$,R1                ;R1 -> $PNAME ENTRY
           000154
   003052  011160          MOV     @R1,-2(R0)              ;Inform handler of its physical name,
           177776
   003056  010365          MOV     R3,<TCBADX-XLLQE>(R5)   ; link the handler to the TCB
           002646
```

HANMC$ disables RT–11 processing of modem control; handler will process modem:

```
   003062  052763          BIS     #<HANMT$!HANMC$>,T.STAT(R3) ; declare line owned by handler
           000000C
           000014
                                                           ; and that handler will process modem,
   003070  010063          MOV     R0,T.OWNR(R3)           ; finally link the TCB to the handler
           000012
   003074  000450          BR      50$
                            .ENDC ;NE XL$MTY

```

The following code protects against vector corruption. Won't allow use of handler in NOMTTY mode if CSR or vector conflicts with a line in multiterminal configuration:

```
003076  032760  20$:    BIT     #MTTY$,$SYSGE(R0)       ;Is this a multiterminal monitor?
        020000
        000372
003104  001444          BEQ     50$                     ;Nope, then there can't be a conflict
003106                  .ADDR   #MTAREA,R0              ;R0 -> .MTSTAT EMT area
003106  010700          MOV     PC,R0
003110  062700          ADD     #MTAREA-.,R0
        000120
003114                  .ADDR   #MTSTAT,R1              ;R1 -> Status block
003114  010701          MOV     PC,R1
003116  062701          ADD     #MTSTAT-.,R1
        000120
003122                  .MTSTA  R0,R1                   ;Get info about multiterminal system
003122  012710          MOV     #31.*^o400+8.,@R0
        017410
003126  010160          MOV     R1,2.(R0)
        000002
003132  005060          CLR     4.(R0)
        000004
003136  104375          EMT     ^o375
003140  103427          BCS     60$                     ;Errors?
003142  013700          MOV     @#$SYPTR,R0             ;R0 -> $RMON
        000054
003146  016701          MOV     MTSTAT,R1               ;R1 -> First TCB in system
        000064
003152  060001          ADD     R0,R1                   ; ...
003154  016702          MOV     MTSTAT+MST.LU,R2        ;R2 = Highest LUN on the system
        000062
                                                        ; (Number_of_LUNs - 1)
003160  005761  30$:    TST     T.CSR(R1)               ;Is this a configured line?
        000016
003164  001410          BEQ     40$                     ;Nope...
003166  026561          CMP     <XIS-XLLQE>(R5),T.CSR(R1) ;Will use of the CSR conflict?
        000124
        000016
003174  001411          BEQ     60$                     ;Yes, reject the load
003176  026561          CMP     <XL$VTB-XLLQE>(R5),T.VEC(R1) ;Will use of the VECTOR conflict?
        000134
        000020
003204  001405          BEQ     60$                     ;Yes, reject the load
003206  066701  40$:    ADD     MTSTAT+MST.ST,R1        ;On to next TCB
        000032
003212  005302          DEC     R2                      ;More TCB's to check?
003214  002361          BGE     30$                     ;Yep...
003216                  .BR     50$                     ;Nope, use of interface won't conflict

003216  005727  50$:    TST     (PC)+                   ;Success return
003220  000261  60$:    SEC                             ;Error return
003222  000207          RETURN

003224          ENTRY$: .BLKW                           ; : -> $ENTRY table entry
003226          SLOT$:  .BLKW                           ; : Size of a monitor handler table

003230          MTAREA: .BLKW   3                       ; : EMT area for .MTSTAT
003236          MTSTAT: .BLKW   8.                      ; : Status block from .MTSTAT

                        .DSABL LSB

                        .ENDC ;EQ XL$PC
```

UNLOAD - UNLOAD/.RELEASE CODE

```
                        .SBTTL  UNLOAD - UNLOAD/.RELEASE CODE

                ;+
                ; UNLOAD
                ;       On entry due to unload command, verifies interrupts have been
                ;       disabled unless the handler is still in use, indicated by
                ;       non-empty internal queues.
                ;
                ;       On entry due to .RELEASE directive,disable interrupts
                ;
                ;-

                        .ENABL  LSB

```

Prevents unload if internal queues are not empty:

```
003256          UNLOAD::
003256  011505          MOV     @R5,R5                  ;R5 -> Handler entry point (XLLQE)
003260  005765          TST     <STATFG-XLLQE>(R5)      ;Is handler in use?
        000014
003264  001013          BNE     10$                     ;Nope, it can be unloaded...
003266  016546          MOV     <XICQE-XLLQE>(R5),-(SP) ;Check internal queues
        000066
003272  056526          BIS     <XOCQE-XLLQE>(R5),(SP)+ ; ...
        000106
003276  001405          BEQ     RELEAS                  ;They're empty...
003300                  .ADDR   #NOUNLO,R0              ;R0 -> Error message string
003300  010700          MOV     PC,R0
003302  062700          ADD     #NOUNLO-.,R0
        000106
                                                        ; (KMON reports error)
003306  000261          SEC                             ;Indicate error
003310  000207          RETURN                          ; and return to KMON

003312          RELEAS::
003312  011505          MOV     @R5,R5                  ;R5 -> Handler entry point (XLLQE)
003314          10$:
                        .IF EQ XL$PC
                         .IF NE XL$MTY
```

Handler hooks code; disconnects TCB and handler:

```
003314  105765          TSTB    <O$MTTY-XLLQE>(R5)      ;Terminal hooks in use?
        000502
003320  001420          BEQ     20$                     ;Nope...
003322  016501          MOV     <TCBADX-XLLQE>(R5),R1   ;R1 -> TCB we're hooked to
        002646
003326  001426          BEQ     30$                     ;We're not...
003330  004765          CALL    <DISINI-XLLQE>(R5)      ;Disable input
        002164
003334  004765          CALL    <DISOUI-XLLQE>(R5)      ; and output interrupts
        002236
003340  005000          CLR     R0                      ;Deassert all modem control bits
003342  004765          CALL    <SETSTT-XLLQE>(R5)      ; ...
        002446
003346  005061          CLR     T.OWNR(R1)              ;Disconnect TCB from handler
        000012
003352  042761          BIC     #<HANMT$!HANMC$>,T.STAT(R1) ; ...
        000000C
        000014
003360  000411          BR      30$
003362          20$:
                         .ENDC ;NE XL$MTY
003362  016501          MOV     <XIS-XLLQE>(R5),R1      ;R1->Device register base
        000124
003366  042711          BIC     #RC.IE,@R1              ;Turn off input and
        000100
003372  042761          BIC     #XC.IE,4(R1)            ;Output interrupts
        000100
        000004
003400  042711          BIC     #RC.DTR,@R1             ;Now turn off DTR
        000002
                        .IFF ;EQ XL$PC
                        MOV     #RPT.R1,@#XL$CSA        ;Select csr A,write register 1
                        CLR     @#XL$CSA                ;Turn off input and output interrupts
                        BIC     #<M0.DTR>,@#XL$MC0      ;Now turn off DTR
                        .ENDC ;EQ XL$PC

003404  000241  30$:    CLC
003406  000207          RETURN

003410          NOUNLO: .NLCSI  TYPE=I,PART=PREFIX
003410     077          .ASCII  "?XL-"
   .
   .
   .
003414     106          .ASCIZ  "F-Handler may not be unloaded while in use"
   .
   .
   .
```

```
                             .DSABL  LSB

                 000001      .END
```

Symbol table

```
ABTIO$  001000          DVM.NS  000001          JNUM    000506R     002
BATCH$  000010          DV2.V2  040000          JOBMK   000370
BRKDRV= 000202          DZ11$   010000          KT11$   010000
BRKFLG  001132R     002 EIS$    000400          KW11P$  040000
BUFSIZ= 000100          ENAINI  002220R     002 KXCPU$  004000
BUS$    000100          ENAOUI  002272R     002 LDREL$  000020
BUS$C   020000          ENTRY$  003224R     002 LIGHT$  000010
BUS$M   020100          EOF$    020000          LKCS$   020000
BUS$Q   000100          ERLG$   000001          LOAD    002704RG    002
BUS$U   000000          ERL$G = 000000          LSI11$  004000
BUS$X   020100          FBMON$  000001          MMGT$   000002
CACHE$  000001          FETCH   002704RG    002 MMG$T = 000001
CHMASK  000160R     002 FILL$   000001          MPTYS$  001000
CIS$    000200          FILST$  100000          MPTY$   000002
CLK50$  000040          FIX$ED= 000001          MST.CT  000002
CLOCK$  100000          FJOB$   000200          MST.LU  000004
CLRDRV= 000201          FPU11$  000400          MST.ST  000006
CONSL$  100000          GETC    002476R     002 MST.SZ  000020
CRFLG   001270R     002 GETSTT  002400R     002 MST.1T  000000
CSRSAV  000250          GNXTCH  001232R     002 MTAREA  003230R     002
CTRLC$  040000          GSCCA$  010000          MTOENX  002644R     002
CTRLU$  000002          GTLNK$  000400          MTSTAT  003236R     002
CTZFLG  001614R     002 HANMC$= ****** GX       MTTY$   020000
C.CR  = 000015          HANMT$= ****** GX       MTYBRX  002646R     002
C.CTLQ= 000021          HDERR$  000001          MTYCTX  002650R     002
C.CTLS= 000023          HIINT   001366R     002 MTYSTX  002652R     002
C.CTLZ= 000032          HNDLR$  004000          NOUNLO  003410R     002
C.LF  = 000012          HNGUP$  004000          OFFDRV= 000205
DBGSY$  002000          HOINT   001130R     002 OSPND   000513R     002
DH11$   020000          HS2.BI  000001          O$LINE  000511R     002
DISCSR  000174          HS2.KI  000002          O$MTTY  000510R     002
DISINI  002172R     002 HS2.KL  000004          O.CSR   000442
DISOUI  002244R     002 HS2.KU  000010          O.ERR   000616
DOC$UN= 000000          HS2.MO  000020          O.LINE  000534
DQFBLK  002634R     002 HWDSP$  000004          O.MTTY  000546
DTACH$  000020          HWFPU$  000100          O.NOR   000614
DTRDRV= 000206          H1.ABT  001002          O.VEC   000514
DVC.CT  000006          H1.BR   001014          PAGE$   000200
DVC.DE  000010          H1.CQE  001010          PDP60$  100000
DVC.DK  000004          H1.FG2  001016          PDP70$  040000
DVC.DL  000012          H1.FLG  001010          PREMTY  000572R     002
DVC.DP  000011          H1.HLD  001004          PROS$   020000
DVC.LP  000007          H1.LDT  001024          PS      177776
DVC.MT  000005          H1.LQE  001006          PUTC    002512R     002
DVC.NI  000013          H1.NDF  001026          P1$EXT  000432
DVC.NL  000001          H1.NOP  001012          QCHG    001176R     002
DVC.PS  000014          H1.SCK  001020          QUEUE$  002000
DVC.SB  000020          H1.SDF  001022          Q$BLKN  000000
DVC.SI  000016          H1.VEC  001000          Q$BUFF  000004
DVC.SO  000017          INCV$   000400          Q$COMP  000010
DVC.TP  000003          INEXP$  000100          Q$CSW   177776
DVC.TT  000002          INPRC   002022R     002 Q$FUNC  000002
DVC.UK  000000          INSCSR  000176          Q$JNUM  000003
DVC.VT  000015          INSDAT  000200          Q$LINK  177774
DVM.DM  000002          INSSYS  000202          Q$MEM   000014
DVM.DX  000001          ISPND   000512R     002 Q$PAR   000012
DVM.NF  000200          I$MTTY  000244          Q$UNIT  000003
Q$WCNT  000006          THK.OE  000004          XIBUF   002526R     002
Q.BLKN  000004          THK.ST  000012          XICQE   000074R     002
Q.BUFF  000010          THK.SZ  000014          XIIN    001542R     002
Q.COMP  000014          THK.TC  000002          XIINT   001360RG    002
Q.CSW   000002          TH.GOC  000001 G        XILQE   000076R     002
Q.ELGH  000024          TH.PIC  000002 G        XIS     000132R     002
Q.FUNC  000006          TIMER$  002000          XITSW$  000040
Q.JNUM  000007          TIMIT$  000004          XLCQE   000010RG    002
Q.LINK  000000          TIM$IT= 000001          XLDONE  000622R     002
Q.MEM   000020          TSXP$   100000          XLDSIZ  000000
Q.PAR   000016          TTBF$I  000206          XLEND = 002704RG    002
Q.UNIT  000007          TTBF$O  000050          XLENQ   002076R     002
```

```
Q.WCNT  000012          T.CNFG  000000          XLFCQE  002662R     002
RC.CD = 010000          T.CNF2  000002          XLFIN   002136R     002
RC.CTS= 020000          T.CSR   000016          XLFLQE  002660R     002
RC.DTR= 000002          T.FCNT  000005          XLHOOK  000516R     002
RC.IE = 000100          T.ICTR  000044          XLINT   001122RG    002
RC.RI = 040000          T.IGET  000046          XLLQE   000006RG    002
RC.RTS= 000004          T.IPUT  000042          XLPNAM  000514R     002
READ    000070R     002 T.IRNG  000040          XLSTRT  000000RG    002
RECS    001170R     002 T.ITOP  000050          XLSTS   007057
RELEAS  003312RG    002 T.JOB   000024          XLSYS   000006RG    002
RESBRK  002322R     002 T.LPOS  000011          XL$COD  000057
RESSTT  002422R     002 T.NFIL  000026          XL$CSR= 176500 G
RONLY$  040000          T.OCHR  000010          XL$DVE= 000000
RSTSIZ= 000060          T.OCTR  000262          XL$END  002664RG    002
RTEM$   000010          T.OGET  000264          XL$LUN= 000001
RTE$M = 000000          T.OPUT  000260          XL$MTY= 000001
SAV30   002024R     002 T.OTOP  000266          XL$NAM= 113740
SETBRK  002350R     002 T.OWNR  000012          XL$PC = 000000
SETSTT  002454R     002 T.PRI   000022          XL$PDP= 000001
SHARE$  002000          T.PTTI  000027          XL$PDT= 000000
SLEDI$  000020          T.PUN   000025          XL$PRI= 000004
SLKMO$  000002          T.STAT  000014          XL$SBC= 000000
SLOT$   003226R     002 T.TCTF  000030          XL$SPC= 000001
SNDS    001140R     002 T.TFIL  000004          XL$VEC= 000300 G
SPECL$  010000          T.TID   000032          XL$VTB  000142RG    002
SPFUN   000162R     002 T.TNFL  000031          XOB     000140R     002
SPFUN$  002000          T.TTLC  000036          XOCQE   000114R     002
SP.CCB= 000004          T.VEC   000020          XOLQE   000116R     002
SP.JOB= 000006          T.WID   000006          XOS     000136R     002
SRDDRV= 000203          UNITMK  000007          $BLKEY  000256
STASK$  040000          UNLOAD  003256RG    002 $CHKEY  000260
STATFG= 000022R     002 USR$    001000          $CNFG1  000300
STPSIZ= 000020          VARSZ$  000400          $CNFG2  000370
STSDRV= 000204          VECSAV  000246          $CNFG3  000466
ST.CD = 000010          VIRTV$  105372          $CNTXT  000320
ST.CTS= 000004          VS6$0   001000          $CSW    000004
ST.RI = 000020          WONLY$  020000          $DATE   000262
ST.XFH= 000001          WRITE   000104R     002 $DECNT  000474
ST.XOF= 000002          WRWT$   000040          $DFLG   000264
SWREG$  000004          XC.BRK= 000001          $DWTYP  000440
TCBADX  002654R     002 XC.IE = 000100          $ELTIM  000422
THK.BK  000006          XIB     000134R     002 $EMTRT  000400
THK.CT  000010          XIBFRE  002632R     002 $ERRBY  000052
THK.LE  000000          XIBIN   002626R     002 $ERRCN  000356
THK.NU  000001          XIBOUT  002630R     002 $ERRLE  000376
$EXTIN  000416          $QHOOK  000456          $USRRB  000053
$E16LS  000316          $RLPTR  002664RG    002 $USRSP  000042
$FKPTR  002702RG    002 $RMON   000000          $USRTO  000050
$FORK   000402          $RM2CO  000472          $VIRT   000000
$GETVE  000436          $RTSPC  000464          $VIRTO  000002
$GTBYT  002670RG    002 $SCROL  000302          $WILDD  000454
$GTVEC  000354          $SLOT2  000502          $XTTPB  000500
$HSUFF  000412          $SPSIZ  000504          $XTTPS  000476
$H2CA   000462          $SPSTA  000414          $$$VER= 000022 G
$H2UB   000460          $SPUSR  000272          .AUDIT  107123 G
$IFMXN  000377          $STATW  000366          .XL     000044 G
$IMPLO  000446          $SYCOM  000040          .XLGEN= 000020 G
$INCH   000007          $SYIND  000364          ...V1 = 000003
$INCL   000006          $SYNCH  000324          ...V10= 000100
$INDDV  000426          $SYPTR  000054          ...V11= 000200
$INDST  000417          $SYSCH  000244          ...V12= 000200
$INPTR  002700RG    002 $SYSGE  000372          ...V13= 000000
$JOBNU  000322          $SYSUP  000277          ...V14= 000000
$JOBS   000455          $SYSVE  000276          ...V15= 000176
$JSW    000044          $SYUNI  000274          ...V16= 000000
$JSX    000004          $TCFIG  000424          ...V17= 000000
$KMONI  000450          $THKPT= ****** GX       ...V18= 000000
$LOWMA  000326          $TIMIT  002676RG    002 ...V19= 000000
$MAXBL  000314          $TRPLS  000434          ...V2 = 000000
$MEMPT  000430          $TRPSE  000442          ...V20= 000000
$MEMSZ  000420          $TTFIL  000056          ...V21= 000000
$MFPS   000362          $TTKB   000306          ...V22= 000000
$MONAM  000406          $TTKS   000304          ...V27= 000000
$MPPTR  002666RG    002 $TTNFI  000057          ...V28= 001714
$MTPS   000360          $TTPB   000312          ...V3 = 000170
$NULJB  000444          $TTPS   000310          ...V4 = 000000
$PNPTR  000404          $TT2RM  000470          ...V5 = 000116
$PROGD  000452          $UFLOA  000046          ...V6 = 001714
```

```
$PROGF  000453           $USUSRAR  000374          ...V9 = 000017
$PTBYT  002672RG    002 $USRLC  000266          ...V97= 000014
$PTWRD  002674RG    002 $USRLO  000352          ...V98= 000000
$QCOMP  000270           $USRPC  000040          ...V99= 177777


. ABS.  000622    000   (RW,I,GBL,ABS,OVR)
        000000    001   (RW,I,LCL,REL,CON)
XLDVR   003467    002   (RW,I,LCL,REL,CON)
Errors detected:  0
```