RT–11
Software Support Manual

AA–H379B–TC

March 1983

This document provides detailed descriptions of the components of the RT–11 operating system. It is most useful to system programmers, but it provides valuable background information for application programmers as well.

This manual supersedes the *RT–11 Software Support Manual*, Order No. AA–H379A–TC.

Operating System: RT–11 Version 5.0

To order additional documents from within DIGITAL, contact the Software Distribution Center, Northboro, Massachusetts 01532.

To order additional documents from outside DIGITAL, refer to the instructions at the back of this document.

digital equipment corporation · maynard, massachusetts

First Printing, March 1980
Revised, March 1983

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

M18500

Contents

Chapter 3 Resident Monitor

Chapter 4 Extended Memory Feature

Chapter 6 Interrupt Service Routines

Chapter 7 Device Handlers

Chapter 8 File Formats

Chapter 9 File Storage

Chapter 10 Programming for Specific Devices

Tables

Figures

Preface

Purpose and Audience

The purpose of the *RT–11 Software Support Manual* is to provide detailed descriptions of the software components of the RT–11 operating system.

It is intended for programmers with experience in MACRO–11 assembly language who are interested in system-level programming, and for all application programmers who want to improve their technical understanding of the RT–11 operating system. (While the *RT–11 Software Support Manual* is not strictly a tutorial manual, it does provide valuable background information for application programmers.)

This manual will be particularly useful to you if you are a system programmer and your job is to support RT–11 for other users, you need to use devices that RT–11 does not already support, or you plan to alter the RT–11 software components. This manual can help you design more efficient programs if you are an applications programmer, especially if you plan to use the foreground/background, extended memory, or multi-terminal capabilities of RT–11.

NOTE

> DIGITAL does not maintain software that you have changed in any way! Altering the RT–11 software components voids your warranty and terminates your maintenance service, so refrain from making changes unless you have the technical expertise to be responsible for the system afterwards.

Before you read this manual you should be familiar with the topics covered in the *RT–11 System User's Guide* and with the programmed requests documented in the *RT–11 Programmer's Reference Manual*. The *RT–11 Software Support Manual* contains information that can help you use system resources and the programmed requests more effectively.

The resource that can best help you while you are using this manual — especially if you are interested in monitor internals — is the microfiche listing of the RT–11 commented source files.

Design

This manual consists of ten chapters and three appendixes. The first two chapters provide an overview of the RT–11 system in general as well as information on the components, their arrangement in memory, and their gross structure. The chapters that follow describe the previously introduced system components in greater depth.

Chapter 1 provides an overview of the history of RT–11's development.

Chapter 2 describes how the software components are arranged in memory and shows how the arrangement changes dynamically. It also provides an overview of the components themselves.

Chapter 3 describes the internals of the Resident Monitor that are generally common to the three RT–11 monitors. Topics that it covers include terminal service, timer service, I/O queuing, foreground/background considerations, system jobs, and data structures.

Chapter 4 describes the internals of the Resident Monitor that are the basis of extended memory systems. It provides information on how the memory management hardware works, how RT–11 implements support for 124K words of memory, and how to design and code application programs.

Chapter 5 covers a special feature of RT–11: the ability to use more than one terminal, or **multi-terminal** support. The chapter includes an example application program.

Chapter 6 is an introduction to interrupt service in RT–11. It is useful to programmers who need to add a device to their system configuration that is not already supported by RT–11. The chapter defines the structure and contents of an in-line interrupt service routine, and includes information for servicing interrupts in different RT–11 monitor environments.

Chapter 7 is a logical continuation of Chapter 6. It explains the differences between in-line interrupt service routines and device handlers. It describes how to design, code, install, and debug a device handler. The chapter also covers some special features of handlers and gives considerations for handlers that will operate in various RT–11 monitor environments. Lastly, it lists requirements for system device handlers, and describes the bootstrap.

Chapter 8 describes the structure and format of RT–11 files. It covers stream ASCII, LDA, REL, OBJ, STB, and SAV files, library files, error logging files, CREF files, and files with overlays.

Chapter 9 provides information on device directories, file storage, and formats. It documents the structure of directories for random-access devices, and shows how to repair a directory that has been corrupted. It also describes the structure of magtapes and cassettes.

Chapter 10 describes unique attributes of various physical devices and provides information necessary for programming specifically for those devices.

Appendix A provides commented listings of three RT–11 device handlers: RK, DX, and PC.

Appendix B explains how to convert device handlers that were written for Version 4 of RT–11 to the current device handler format.

Appendix C contains a listing of a sample application program that uses in-line interrupt service to control an analog-to-digital converter in a typical laboratory situation.

Documentation Conventions

The following symbolic and vocabulary conventions are used throughout this manual. Familiarize yourself with them before you continue reading.

Memory refers to all kinds of physical storage in the computer itself; it includes core and semiconductor memory. It is distinguished from storage on peripheral devices, such as disk or tape.

In all diagrams of memory, the high addresses are at the top of the picture and the bottom of the figure represents address 0. In descriptions of data structures and tables, low addresses and offsets are at the top of each table.

In discussions of extended memory systems, **low memory** refers to memory below the 28K-word boundary. However, for LSI computers with the MSV11-DD memory board and a special jumper installed, low memory consists of the memory locations below the 30K-word boundary.

The following acronyms are used throughout this manual:

Name	Meaning
USR	User Service Routine
RMON	Resident Monitor
KMON	Keyboard Monitor
FB	Foreground/Background
XM	Extended Memory
SJ	Single-Job
BL	Baseline
EOT	End-of-tape
EOF	End-of-file
LEOT	Logical end-of-tape
BOT	Beginning-of-tape
CSW	Channel Status Word
PS	Processor Status Word

For your convenience, the following table shows the octal mask used to set, clear, or test each bit in a 16-bit word.

Bit	Octal Mask
0	1
1	2
2	4
3	10
4	20
5	40
6	100
7	200
8	400
9	1000
10	2000
11	4000
12	10000
13	20000
14	40000
15	100000

Chapter 1
Historical Overview

At its conception in 1972, RT–11 was designed to be a small, fast, easy-to-use operating system for the PDP–11 family of minicomputers. It was developed as a single-user system for real-time and computational use; its target applications were data acquisition, process control, and, of course, program development.

The following sections provide an overview of the history of RT–11's development, showing how the operating system has evolved over the course of eight years and four major releases. For a comprehensive overview of the hardware, software, and documentation components of today's RT–11 operating system, see Chapter 1 of the *RT–11 System User's Guide*.

The year 1971 was an exciting time for the computer industry. The PDP–11 computer was only a year old and DIGITAL was making computing power feasible for thousands of applications with the introduction of this relatively inexpensive 16-bit minicomputer.[1]

The software then available for the PDP–11 consisted of PTS (Paper Tape Software, which included the PAL–11S Assembler) and DOS–11 (a batch-oriented system). Clearly, the situation called for a low-cost, interactive system that could be used for real-time and computational applications, and for program development.

A popular operating system for the PDP–8, called OS/8, was the design model for the new PDP–11 operating system, tentatively called OS–11. The new operating system was designed to be a small, single-user, interactive system with event-driven real-time I/O, that would run on PDP–11 computers with 28K words of memory or less. It was designed to have a simple, modular structure; device handlers would be used for I/O transfers so application programming could be device-independent, and files would be stored in contiguous blocks on disk so record management would not be a programming concern.

1.1 Version 1

Actual development work on OS–11 began in the fall of 1972. A group of five system programmers and one technical writer set about refining the design for OS–11 and producing the software and the manual. The groundwork was laid to make OS–11 compatible with OS/8 and TOPS–10.

[1] *Computer Engineering: A DEC View of Hardware Systems Design*, by C. Gordon Bell, J. Craig Mudge, and John E. McNamara, Digital Press, 1978.

The first version of OS–11 included the single-job monitor and a set of program development tools: EDIT, MACRO–11, LINK, ODT, PIP, PATCH, EXPAND and ASEMBL (tools for developing macros in 8K-word systems), and PIPC (for cassettes). BASIC–11, the first product to require RT–11 as its base system, was also part of Version 1. The single-job monitor provided necessary services to running programs and supervised the queued I/O system. The operating system supported seven devices: RK, LP, TT, CT, PR, PP, and DT.

OS–11 was renamed first to RTPS–11 (Real-Time Programming System), then to RT–11 (Real Time). Version 1 of RT–11 was completed in the fall of 1973, and support for the GT40 video display was added in early 1974.

1.2 Version 2

It soon became apparent that RT–11 was successful. More system programmers and technical writers were added to the group, and development for another release was begun. Versions 2, 2B, and 2C brought some significant new features to the operating system. A new monitor was developed that permitted two jobs to run in a foreground/background environment. Support was added for new peripheral devices, including MM, MT, CR, DP, RF, DX, and DS. A number of utility programs were added to improve the set of program development tools. These included CREF, LIBR, PATCHO, DUMP, FILEX, SRCCOM, and BATCH. FORTRAN IV was released with Version 2, and the operating system software included a library of FORTRAN-callable subroutines, called SYSLIB. Version 2 was completed in the fall of 1974; the 2C update was released in early 1976.

1.3 Version 3

Version 3 of RT–11 was another major release. Most significant was the development of the extended memory monitor from a conditional assembly of the foreground/background monitor source files. This permitted RT–11 to support systems with up to 124K words of physical memory. Products such as FORTRAN IV, CTS–300, and Multi-User BASIC–11 took advantage of this feature in ways that were transparent to application programs. Support was included for multi-terminal systems as well, and device error logging was implemented. DCL (DIGITAL Command Language) was developed so that almost all system programs could be accessed by English-like monitor commands. Indirect files provided an easy-to-use alternative to BATCH.

Again, support was added for new DIGITAL peripheral devices: DL, DM, DY, NL, and PC (which replaced PR and PP). And, more system utility programs were introduced: PIP was divided into PIP, DUP, and DIR. Other new utilities included PAT, FORMAT, and RESORC. System generation was designed to permit customization and provide system flexibility. The TECO editor was included in the distribution kits for the first time. Version 3 was completed in the fall of 1977, and the 3B update was made available in early 1978.

1.4 Version 4

With Version 4, RT–11 could be called a mature product. The specific goals of this development effort were to make RT–11 easier to install and maintain. Tools were provided, in the form of BINCOM, SIPP, SRCCOM, and SLP, to make the generation and installation of patches almost automatic. System jobs (special foreground jobs provided by DIGITAL) handled error logging and file queuing. Monitor files were separated from system device handler files, providing greater flexibility while saving storage space. Not least among the accomplishments was a change to the linker that permitted overlays to reside in extended memory rather than on a mass storage device. The KED and K52 Keypad Editors were included in the distribution kits.

Version 4 was completed in early 1980. By then there were well over seventeen thousand RT–11 systems installed around the world, making this operating system a successful venture indeed.

1.5 Version 5

Nothing stands still in the computer industry. New hardware and expanding user needs create demands for up-to-date software. Version 5 of RT–11, released in the spring of 1983, included support for new hardware such as MSCP and the MICRO/PDP–11. The extended memory monitor was changed to support 22-bit memory addressing on Q-bus central processors and to allow use of the .FETCH programmed request under the extended memory monitor. A new virtual memory handler allowed extended memory to be used as though it were a disk. The LD handler was added to support logical disks and console logging. The backup utility BUP and the indirect file processor IND were added to the distribution kit, and SYSGEN was rewritten to make installation and customization still easier. New DCL commands and options were added, as well as CCL (Concise Command Language) and UCL (User Command Linkage). At the same time, however, a minimum system could still run in 16K words of memory, maintaining the RT–11 tradition of being small, fast, interactive, and easy to use.

Chapter 2
System Components and Memory Layouts

This chapter introduces the components of the RT–11 system that can be memory resident. It provides maps of physical memory that show where the components are located, and it indicates how their positions can change dynamically. The components this chapter covers are divided into two groups: static components, which have a relatively fixed position in memory, and dynamic components, whose locations are changeable.

The components are arranged to leave the most space available for user programs and to be flexible. Flexibility is obtained by positioning the components after determining the total amount of memory at bootstrap time. Normally, you do not have to take any special steps to move RT–11 from one PDP–11 computer to another.

2.1 Static Components

The static components have fixed locations in memory. Their actual addresses vary from one PDP–11 computer to the next, depending on how much memory each computer has available. The static components or areas are as follows:

1. Trap vectors
2. System communication area
3. Interrupt vectors
4. I/O page
5. System device handler
6. Resident Monitor
7. Background job

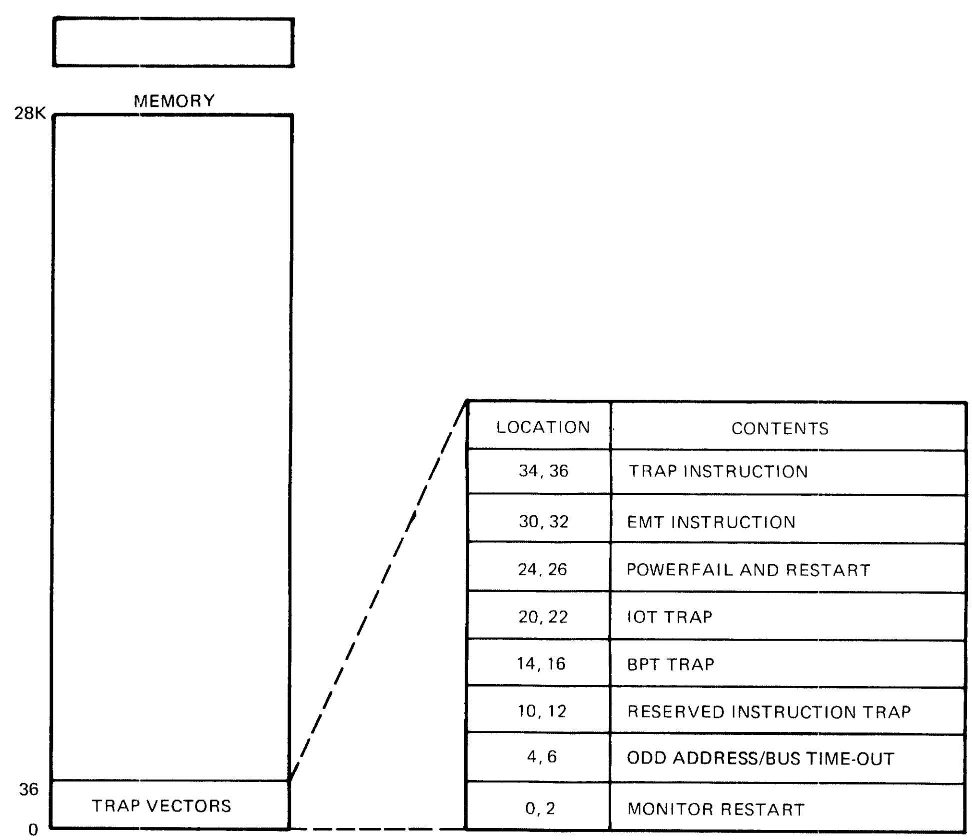
2.1.1 Trap Vectors

Table 2–1 shows the memory locations from 0 to 36, an area that contains the trap vectors. A plus sign (+) marks the locations that are reserved for use by RT–11. You should not attempt to modify these locations; a bitmap protects them each time you load a program. An asterisk (*) marks the locations that your programs can use. Figure 2–1 is a summary of the trap vector area information.

Table 2–1: Trap Vectors

Location	Contents
0,2+	Monitor restart, executes the .EXIT request and returns control to the monitor (has additional uses in XM systems).
4,6+	Odd address and bus time-out trap; RT–11 sets this to point to its internal trap handler.
10,12+	Reserved instruction trap; RT–11 sets this to point to its internal trap handler.
14,16*	BPT (breakpoint trap), T-bit trap (used by debugging utility programs).
20,22*	IOT, input/output trap.
24,26*	Powerfail and restart trap. Your programs can use this location unless you included support for powerfail restart through system generation. If your system includes the powerfail restart feature, locations 24 and 26 are reserved for use by RT–11.
30,32+	EMT, emulator trap; RT–11 uses this for programmed requests.
34,36*	TRAP instruction. Note that you cannot use the TRAP instruction in assembly language subroutines linked with FORTRAN IV, DIBOL, BASIC–11, or MU BASIC–11 programs; these languages use the TRAP instruction for internal error reporting.

Figure 2–1: Trap Vector Area



2.1.2 System Communication Area

The memory locations from 40 through 57 are called the **system communication area**. This area holds information about the program currently executing, as well as certain information normally used only by the monitor.

The diagram in Figure 2–2 is a summary of the system communication area information. Table 2–2 describes the contents of each location.

Figure 2–2: System Communication Area

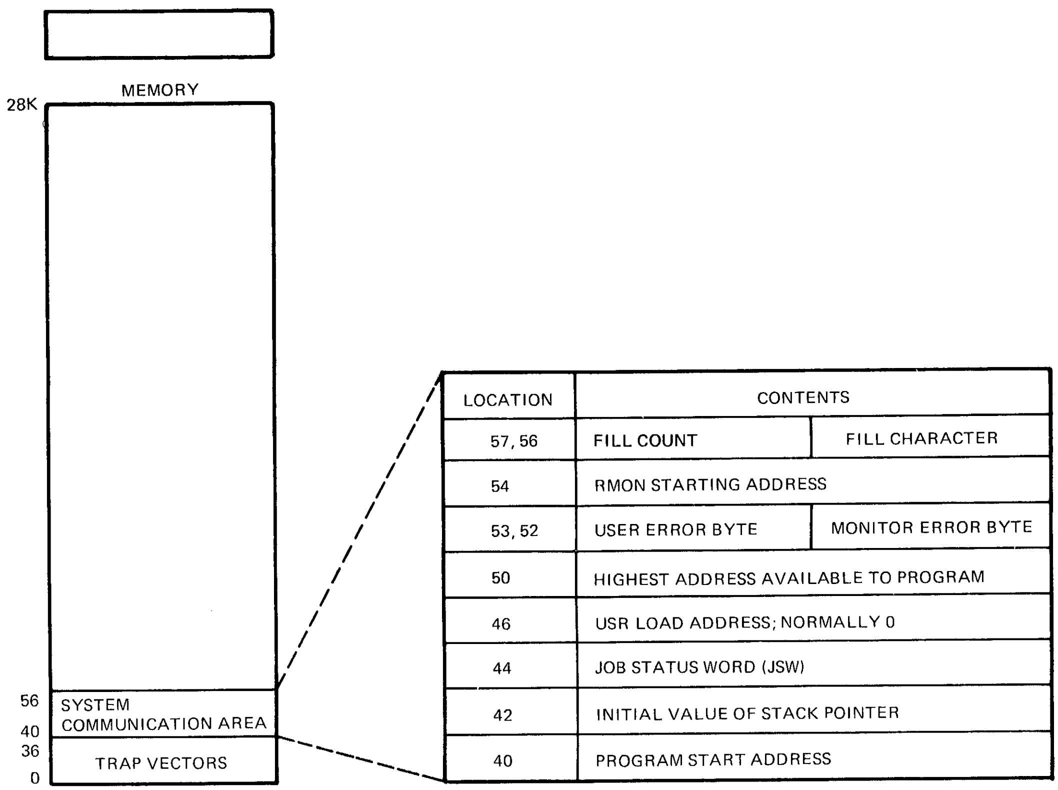


Table 2–2: System Communication Area

Location	Contents
40,41	Start address of job. When you link a file to create an RT–11 executable image, the linker sets the word at address 40 in the program's file to the starting address of the program. This word is loaded into memory location 40 at run time. When a foreground job executes, the FRUN processor relocates this word to contain the actual starting address of the program.
42,43	Initial value of stack pointer. If the user program does not set this value with an .ASECT directive, the value defaults to 1000 or to the top of the program's absolute section, whichever is larger. You can use the linker /B:n option to set the initial value of the background job's stack pointer. If a foreground program does not specify a stack pointer in this word (by using an .ASECT directive), the FRUN processor allocates a default stack of 128 decimal bytes immediately below the program, and the initial stack pointer value is 1000, relative to the base of the foreground job.

(Continued on next page)

Table 2–2: System Communication Area (Cont.)

Location	Contents
44,45	Job Status Word (JSW). This is a flag word for the monitor. The monitor maintains some of the bits itself, and your program can set or clear others. See Section 2.1.2.2 for more information on the JSW.
46,47	USR load address. This word is normally 0, but you can set it in the file or at run time to any valid word address in your program. If this word is 0, the USR loads in its default location through an address contained in offset 266 of RMON. If this word is not 0, the USR loads at the address it specifies, unless the USR is set NOSWAP. This location is cleared by an exit to KMON (via .EXIT, CTRL/C, or fatal error).
50,51	High memory address. In this word the monitor maintains the highest address your program can use. The linker sets this word initially to the high-limit value. You can modify it by using the .SETTOP programmed request. Your program must never modify this word directly. In XM systems, locations 50 and 51 in the file contain the address that is the top of the root section plus the low memory (/O) overlays. In memory, locations 50 and 51 contain the same value unless the program issues a .SETTOP. In this case, these locations contain the highest available virtual address (see Section 4.4.4.6).
52	EMT error code. If a monitor request results in an error, the code number of the error is always returned in byte 52 in memory and the carry bit is set. Each monitor call has its own set of possible errors. Byte 52 in the job's file has a different meaning (see Chapter 8). **NOTE** Always address location 52 as a byte, never as a word, since byte 53 has a separate function.
53	User program error code (USERRB). If a user program encounters errors during execution, it indicates the error by using this byte in memory. See Section 2.1.2.1 for more information about this byte. See Chapter 8 for its meaning in the job's file.
54,55	Address of the beginning of the Resident Monitor. RT–11 always loads the monitor into the highest available memory locations of low (rather than extended) memory; this word in memory points to its first location. Never alter this word – doing so causes RT–11 to malfunction. See Chapter 8 for the meaning of this word in the job's file.
56	Fill character (seven-bit ASCII). Some high-speed terminals require fill (null) characters after printing certain characters. Byte 56 in memory should contain the ASCII seven-bit representation of the character after which fills are required. See Chapter 8 for the meaning of this bit in the job's file.
57	Fill count. This byte in memory specifies the number of fill characters that are required. The number of characters is determined by hardware. If bytes 57 and 56 are 0, no fill is required. See Chapter 8 for the meaning of this byte in the job's file. For more information on the terminals that require fill characters, see the *RT–11 Installation Guide.*

2.1.2.1 User Error Byte — The Keyboard Monitor examines the *user error byte* when a program terminates. If your program has reported a significant error in this byte, KMON can abort any indirect command files in use. This prevents spurious results from occurring if subsequent commands in the indirect file depend on the successful completion of all prior commands.

A program can exit in one of the following states:

- Success
- Warning
- Error
- Severe error
- Unconditionally fatal error

The program status is **success** when the execution of the program is free of errors.

The **warning** status indicates that warning messages occurred, but the program ran to completion.

The **error** status indicates that a user error occurred and the program did not run to completion. This level is also used by RT–11 system programs when they produce an output file even though it may contain errors. For example, a compiler can use the error level to indicate that an object file was produced, but the source program contains errors. Under these conditions, execution of the object file will not be successful if the module containing the error is encountered.

The **severe** status indicates that the program did not produce any usable output, and any command or operation depending upon this program output will not execute properly. This type of error can result when a resource needed by the program to complete execution is not available — for example, insufficient memory space to assemble or compile an application program.

The **unconditionally fatal** status indicates that not only has an operation completely failed, but that the integrity of the monitor itself is questionable.

Utility programs and the Keyboard Monitor always set the user error byte to reflect the result of each monitor command you issue. Normally, indirect command files abort when there has been a monitor command error. By setting the error level to unconditionally fatal with the SET ERROR NONE command, you guarantee that indirect command files will continue to execute despite individual monitor command errors. Only unconditionally fatal errors that indicate problems within the Keyboard Monitor itself abort indirect files at the SET ERROR NONE level. Table 2–3 shows the bits of byte 53, their status, and the status code printed by the RT–11 system utility program messages.

Table 2–3: User Error Byte

Bit	Mask	Status	RT–11 Message
0	1	Success	?prog-I-text, or none
1	2	Warning	?prog-W-text
2	4	Error	?prog-E-text
3	10	Severe	?prog-F-text
4	20	Fatal	?prog-U-text

Bits 5 through 7 of the user error byte are reserved for DIGITAL's future use; do not use them in your programs. Programs should never clear byte 53, and should set it only through a BISB instruction, as the following example shows. If more than one bit is set at any given time, the highest bit is the one that RT–11 recognizes.

```
        USERRB = 53
        SUCCS$ = 1
        WARN$  = 2
        ERROR$ = 4
        SEVER$ = 10
        UFATL$ = 20
             .
             .
             .
ERROR:  BISB #ERROR$,@#USERRB       ;SET ERROR STATUS
        CLR  R0                     ;HARD EXIT
        .EXIT
```

Note that this byte is meaningful only for the Keyboard Monitor and for background jobs. This is because it was designed to be used by system utility programs and language processors, which run as background jobs. A foreground job can set it, but that action has no effect on the system.

2.1.2.2 Job Status Word (JSW) – Bytes 44 and 45 make up the *Job Status Word*, or JSW. Table 2–4 shows the meanings of the bits in this word. The bits marked with an asterisk (*) can be set by a user program during execution. Bits marked with a plus sign (+) are set at load time. Note that some bits can be set at both load and run time. Unused bits are reserved for future use by DIGITAL. Figure 2–3 shows a summary of the JSW.

Table 2–4: Job Status Word (JSW)

Bit Number	Meaning When Set
15	USR swap bit (SJ only). The monitor sets this bit when a program does not require the USR to swap. (See Section 2.2.3 for details on the USR.) Your program must alter this bit.
14+*	Lower-case bit. Disables automatic conversion of typed lower-case to upper-case characters. EDIT sets it when you type the EL command.

(Continued on next page)

Table 2–4: Job Status Word (JSW) (Cont.)

Bit Number	Meaning When Set
13+*	Reenter bit. Indicates that a program can be restarted from the terminal when you type the REENTER command.
12+*	Special mode terminal bit. Indicates that the job is in a special keyboard mode of input. Refer to the explanation of the .TTYIN and .TTINR programmed requests in the *RT–11 Programmer's Reference Manual* for details.
11+*	Pass line to KMON bit. Indicates, when a program exits, that the program is passing a command line to KMON. This action causes any open indirect file to abort. The command line should be stored in the CHAIN information area, locations 500 through 776. R0 must be cleared before exiting. Refer to the example program for .EXIT in the *RT–11 Programmer's Reference Manual.* This bit is not available to foreground or system jobs under the FB and XM monitors.
10+	Virtual image bit (XM only). Indicates that the job to be loaded is a virtual job. You must set this bit yourself in the executable file before you attempt to run the program. Do this at assembly time by using an .ASECT directive and modifying the JSW, or before run time by patching this location in the file. See Chapter 4 for more information on virtual jobs.
9	Overlay bit. This bit is set by the linker if the user program uses the linker overlay feature.
8+	CHAIN bit. This bit can be used in two ways. If it is set in a job's save image, the monitor loads words 500 through 776 from the save file when the job is started, even if the job is entered with .CHAIN. (These words are normally used to pass parameters from one job to another across a .CHAIN.) The monitor sets this bit when the job is running if and only if the job was actually entered with a .CHAIN.
7+*	Error halt bit (SJ only). Indicates that the system should halt when an I/O error occurs. If you want the system to halt when a device I/O error occurs, you should set this bit.
6+*	Inhibit terminal wait bit (FB and XM only). Inhibits the job from entering a console terminal wait state. For more information, refer to the sections concerning .TTYIN, .TTINR, .TTYOUT and .TTOUTR in the *RT–11 Programmer's Reference Manual.*
5+*	Special chain exit bit. If set when a program exits, text in the chain area, locations 510 to 777, is passed to KMON and appended to the command buffer. R0 must be cleared before exiting. This does not abort an open indirect file. Refer to bit 11, above. If you pass multiple command lines, any line containing the @ indirect file command must be the last line of the series.
4+*	Disable single-line editor bit. Setting this bit disables all single-line editor functions.

(Continued on next page)

Table 2–4: Job Status Word (JSW) (Cont.)

Bit Number	Meaning When Set
3+*	Nonterminating .GTLIN bit. When bit 3 of the JSW is set and your program encounters a CTRL/C in an indirect command file, the .GTLIN request collects subsequent lines from the terminal. If you then clear bit 3 of the JSW, the next line collected by the .GTLIN request is the CTRL/C in the indirect command file; this causes the program to terminate. Further input will come from the indirect command file, if there are any more lines in it. The LINK, DUP, SIPP, SLP, QUEMAN, SRCCOM, and LIBR utilities make use of this feature. To activate it in an indirect file, put an uparrow (^) followed by a C on a line by itself in the file. This causes the utilities to accept the response from the terminal instead of taking it directly from the file. The following indirect file shows how to obtain a response from the terminal: RUN LINK TEST,TEST=MOD1,LIB/I ^C All further input to the linker will come from the terminal, as a result of the ^C in the indirect command file.
0–2	Reserved.

Figure 2–3: Job Status Word (JSW) Summary

15	14*+	13*+	12*+	11*+	10+	9	8+
1 = NO USR SWAPPING (SJ ONLY)	1 = LOWER CASE ENABLED	1 = REENTER CAN START JOB	1 = TT SPECIAL MODE	1 = PASS LINE TO KMON	1 = VIRTUAL JOB (XM ONLY)	1 = OVERLAID JOB	CHAIN BIT
1 = HALT ON I/O ERROR (SJ ONLY)	1 = NO TT WAIT STATE	1 = DISABLE SINGLE-LINE EDITOR	1 = SPECIAL .CHAIN EXIT	NON-TERMINATING .GTLIN		RESERVED	
7*+	6*+	5*+	4*+	3*+	2	1	0

BITS MARKED WITH AN ASTERISK (*) ARE BITS THAT YOU CAN SET DURING EXECUTION.
BITS MARKED WITH A PLUS SIGN (+) CAN BE SET AT LOAD TIME.

2.1.3 Interrupt Vectors

Table 2–5 shows the locations in the low memory area that are reserved for interrupt vectors. Figure 2–4 shows how the interrupt vector area relates to the rest of memory.

Table 2–5: Interrupt Vectors

Location	Contents
60,62	DL11: Console terminal input
64,66	DL11: Console terminal output
70,72	PC11: Paper tape reader
74,76	PC11: Paper tape punch
100,102	KW11-L: Line clock
104,106	KW11-P: Programmable clock
110,112	Reserved[1]
114,116	Memory system errors: parity, cache, and uncorrectable ECC errors
120,122	XY11: X/Y Plotter[2]
124,126	DR11-B: DMA interface[2]
130,132	AD01: Analog to digital subsystem[2]
134,136	AFC11: Analog input subsystem[2]
140,142	AA11: Digital to analog subsystem[2]
144,146	AA11: (requires two vectors)[2]
150,152	MSCP device number 1
154,156	MSCP device number 0
160,162	RL11/RLV11: RL01/RL02 Disk cartridge
164,166	Reserved
170,172	LP/LS/LV11 Line printer number 1[2]
174,176	LP/LS/LV11 Line printer number 2[2]
200,202	LP/LS/LV11 Line printer number 0 (includes LA180 parallel interface)
204,206	RH11,RH70: RS03/RS04 Fixed-head disk; RF11: Fixed-head disk
210,212	RK611/RK711: RK06/RK07 Disk cartridge
214,216	TC11: DECtape
220,222	RK11/RKV11: RK05 Disk cartridge
224,226	RH11/RH70: TU16, TE16, TU45 Magtape; TM11: TU10/TE10 Magtape; TS03: Magtape TS11: Magtape first controller (others float) TS05/TSV05: Magtape

(Continued on next page)

1 This vector is used by RSTS/E. Take this into consideration if you run both RT–11 and RSTS/E on the same PDP–11.

2 This vector is assigned to a hardware device that is optional in RT–11. If your configuration includes this device, use this vector for it.

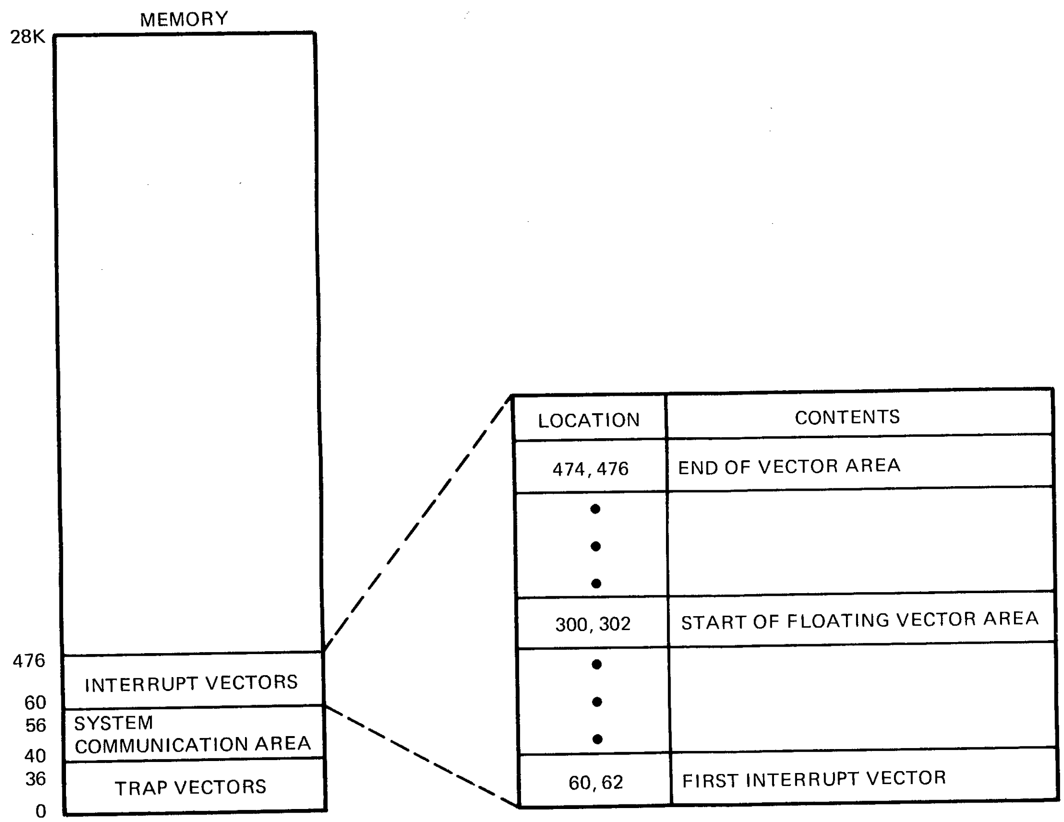
Table 2–5: Interrupt Vectors (Cont.)

Location	Contents
230,232	CD11/CM11/CR11: Card reader
234,236	UDC11: Digital control subsystem[2]
240,242	PIRQ, (programmed interrupt request)[3]
244,246	FPP or FIS floating-point exception
250,252	KT11: Memory management fault
254,256	RP11: RP02/03 Disk; RH11/RH70: RP04/05/06/RM02/03 Disk
260,262	TA11: Cassette tape
264,266	RX11/RXV11/RX211/RX2V1: RX01, RX02 Diskette
270,272	LP/LS/LV11 Line printer number 3[2]
274,276	LP/LS/LV11 Line printer number 4[2]
300,302	Start of the floating vector area
320,322	VT11/VS60 Graphics terminal (requires three vectors)
324,326	VT11/VS60
330,332	VT11/VS60

[2] This vector is assigned to a hardware device that is optional in RT–11. If your configuration includes this device, use this vector for it.

[3] This vector is assigned to hardware that is not supported by RT–11.

Figure 2–4: Interrupt Vector Area

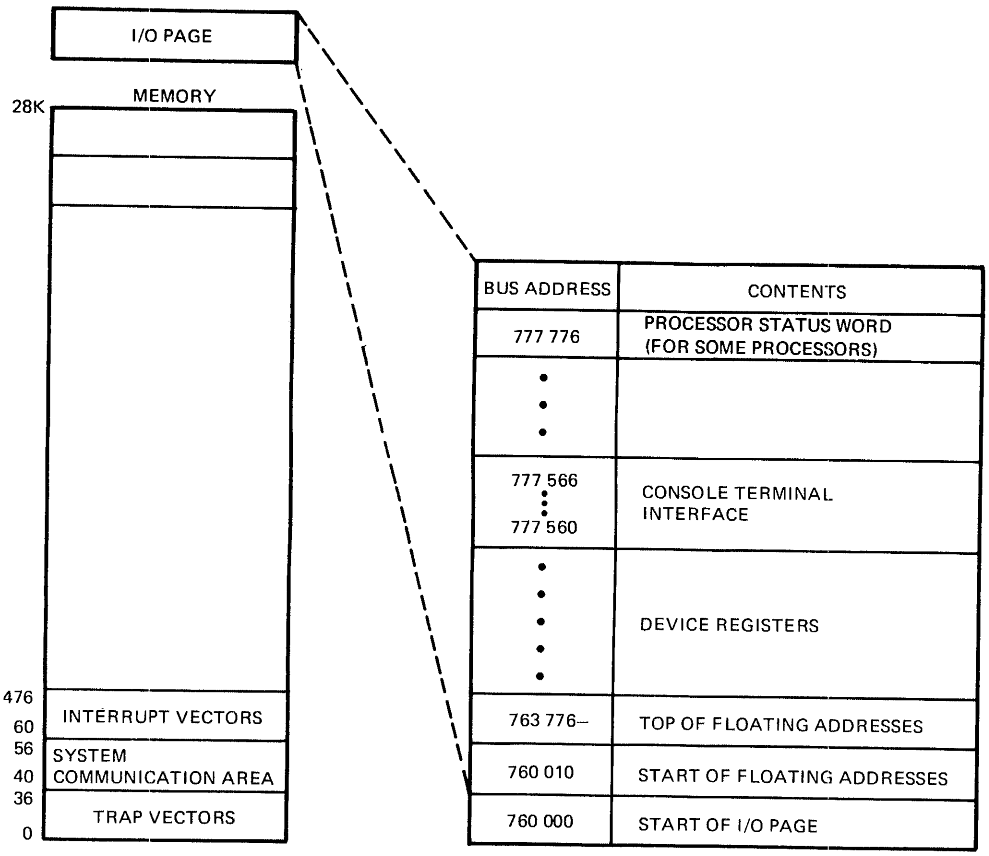


2.1.4 I/O Page

The highest 4K words[1] of addressing space in PDP–11 computers are reserved for device control, status, and data buffer registers. This area is called the **I/O page**. In addition to the device registers, it also contains the Processor Status word (except on the LSI–11/02, PDP–11/03, and PDT), and, for some processors, the system's general registers (R0 through R5), the stack pointer (R6), and the program counter (R7). Locations in the I/O page are directly addressable by application programs and system software, but since they are bus addresses and not memory locations, they cannot be used to store code and data. Figure 2–5 shows where the I/O page is addressed in relation to the rest of the system components. You can find more information on the I/O page and the device registers for your own processor and peripherals in the *PDP–11 Processor Handbook*, the *PDP–11 Peripherals Handbook*, the *Microcomputer Processor Handbook*, the *Memories and Peripherals Handbook*, and in most hardware manuals.

[1] An LSI–11 with MSV–11DD and memory jumper has a 2K-word I/O page and 30K words of regular memory. Throughout this manual, however, a 4K-word I/O page is assumed.

Figure 2–5: I/O Page

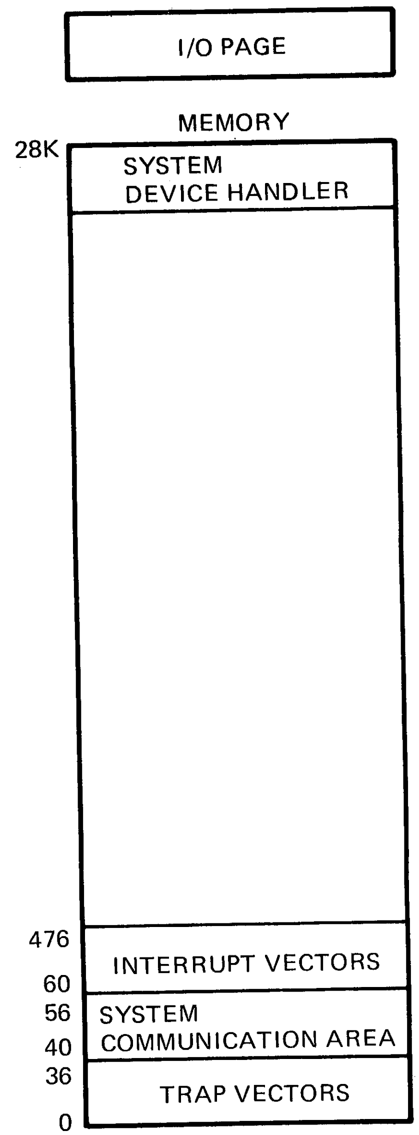


2.1.5 System Device Handler

The **system device handler** is the handler for the device from which the system was bootstrapped. Chapter 7 describes the structure of a system device handler in detail.

At bootstrap time, the monitor is linked together with the system device handler file found on the system volume. The system device handler is loaded into memory first, immediately below the I/O page. The Resident Monitor is loaded below the system device handler. Once it is read into memory, the system device handler remains resident and does not change its location. Figure 2–6 shows where the system device handler resides in memory.

Figure 2–6: System Device Handler



2.1.6 Resident Monitor (RMON)

The **Resident Monitor** (RMON) is the RT–11 monitor component that is always resident in memory. When you bootstrap an RT–11 system, the bootstrap routine determines how much main memory is available. RMON loads at the highest possible low memory address, just below the system device handler. It does not move during system operation.

RMON contains routines to handle the programmed requests in RT–11. It also contains the background job's impure area in FB and XM systems, the error processor, timer routines, console terminal service routines, USR swap routines, and other monitor functions. Figure 2–7 shows a summary of the contents of the Resident Monitor. In the figure, components marked with an asterisk (*) are not part of the SJ Resident Monitor. See Chapter 3 for more information on the Resident Monitor.

Link maps of the distributed RT–11 monitors (base-line, single-job, and foreground/background) are part of the distribution kit. They exist as files named RTBL.MAP, RTSJ.MAP, RTFB.MAP, and RTXM.MAP. Listings of the maps also appear in the *RT–11 Installation Guide*. Table 2–6 lists the p-sects that make up the Resident and Keyboard Monitors.

Figure 2–7: Resident Monitor (RMON)

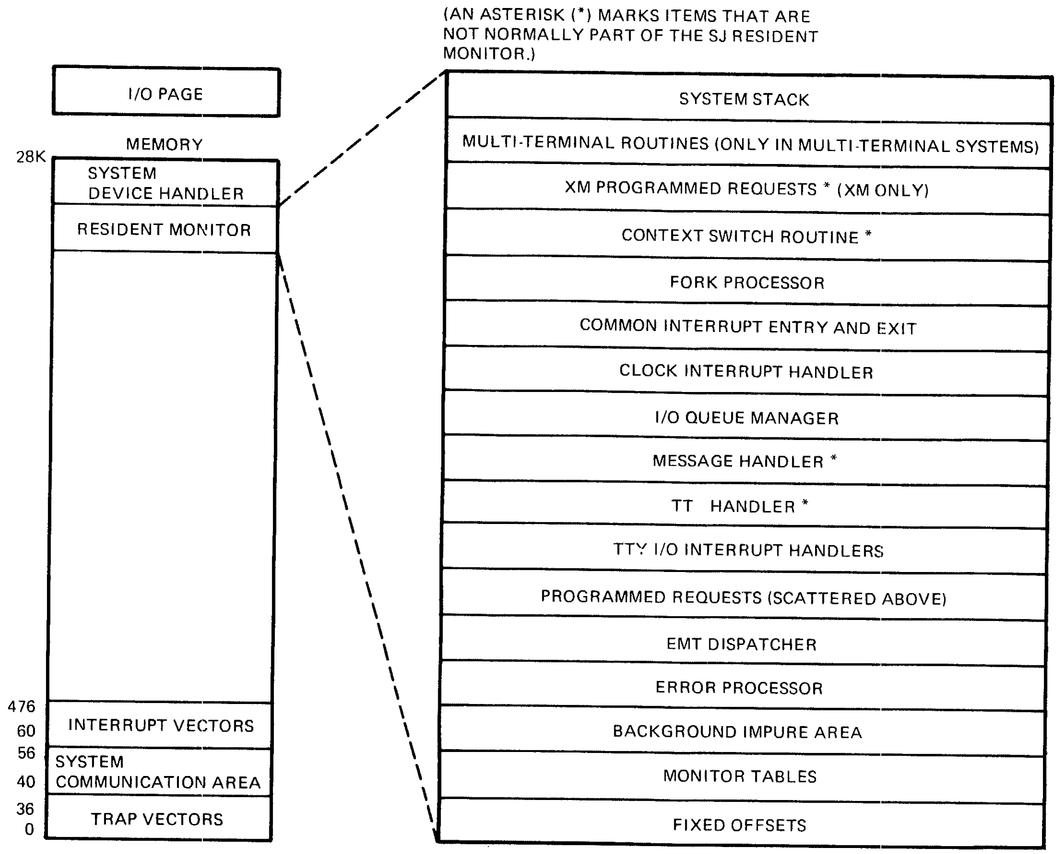


Table 2–6: Monitor P-sects

P-sect Name	Contents
RT11	Keyboard Monitor
RMNUSR	USR buffer and code
RTDATA	Resident Monitor fixed offsets and database
OWNER$	$OWNER table
UNAM1$	$UNAM1 table
UNAM2$	$UNAM2 table
PNAME$	$PNAME table

(Continued on next page)

Table 2–6: Monitor P-sects (Cont.)

P-sect Name	Contents
ENTRY$	$ENTRY table
STAT$	$STAT table
DVREC$	$DVREC table
DVINT$	$DVINT table
MTTY$	Multi-terminal terminal control blocks
RMON	Resident Monitor
XMSUBS	Extended Memory routines
MTEMT$	Multi-terminal programmed requests
MTINT$	Multi-terminal interrupt service
STACK$	Resident Monitor stacks (not in SJ)
PATCH$	Patch space
OVLYnn	Keyboard Monitor overlays containing command processors

2.1.7 Background Job

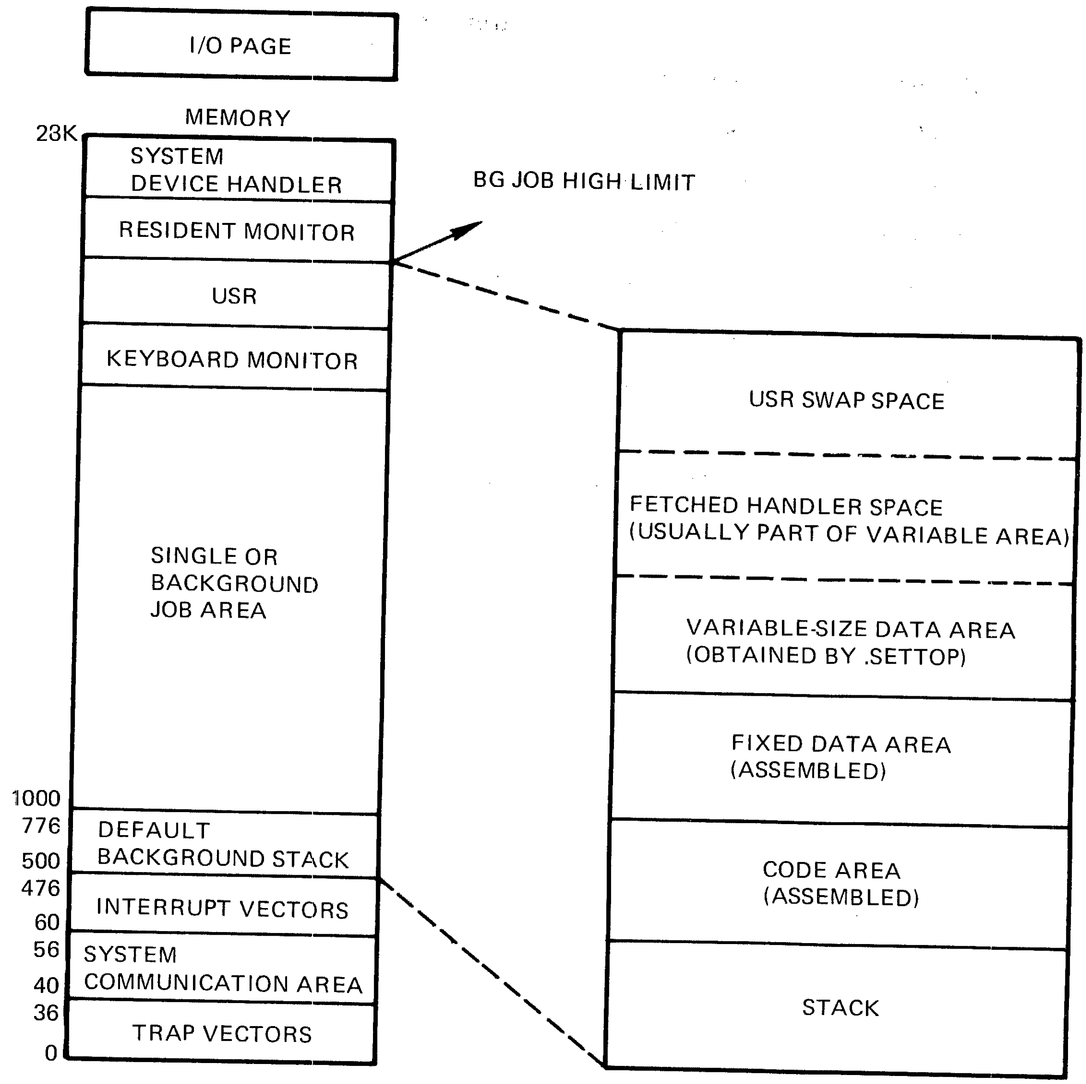
The **user job** in an SJ system and the **background job** in an FB system are essentially identical for the purpose of this discussion. The RT–11 utility programs, such as PIP, DUP, and DIR, run as user jobs. In FB systems, they run as background jobs. Figure 2–8 shows the general structure of a background job, as well as its relative location in memory.

As you can see from Figure 2–8, the background job usually begins loading into memory at location 1000, and loads up to its high limit. There are three ways in which RT–11 can load a background job: RUN, R, and .CHAIN. They are described in the following three sections.

2.1.7.1 RUN Command — One way to load a job is to use the keyboard monitor *RUN command.* The RUN command is the same as the GET and START commands combined. First, if the SAV file is not on the system device, RUN (or GET) loads the handler for the proper device. When this occurs the Keyboard Monitor and the USR, which normally occupy the space above the background job and below RMON, relocate themselves, if necessary. For more information on the USR and the Keyboard Monitor, see later sections of this chapter.

The space available for background job loading consists of the background job area, the space occupied by KMON, and the space occupied by the USR (unless the USR is set to NOSWAP). If the job needs more space than these three areas, an error message prints and then control returns to the Keyboard Monitor.

Figure 2–8: Background Job



Once the job passes the size tests, RUN loads memory locations 0 through 476 from the file, if they are not protected. To check for protection, RUN looks at the bitmap in RMON, and does not load any locations that are protected either by RMON or by another job.

Next, RUN loads all the memory locations from 500 through 776 from the file. This area is the default stack for the background job.

To load locations 1000 and up, RUN examines the *core control block*, called the CCB, which starts at location 360 in the job file. The CCB is a bitmap created by the linker in which each bit represents one block in the file. When the linker takes data out of the OBJ file to go into the SAV file, it sets the CCB bit for each block of the SAV file that actually contains code or data. For example, if you link a file with a base address of 2000, the locations in your file from 1000 through 1776 do not contain data, and therefore the linker does not set the corresponding bit in the CCB. RUN loads blocks from the file into memory only if the corresponding CCB bits for them are set.

If a block fits in memory in the area below KMON that is reserved for the background job, RUN loads it directly. If a block would overlay either KMON or the USR, RUN copies the block out to the disk file SWAP.SYS. This process continues until the entire file is loaded into memory, or into memory plus SWAP.SYS. SWAP.SYS is just large enough to hold the amount of program code that would overlay the KMON and the USR.

Finally, RUN (or START) jumps to RMON. If SWAP.SYS is in use, RMON reads its contents into memory, overlaying KMON and possibly the USR as well. Then RMON starts the program's execution. Figure 2–9 summarizes how the RUN command loads a job image into memory.

If SET EXIT SWAP is in effect when the program terminates, RMON reads KMON and the USR back into memory from the monitor .SYS file. The memory area up to the bottom of KMON contains the background job image. If the job overlaid KMON, the remainder of the job image is written out to SWAP.SYS. This procedure allows the Examine and Deposit commands to operate on the job image on disk, even though KMON has written over the job's locations in memory, and the RESTART command can restart the program.

2.1.7.2 R Command – The **R command** is similar to the RUN command. One initial difference, however, is that the file to be loaded must reside on the system device (SY:). The reason for this restriction is that the R command is not capable of loading another device handler in order to read the file.

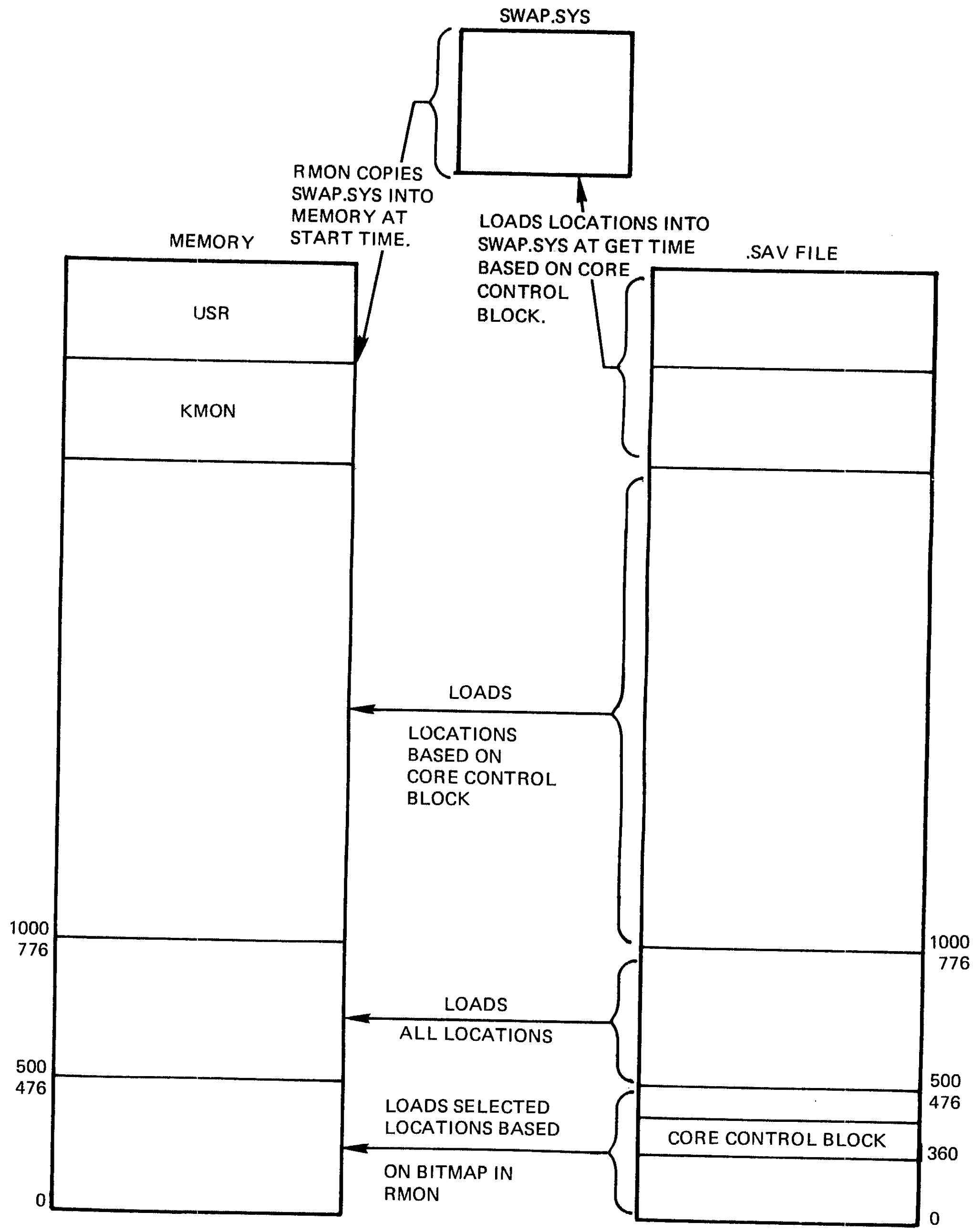
The R command loads memory locations 0 through 776 the same way the RUN command does. It has a different procedure for loading locations 1000 and up. The R command ignores the core control block in the file and it sets up parameters for RMON. RMON loads the rest of the file (up to its high limit; it does not load overlays) even if it overlays KMON and the USR. It ignores the file SWAP.SYS. Figure 2–10 summarizes how the R command loads a job image into memory.

If the job is a virtual job, the monitor creates for the job a virtual memory partition, a static window and static region definition block, and then sets up the user mapping registers. At this point it starts the job's execution. (See Chapter 4 for more information on virtual jobs.)

As with the RUN command, jobs (excluding virtual jobs) loaded with R use the SWAP.SYS file, if necessary, at program termination so that the Examine and Deposit commands function correctly. Note that if a job issues a .SETTOP request to lower its high limit before it exits, it may prevent the monitor from writing SWAP.SYS.

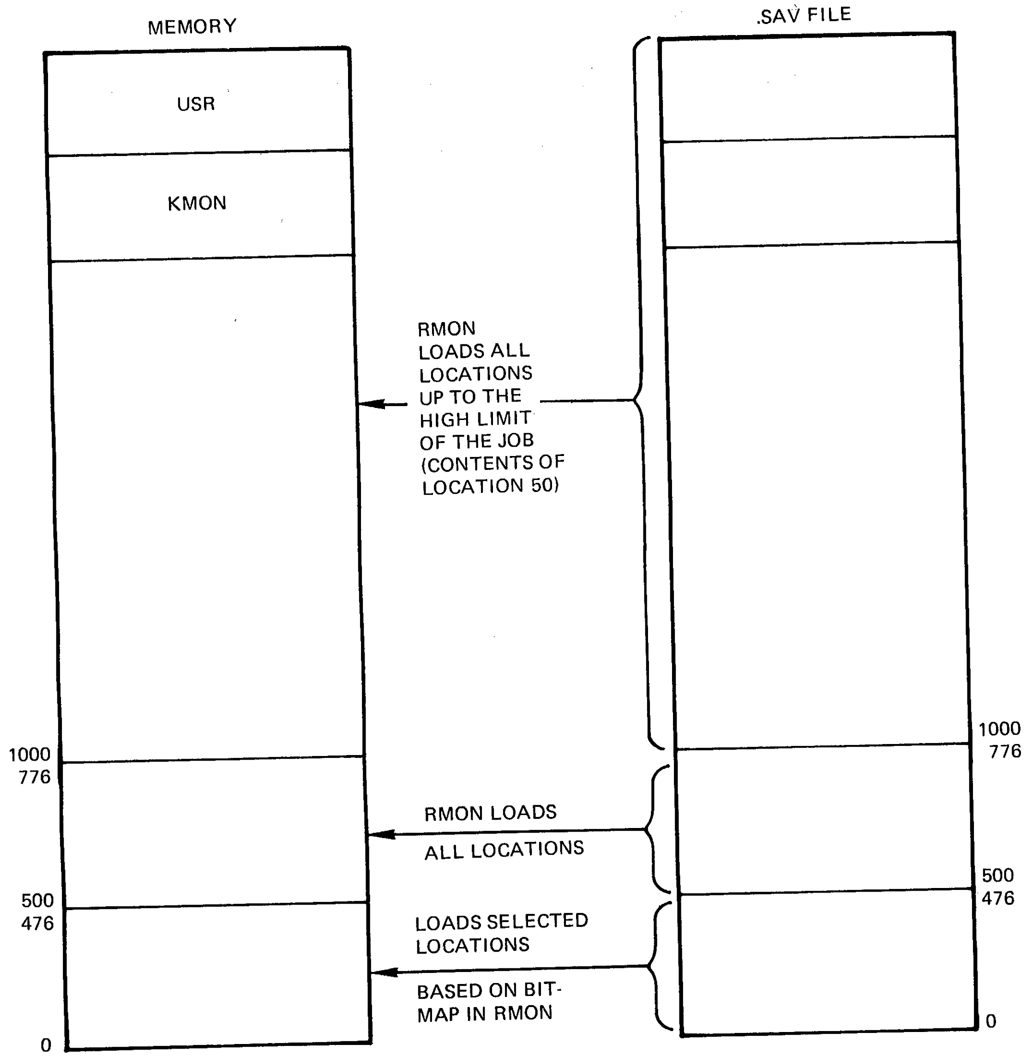
2.1.7.3 .CHAIN Request – The third way to load a job is to chain to it from another job. The first job issues the *.CHAIN* programmed request to do this. The second job can use information in memory locations 500 through 776 that was placed there by the first job. Consequently, the only difference between loading a job with the RUN command and starting a job by chaining to it is that chaining does not load memory locations 500 through 776 from the second file unless you set the chain bit in the JSW of the second file at assembly time.

Figure 2–9: RUN Command



Note that in XM systems, a virtual job cannot pass information when chaining to another job. In addition, you cannot chain to a virtual job. (See Chapter 4 for more information on virtual jobs.) Note also that chaining to a FORTRAN job does not preserve channel information from the previous job. This is because FORTRAN itself closes the channels and discards the impure area.

Figure 2–10: R Command



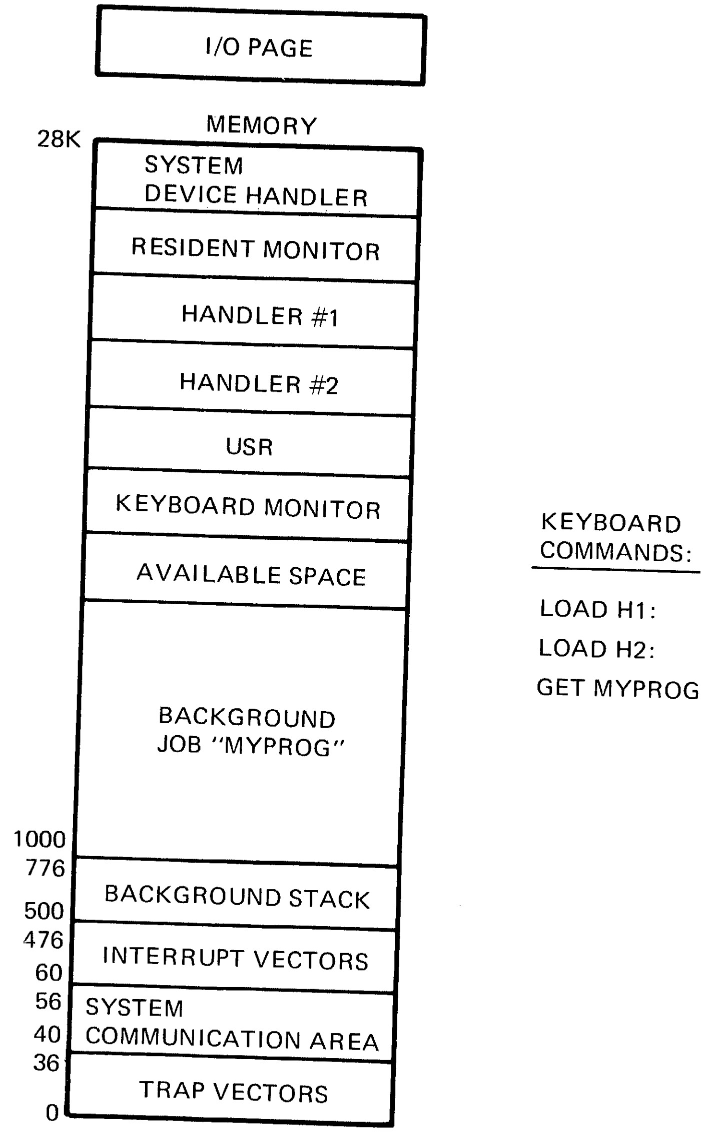
2.2 Dynamic Components

Dynamic components do not always load into fixed places in memory. Once loaded, some of them can continue to shift location based on the state of the rest of the system. The dynamic components and areas are as follows:

- Device handlers (device drivers) and free space
- Foreground and system jobs
- User Service Routine
- Keyboard Monitor

As you read about the rest of the dynamic components, you will also learn how the system manages free space in memory. You have already seen how the system device handler and the Resident Monitor load at the highest possible addresses, and how the background job begins loading at location 1000 and up. The strategy behind the way the system manages free memory is that it attempts to make the most space available for foreground and background application jobs.

Figure 2–11: SJ System with Two Loaded Handlers



2.2.1 Device Handlers and Free Space

Device handlers (drivers) are routines that provide the interface to the computer's hardware devices. The handlers **drive**, or **service**, peripheral devices and take care of moving data between memory and devices. Chapter 7 describes device handlers in greater detail.

RT–11 uses a dynamic scheme to provide memory space for loaded handlers, foreground jobs, system jobs, indirect file and command line expansion, and the display text scroller. Memory is allocated in the region above the

KMON/USR section and below RMON. If there is not enough memory in this region (initially, after the system is bootstrapped, there is none), memory is taken from the background region by "sliding down" the KMON and USR the required number of words.

When memory allocated in this manner is released, the memory area is returned to a singly-linked free memory list, the head of which is located in RMON. Any contiguous blocks are concatenated into a single larger block. A block found to be contiguous with the KMON/USR is reclaimed by "sliding up" the KMON/USR, thus removing the block from the list.

Figure 2–12: SJ System with One Handler Unloaded

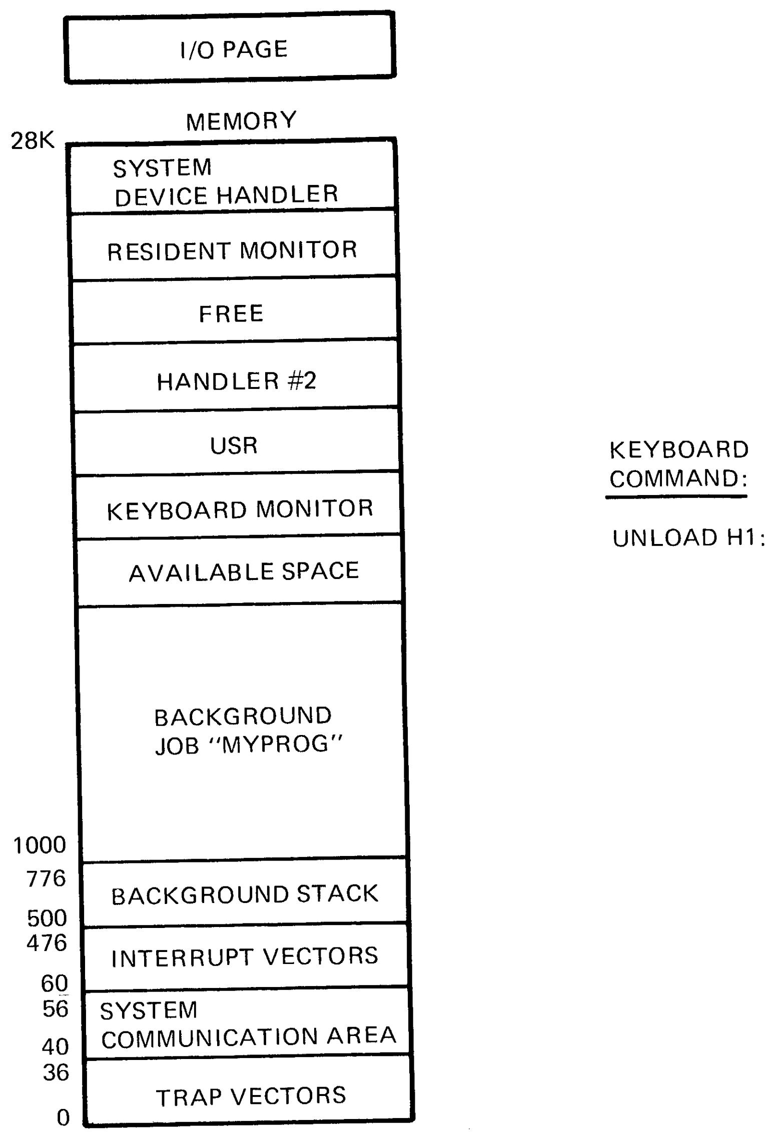


Figure 2–11 shows an SJ system with a small application job and two loaded device handlers. When you issue the LOAD monitor command the handler loads into the memory area just above the USR and KMON. The USR and KMON slide down in memory to provide the handlers with enough space,

leaving less space for the user program. The GT ON command is similar to the LOAD command, except that it specifically loads the VT11/VS60 video display handler. The GT handler is located in a Keyboard Monitor overlay instead of a .SYS file on a storage volume. Except for the fact that it is not stored as a separate handler file on a mass storage device, it functions the same as other handlers.

Once handlers are brought into memory, they do not move up or down, as the USR and KMON do. Figure 2–12 shows the system after the monitor UNLOAD command has removed one handler from memory. In the figure, the free space above handler #2 has not been reclaimed and is available for later use. A handler that is the same size as the empty space, or smaller, can be loaded there without causing any other components to move.

Figure 2–13: SJ System with Both Handlers Unloaded

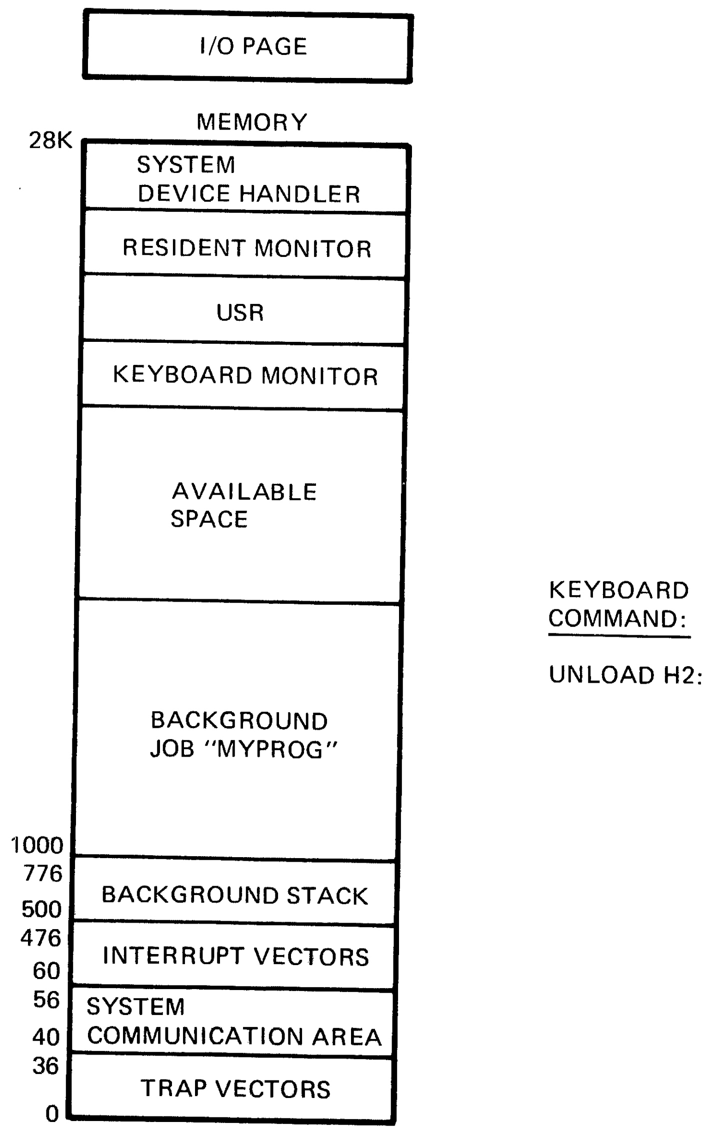
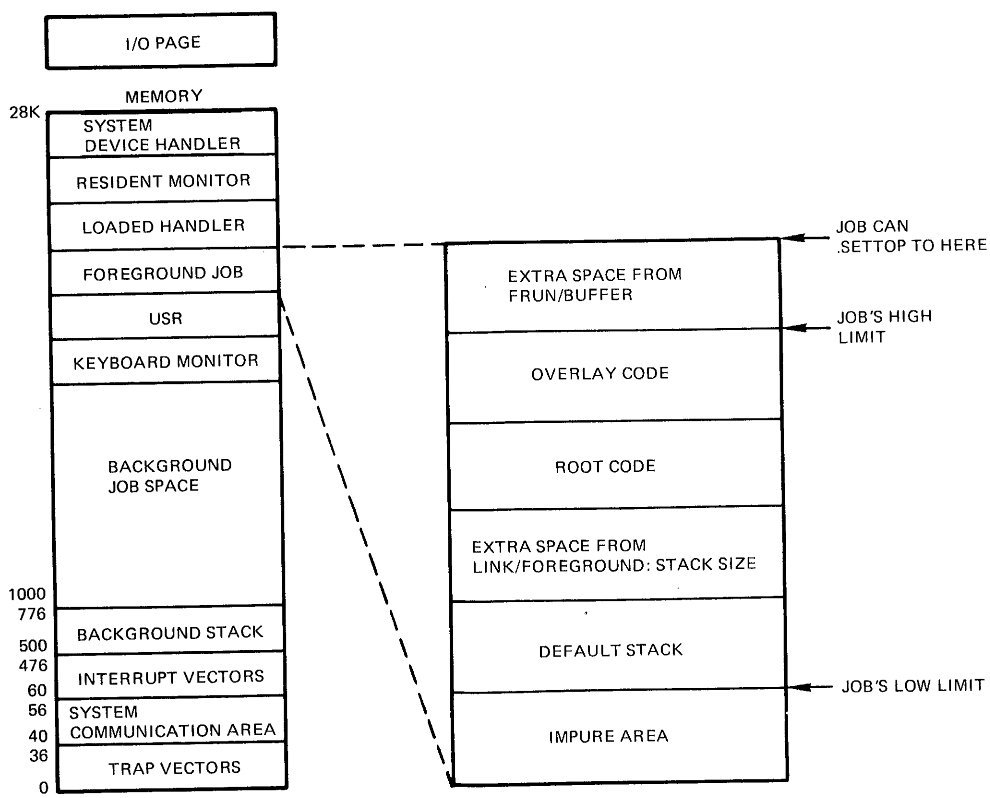


Figure 2–13 shows the system after the second handler was unloaded. This time there is free space directly above the USR (the space formerly occupied by the two handlers), so the USR and KMON slide up into it, making more space available for the user program. The GT OFF command is similar to the UNLOAD command, except that it specifically unloads the VT11/VS60 video display handler.

2.2.2 Foreground and System Jobs

In an FB or XM system, **foreground jobs** and **system jobs** are essentially identical. A system job is simply a special kind of foreground job that DIGITAL provides for you. The two RT–11 system jobs in the FB and XM environments are the error logger (ERRLOG) and the file queuing program (QUEUE). Figure 2–14 shows the general structure of a foreground job, as well as its relative location in memory. Handlers loaded after the foreground job are placed below it in memory, and above the USR. (See Chapter 3 for more information on foreground and system jobs.)

Figure 2–14: Foreground Job



2.2.2.1 Differences Between Foreground and Background Jobs – There are some significant differences between foreground and background jobs.

1. The impure area (described in Chapter 3) for the foreground job is located immediately below the job area itself. For a background job, the impure area is always in the Resident Monitor.

2. Another major difference is that a foreground job cannot dynamically change its memory allocation: the job is a fixed size. You can only change the size at FRUN time by using the /BUFFER:n option to increase the memory allocation. (Note that this option is ignored in XM systems for virtual .SAV files started with the FRUN or SRUN command.)

3. You must load all the handlers a foreground job needs before the job attempts to use them. A background job, on the other hand, can use the .FETCH programmed request to load a handler when it is needed.

4. For FB systems only, if the USR is swapped out and the foreground job needs it, the foreground job must allocate 2K words of program space for the USR to swap over. (See Section 2.2.3 for more information on the USR.)

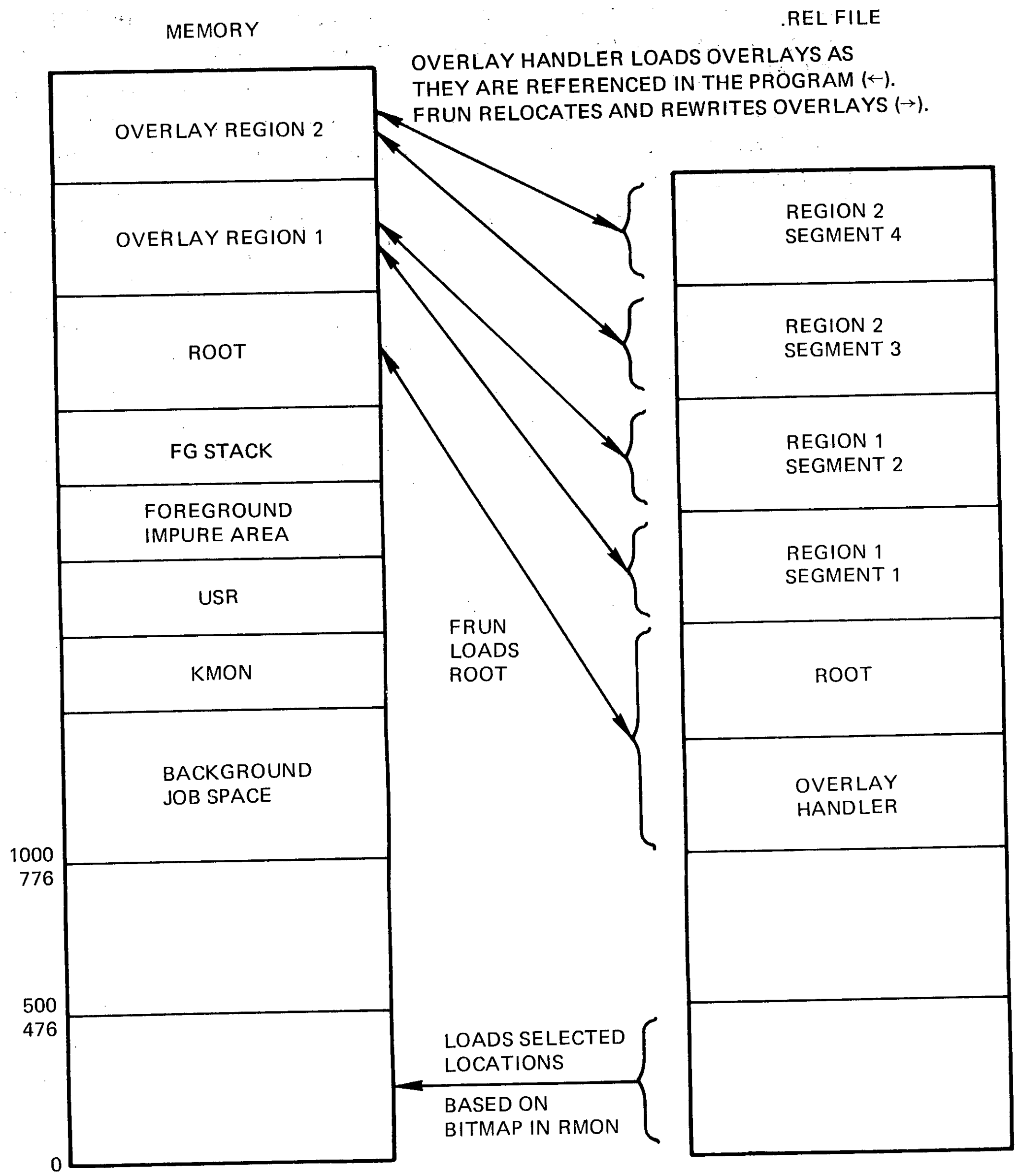
2.2.2.2 FRUN Command – The **FRUN command** loads a foreground program into memory and starts execution. The **SRUN command**, which performs the same functions for system jobs, is essentially identical. You can also use FRUN or SRUN to start a virtual .SAV job, since these jobs do not require relocation. (See Chapter 4 for more information on virtual jobs.) Before you start a job with FRUN, you must load all the handlers the job requires. You can use the FRUN/PAUSE option, load the handlers, then resume the foreground job. In any case, the handlers need to be loaded only before the job actually uses them.

FRUN first opens the .REL file or virtual .SAV file, reads its first block (locations 0 through 776), and determines how much memory the job requires. The job's total memory requirement is equal to the sum of the program itself (as indicated by location 50 in block 0 of the file), the size of the impure area, the extra space allocated with the FRUN/BUFFER:n command, and the extra space (if any) allocated with the LINK/FOREGROUND:stacksize command. If you do not allocate extra stack space, the default stack size is used. If there is not enough memory available to run the job, an error message prints and the monitor dot prints on the terminal.

Once FRUN gets the memory space the job needs, it sets up the job's impure area. FRUN also sets up the job context on the foreground job's stack, for FB systems, or in the job's impure area, for XM systems. So, when you first load a foreground job, it appears to be context-switched out. (See Chapter 3 for more information on context switching and other FB monitor functions.)

Next, FRUN loads the foreground main program into memory and relocates addresses in the root to reflect the current load address. Virtual .SAV files do not require relocation. If the job is overlaid, there is one more step before execution can begin. FRUN reads and relocates just the root of an overlaid program. Then it reads the overlay relocation information into a buffer. One by one, each overlay segment is then read into memory, relocated, and written back to disk. Finally, FRUN starts job execution. Figure 2–15 shows a summary of how the FRUN command loads a foreground job image into memory.

Figure 2–15: FRUN Command



2.2.2.3 Starting Foreground and System Jobs – Figure 2–16 illustrates the procedure DIGITAL recommends for starting up a system that has both system jobs and a foreground job. In general, group high in memory the device handlers and programs that you expect to be running for the longest time. Lower in memory, put the handlers and programs that you plan to run only for a short time. This organization enables the Resident Monitor to reclaim free memory when you unload programs and handlers that you no longer need.

In the example in Figure 2–16, the two handlers that the QUEUE program needs are loaded first, since the error logger and the QUEUE program are both intended to run as long as the system runs. (The QUEUE program needs handlers for the device to which it will copy files, as well as handlers for the devices on which those files are currently stored. The error logger needs no specific handler; it logs errors from any handler that calls it.) The SRUN command is used next to start the more important of the two system jobs (the error logger). Then the second system job (QUEUE) is started, also with SRUN. This ordering of system jobs gives the error logger higher priority by default than the QUEUE program. (Note that if it is not convenient for you to load the higher priority system job first, you can assign priorities to the system jobs with the SRUN/LEVEL:n command.) Lastly, the foreground job, which requires no other handler, is started with the FRUN command. In Figure 2–16 the foreground job, which always has the highest priority, is loaded last because it will only run for a short time before it is stopped, unloaded, and replaced by a different foreground job. After you stop a job by typing two CTRL/Cs or the ABORT command, you must use the monitor commands to unload it and replace it with another. RT–11 does not provide a way for one foreground job to automatically start another.

NOTE

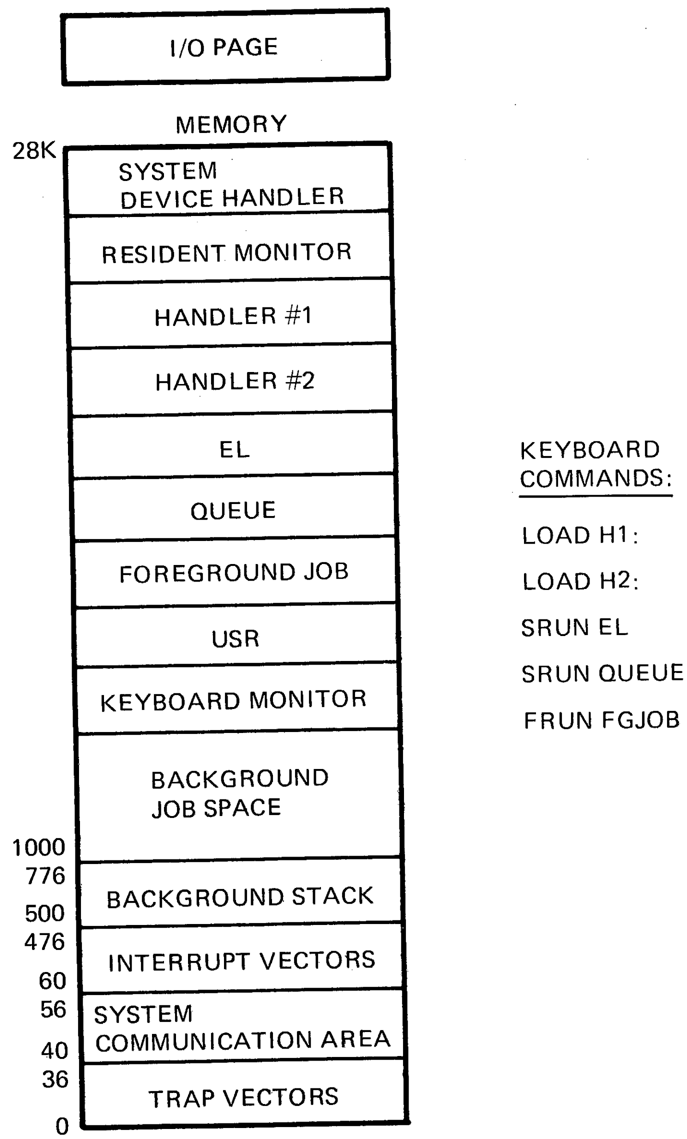
> Since the system job feature permits up to six system jobs to execute simultaneously, it is possible to have more than one copy of a specific job in memory at any one time. That is, you can use SRUN to start a job called STAT.REL, for example, and then use SRUN again to start up a second copy in memory of the same job from the same disk image, STAT.REL. However, this procedure is valid only for programs that are not overlaid.
>
> The disk image of an overlaid program is in constant use, since the relocated overlay segments are occasionally read into memory from the file. Thus, to execute multiple copies of overlaid programs, you must maintain separate copies of the programs on disk. For example, to run two copies of an overlaid program called STAT.REL, store an additional copy of the program on disk as STAT1.REL, and use SRUN to start both jobs.

2.2.2.4 Foreground Stack – The foreground job's stack is located immediately above the impure area. Its default size is 128 decimal bytes. You can change the size of the stack at link time by using the /FOREGROUND:stacksize option.

You can also change the location of the foreground stack. To do this, use the /STACK:n option at link time, and specify either an octal value for the stack pointer or a global symbol name. If you change the stack location, you are responsible for allocating space for the stack in your program.

Be careful not to let the stack overflow during execution. Since RT–11 neither checks for this error nor makes any attempt to correct it, the most likely result is that your program or the impure area will be corrupted.

Figure 2–16: FB System



2.2.2.5 Foreground Impure Area – The memory locations just below the foreground job area contain job-dependent information. This area is called the **impure area**, and its contents are maintained by the Resident Monitor. Chapter 3 lists the information contained in this area.

2.2.3 User Service Routine (USR)

The **User Service Routine** (USR) is the part of the RT–11 operating system that provides support for the RT–11 file structure. It contains instructions to:

- Fetch device handlers
- Get the status of device handlers

- Open existing files
- Create new files
- Delete and rename files
- Close files

In addition, the USR contains the Command String Interpreter (CSI) which interprets device, file, and option specifications. The default memory location for the USR is directly above the background area, or directly below the system jobs, foreground job, and loaded device handlers, if there are any. You can change this default location by setting an address in location 46 in low memory.

The USR does not always have to be resident in memory. In fact, it is designed to be swappable in order to make as much space as possible available for user jobs when they need it. In general, for SJ and FB systems, the USR is needed only when file-oriented operations are required. The USR is always resident in the XM monitor, so swapping is not a consideration for XM jobs.

2.2.3.1 Structure – The USR consists of two basic parts: the buffer area and the permanent code area. The first section, which is two blocks long, contains code when the USR is brought into memory. This area also serves as the buffer in which the USR stores a device directory segment. The second section contains permanent code. Figure 2–17 shows an overview of the USR's structure and its memory location in an SJ system.

The first routine in the USR buffer section consists of initialization code to relocate pointers in the USR and KMON. This relocation code becomes active the first time the USR is entered after it is brought into memory. It relocates internal pointers in the USR that point to the Resident Monitor and to other important locations within the USR. If the USR was called from KMON, it also relocates pointers to RMON within KMON.

For SJ systems, the next segments of code are:

1. The EMT 376 processor, which contains the text and the routines to print fatal monitor error messages.
2. Code that processes the .CDFN programmed request.
3. Routines to handle the .SRESET and .HRESET programmed requests.

For FB and XM systems, the next section of code handles the .EXIT programmed request. The last segment of code in the buffer area processes the .QSET programmed request for SJ and FB monitors. A small amount of scratch space takes up the remainder of the two-block buffer area.

Following the buffer area is the USR's permanent code which starts at offset 2000 from the beginning of the USR. The permanent code consists of routines that process the following programmed requests:

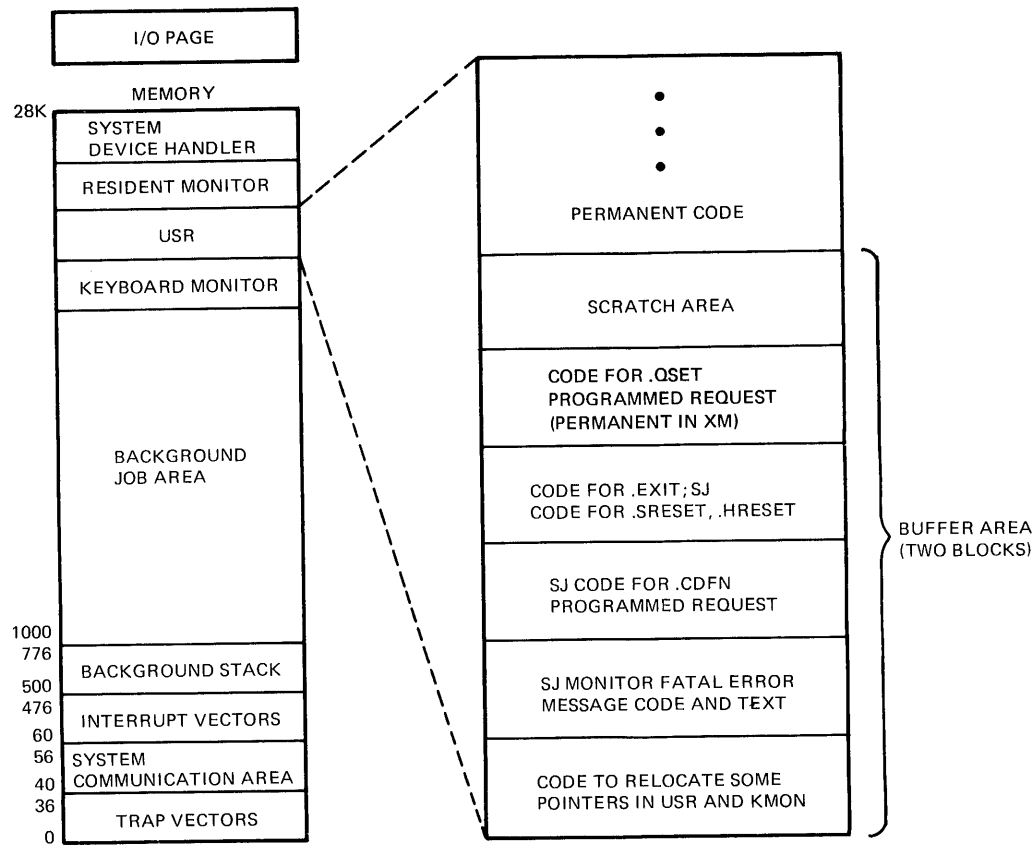
.DELETE	.LOOKUP
.FETCH	.RENAME

.CLOSE .DSTATUS

.ENTER .QSET (for XM only)

The Command String Interpreter occupies the end of the USR, where the .GTLIN, .CSIGEN and .CSISPC programmed requests are processed.

Figure 2–17: USR



2.2.3.2 Execution – The general flow of execution in the USR is straightforward. When a fresh copy of the USR is brought into memory, its buffer area contains the code described in the previous section. When a program issues a USR programmed request, the first code to execute is the relocation code. This code then calls the routine to process the particular request that was issued. If the USR stays in memory, subsequent USR requests go directly to the routines that process them. The initialization code is not called again.

Usually, a USR request requires a device directory segment. If the correct segment is already in the USR buffer, the USR does not read in a fresh copy of that segment. If the correct segment is not in memory, or if the USR has no segment at all, the USR reads the directory segment into its buffer. When it does this, the USR stores two words of information in the Resident Monitor fixed offset area. BLKEY, at offset 256, contains the number of the directory segment currently in the USR buffer. CHKEY, at offset 260, contains the device's unit number in the high byte, and an index into the monitor device tables in the low byte.

It can be useful to you to know under what circumstances the USR reads in a new directory segment. The following conditions cause the USR to read in a new directory segment:

1. Anything that causes the USR to swap out. When a fresh copy of the USR is brought into memory, it will have no directory segment in its buffer and will be forced to read one from a device.

2. Executing code in the buffer area. Since the code to process some programmed requests is located in the USR buffer area, attempting to process one of those requests always causes a fresh copy of the USR to be brought into memory. The requests that cause this to happen are:

   .CDFN (for SJ)
   .SRESET (for SJ)
   .HRESET (for SJ)
   .QSET (for SJ and FB)
   .EXIT (if your program was loaded over any part of KMON)

3. An SJ monitor error occurs. This situation requires the EMT 376 processor code, which is located in the USR buffer area and causes a fresh copy of the USR to be read into memory.

4. Issuing an .ENTER programmed request. This always causes the USR to read a fresh directory segment.

5. Issuing a .LOOKUP programmed request with a different device or file specification from the previous .LOOKUP. Note that doing a .LOOKUP with the same device specification as the previous .LOOKUP does not necessarily cause the USR to read in a fresh copy of the same directory segment. This is why you cannot remove a volume from a given device unit, replace it with another volume, and expect the USR to have the new volume's directory segment in memory. However, in this situation, you can force the USR to read a directory segment from the new volume by locking the USR to gain exclusive use of it, storing a value of 0 in BLKEY (RMON fixed offset 256), and then issuing a .LOOKUP programmed request with the same arguments as the previous .LOOKUP. Clearing BLKEY causes the USR to "forget" the current directory segment and read a fresh one from the new volume.

2.2.3.3 Swapping Considerations — Because the USR does not always have to be resident in memory for SJ and FB systems, you have a variety of options to consider when you design an application program. You can keep the USR in memory at all times (the simplest case), or you can arrange to have the USR swap into memory only when your program needs it. The latter procedure permits your program to use an extra 2K words of memory when the USR is swapped out. The guidelines that follow can help you design programs that handle the USR efficiently.

In XM systems, the USR is always resident (that is, SET USR NOSWAP is always in effect). Of the sections that follow, only those that describe a resident USR are meaningful for programs in XM.

NOTE

In general, the burden of USR swapping should be undertaken by the program, not by the operator who runs it. SET USR NOSWAP is useful to override the default action of programs outside an operator's control (such as FORTRAN), but its use requires operators to understand internal programming details – a requirement that should be avoided if at all possible.

Keeping the USR Resident in an SJ System

In an SJ system, the normal location for the USR is just below the Resident Monitor and loaded device handlers (see Figure 2–17). If your program does not need the space the USR occupies, you can force the USR to remain resident while your program is executing by issuing the monitor SET USR NOSWAP command before you run the program. In any case, if the space is not needed, the USR does not swap. Note that the USR can still slide up or down in memory, as Section 2.2.1 describes.

For a FORTRAN main program, you can keep the USR resident by using the FORTRAN/NOSWAP command (or the /U compiler option) at compile time. This forces the USR to remain resident while the program is executing. You cannot use this option if your FORTRAN programs require the extra 2K words of memory.

Keeping the USR resident means that 2K words less memory is available to your program. However, the directory operations involved in file opening and closing and in program loading will be faster because this arrangement eliminates swapping and disk I/O. In addition, the program will have a much simpler design. To keep the USR resident, a MACRO program should avoid issuing a .SETTOP request for memory above the base of the USR.

Remember that even though the USR is set to NOSWAP, there are some programmed requests that can cause a fresh copy of the USR to be brought into memory. For an SJ system, these requests are .CDFN, .SRESET, .HRESET, .EXIT, and .QSET. If the USR is swappable and if the background program issues a .SETTOP request for memory above the base of the USR, th USR loads into the area specified by the contents of location 46 in low memory. If location 46 contains 0, as it should when you intend to keep the USR resident, the USR loads in its usual place, below RMON. However, if for any reason you move a different value to location 46 and then execute one of the requests that loads a fresh copy of the USR, the USR will then load into the area you specified. If you execute a program that keeps the USR resident, the monitor ignores the contents of location 46.

Allowing the USR to Swap with an SJ MACRO Program

The only reason to allow the USR to swap in an SJ system is to gain access to the extra 2K words of memory that swapping makes available. To enable USR swapping, make sure that the SET USR SWAP command is in effect. (This is the default condition.)

A MACRO program gains access to the 2K words of memory because its high limit requires it, or because it does a .SETTOP to an address within the USR area. (Refer to Figures 2–9 and 2–10 for a summary of how the RUN and R commands load programs that overlay the USR area.) When the program issues a programmed request that requires the USR, the part of the program that occupies the USR area is written out to SWAP.SYS, and a fresh copy of the USR is brought into memory from the monitor file on the system volume. Location 46 should contain a value of 0 if you want the USR to swap into memory at its default location. If you want it elsewhere, put the starting address into location 46 during your program's initialization routine. When the programmed request completes, the part of the program in SWAP.SYS is copied back into memory, overlaying the USR. This sequence of events occurs for each programmed request that requires the USR, even if your program issues two or more requests in a row.

To make more efficient use of the USR, your program can issue the .LOCK programmed request before any other USR requests. This swaps part of your program out, reads the USR in, and returns to your program. After this, the USR remains in memory at the location you specified in location 46 (if any). You can now issue a number of USR programmed requests and avoid the overhead of USR swapping. When your program next needs the 2K words of space, use an .UNLOCK request to release the USR.

When the USR is swappable, it is important that you put it in a safe place in your program. This means that the area the USR will swap over must not contain code or data that will be needed at the same time the USR is in memory. The following is a list of code and data that must not be overlaid by the USR:

- Device block and/or CSI or .GTLIN file description string for the current request
- Active device handlers
- Active completion routines
- Active interrupt service routines
- Active I/O buffers
- Queue elements from .QSET
- I/O channels from .CDFN
- The program stack
- The memory list from .DEVICE
- Trap service routines from .SPFA and .TRPSET
- Code executed between the .LOCK and .UNLOCK requests

You can control USR swapping by careful use of the .SETTOP request. A typical practice that many system utility programs use is to issue a .SETTOP request to obtain space up to the base of the USR. The programs then perform all their USR operations. Finally, the programs issue an additional .SETTOP request to obtain as much memory as possible, if necessary.

Another situation to be aware of occurs when a program issues a .SETTOP request for more memory than is available. In this case, the program is given only the amount of memory that is available. After issuing a .SETTOP request, a program must always use the value returned in R0 (or location 50 in low memory) as the true high limit of the program. For example, a program can issue a .SETTOP request for memory above the base of the USR when the USR is set to NOSWAP. However, the value returned to the program as its true high limit is just below the base of the USR.

Allowing the USR to Swap with an SJ FORTRAN Program

As with a MACRO program in an SJ system, the only reason to permit the USR to swap with a FORTRAN program is to gain access to an additional 2K words of memory. The USR normally swaps over the FORTRAN OTS (Object Time System). However, problems occur when the FORTRAN OTS and the program together are less than 2K words long. In this case, the USR swaps over the program's impure data area, with unpredictable results. (Since this error is frequently made by inexperienced programmers, setting the USR to NOSWAP and retrying a program is the first thing you should do when debugging a FORTRAN program that doesn't execute properly.) And, unlike MACRO, USR swapping does not depend on your program's high limit – that is, if the USR is allowed to swap, it most definitely will swap. So, do not permit USR swapping unless your program really needs the extra memory. To enable swapping for a FORTRAN program, make sure the SET USR SWAP command is in effect, and eliminate the /NOSWAP or the /U option at compile time.

You have already read about the role that location 46 plays in determining where the USR will swap. For a FORTRAN program, the FORTRAN OTS places a value in location 46 to set up the USR swapping location. It is important to understand where and how the USR swaps so you can design your FORTRAN program correctly.

The FORTRAN compiler examines the sections of your program and sorts them based on two major attributes: read-only versus read-write, and pure code versus data. Generally, program instructions are read-only, and program data is read-write. If you use assembly language routines, use the same p-sects as the FORTRAN compiler. That is, place pure code and read-only data in section USER$I, and impure data in USER$D. The compiler forces p-sects into the order shown in Table 2–7.

This ordering collects all pure sections before impure data in memory. The USR can safely swap over sections OTSI, OTSP, SYS$I, USER$I, and $CODE. Figure 2–18 shows the arrangement of components when a FORTRAN program is loaded into memory. The global symbol $$OTSI marks the start of the pure code area. The global symbol $$OTSC marks its end, and the beginning of the impure data area. FORTRAN puts the value of $$OTSI into location 46, and the USR swaps into memory starting at that address, thus overlaying the first 2K words of your program.

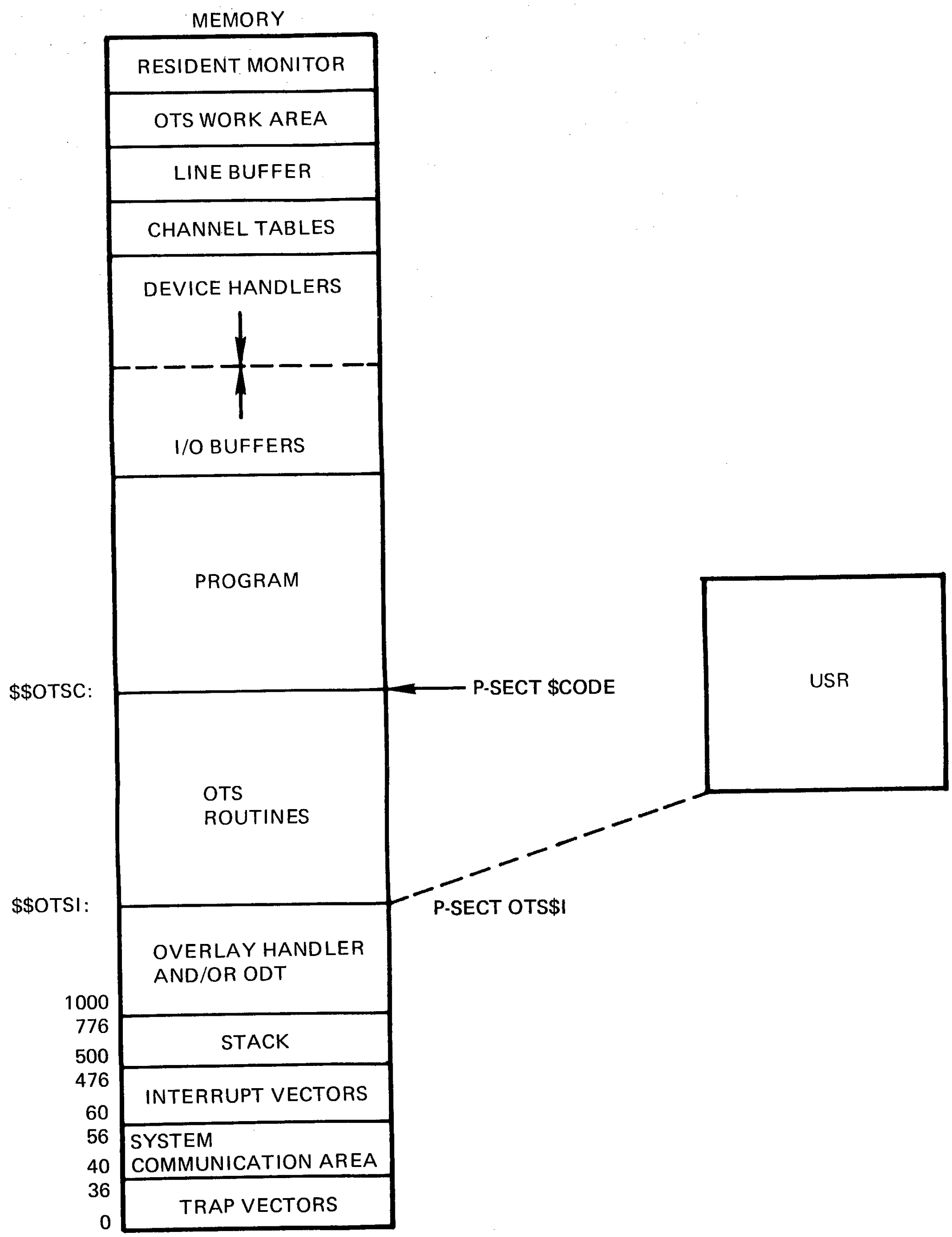
Table 2–7: P-sect Ordering for FORTRAN Programs (Low to High Memory)

Section Name	Attributes	Contents
OTS$I	RW,I,LCL,REL,CON	Pure code and data for the OTS initialization module
OTS$P	RW,D,GBL,REL,OVR	Pure tables of addresses of other OTS modules
SYS$I	RW,I,LCL,REL,CON	RT–11 SYSLIB routines
USER$I	RW,I,LCL,REL,CON	Program's pure code and read-only data
$CODE	RW,I,LCL,REL,CON	Start of program; read-write data
OTS$O	RW,I,LCL,REL,CON	OTS routines sensitive to USR swapping
SYS$O	RW,I,LCL,REL,CON	
$DATAP	RW,D,LCL,REL,CON	Constants
OTS$D	RW,D,LCL,REL,CON	Pure data referenced by the OTS modules
OTS$S	RW,D,LCL,REL,CON	Scratch storage referenced by the OTS modules
SYS$S	RW,D,LCL,REL,CON	
$DATA	RW,D,LCL,REL,CON	Local variables
USER$D	RW,D,LCL,REL,CON	Program's impure data
.$$$$.	RW,D,GBL,REL,OVR	Blank COMMON
Named COMMON blocks	RW,D,GBL,REL,OVR	

As with a MACRO program, your FORTRAN program should not have certain instructions or data in the area where the USR will swap. As a general rule, the following items should not be in the USR swap area:

- Routines that request USR functions (such as IENTER and LOOKUP)
- Data structures for USR requests
- Interrupt service routines
- Completion routines
- Data areas for interrupt service routines and completion routines

Figure 2–18: A FORTRAN Program in Memory



The FORTRAN system itself must also be concerned with USR swapping and its inherent restrictions. For example, the p-sect OTS$O contains the FORTRAN OTS routines to open files. This p-sect follows $CODE in the p-sect ordering. If the start of OTS$O is within 2K words of $$OTSI, the essential information for the file operation is stored on the job stack before the USR swaps over the code in OTS$O.

The best way to make sure that the USR swaps into a safe place in your FORTRAN program is to examine the link map to determine if the USR will swap over restricted sections. That is, see if the first 2K words above $$OTSI can be overlaid safely. If not, relink the program and change the order of object modules and libraries you specify to the linker. One problem is caused by using SYSLIB routines that place important USR data in the lower 2K words of the job image. An example is the IFETCH routine, which uses a device block in the program. The USR swaps over the device block just before it is used, causing an error. To avoid a situation like this, do not set up device names as constants for a SYSLIB call. Instead, use DATA-initialized variables. This ensures that the information will be stored high enough in the job image to avoid being overlaid by the USR.

For more information on this topic, see the *RT–11/RSTS/E FORTRAN IV User's Guide* and the *PDP–11 FORTRAN Language Reference Manual.*

Keeping the USR Resident in an FB System

As with an SJ system, the easier way to deal with the USR in an FB system is to keep it resident. Use the SET USR NOSWAP command, or the /NOSWAP (/U) FORTRAN compiler option. This arrangement is suitable if the background, foreground, and system jobs have enough memory. The USR is brought into memory at its usual place, just below any loaded handlers and below the foreground job and it remains in memory during program execution. Neither job has to allocate program space for the USR, and programs execute faster without the overhead of USR swapping and disk I/O.

The important issue in an FB system with the USR resident is determining which job should have control of the USR. Because only one job can use the USR at a time, both jobs must be aware of sharing this resource. Since a program in an SJ system can lock the USR in order to process a number of USR programmed requests, in an FB system, either the background job or the foreground job can lock the USR to gain exclusive use of it.

The .LOCK request gives ownership of the USR to one job. The .UNLOCK request releases the USR, making it available for the other job. The request .TLOCK can determine whether or not the other job has exclusive ownership of the USR. It permits a program to try for a .LOCK, but to continue with execution if the attempt fails.

The LOCK/UNLOCK system permits one job to lock out another for a considerable length of time. During a lockout, interrupt service and completion routines can run, but not mainline code. This could cause serious difficulties in a real-time foreground program. There are some ways to minimize or eliminate this lockout problem:

1. Be sure to separate USR operations from real-time operations.
2. Avoid using devices with slow directory operations, such as cassette, magtape, and DECtape II.

3. Organize your real-time foreground program so that real-time operations are in interrupt service routines and completion routines and will not be affected if the mainline code is locked out with a pending USR request.

Typically, a real-time foreground job can be organized in three parts: an initialization phase, which opens all required channels and begins real-time operations; a real-time phase, which does interrupt service and I/O operations; and a completion phase, which stops real-time activity and closes the channels. With this arrangement, the background program can perform USR operations during the real-time phase without locking out the foreground. The foreground program can use .LOCK and .UNLOCK to prevent interference from the background job during initialization and completion phases.

Swapping Considerations for Background Jobs

When either the background job or the foreground job needs the extra 2K words of memory that swapping the USR provides, both jobs must be concerned with USR swapping. The general concerns for background jobs are those listed in the previous sections.

The easiest approach for the background job is to swap the USR into its default location, the highest 2K words of program space. If this is not convenient for any reason, the background job can select any other contiguous 2K words of program space. In this case, it must also put the starting address of the USR swap area into location 46 in the system communication area. This location is context-switched in the FB system, so it always contains the correct value for the job that is currently executing.

The background job must not place any USR-sensitive code or data in the area where the USR will swap. In addition to the list in the section Allowing the USR to Swap with an SJ MACRO Program, the following items must not be in the USR swap area:

- Memory list from the .CNTXSW request
- Active message buffers
- Code containing the .LOCK or .TLOCK requests

You must also be careful that the background job does not lock the USR for an unreasonable length of time so it can block the foreground job from running. If you lock the USR in a background job, remember to unlock it as well.

Swapping Considerations for Foreground Jobs

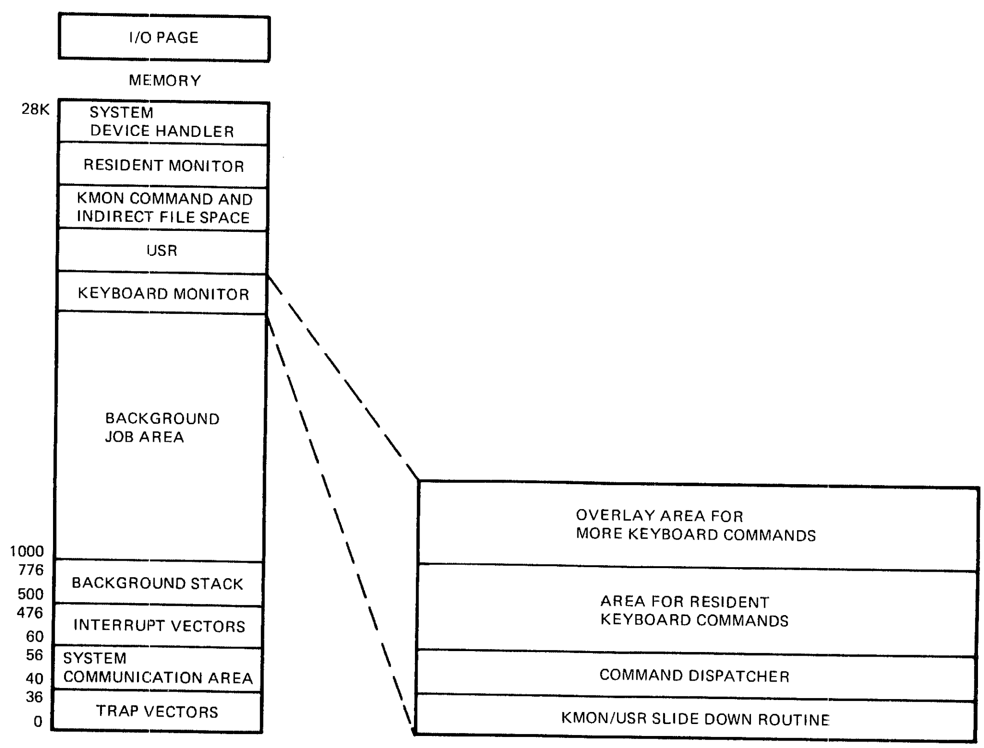
If the background job issues a .SETTOP that causes the USR to swap, or if the background job is large enough to force the USR to swap, the foreground job must be concerned with USR swapping. However, while the background job can simply allow the USR to swap into its default position (the highest 2K words of the background job area), the foreground job has no default location for the USR. It must allocate 2K words within its program bounds in which to swap the USR – space that must not contain any USR-sensitive code or data. The foreground job must also place the starting address of that space in location 46 in the system communication area. This location is context-switched during normal foreground/background execution, so it always contains the correct swapping address for whichever program is currently executing.

The foreground program could also be concerned with sharing the USR with the background job. The .LOCK/.UNLOCK requests can give the foreground job exclusive ownership of the USR to prevent interference by the background job. The foreground job should avoid keeping the USR permanently locked, which sometimes happens strictly because of a programmer's oversight.

2.2.4 Keyboard Monitor (KMON)

The **Keyboard Monitor** (KMON) is the part of the RT–11 system that provides the communication link between you at the console terminal and the rest of the RT–11 system. Keyboard monitor commands permit you to assign logical names to devices, load device handlers, run programs, control foreground/background operations, control system jobs, invoke indirect command files, and examine or modify memory locations. KMON is brought into memory when the background job completes. When KMON is in memory, the USR is also present directly above it.

Figure 2–19: Keyboard Monitor



The Keyboard Monitor consists of a root segment and a number of overlays that contain the command processors. KMON runs as an ordinary background job, in user mode. The root segment is contained in the p-sect RT11. See Table 2–6 for a summary of all monitor p-sects.

When KMON interprets a keyboard monitor command that you type at the terminal, it expands the command text into an internal indirect file. For example, the command COPY MYFILE DL:MYFILE(RET) expands internally into:

```
R PIP (RET)
DL:MYFILE=DK:MYFILE (RET)
^C
```

KMON stores this internal indirect file in the *command expansion buffer area*. KMON creates space in memory for this buffer area immediately above the USR. When KMON and the USR slide up or down in memory, the command buffer spaces moves with them. Figure 2–19 shows the Keyboard Monitor in memory.

Chapter 1 of the *RT–11 System User's Guide* gives an overview of KMON command processing. The *RT–11 System Generation Guide* describes how to remove individual commands or groups of commands from a system you create through the system generation process. If you are interested in modifying KMON itself to change the monitor command set, obtain the microfiche listings of the commented sources. Extensive comments in KMON sources outline the procedure for adding new commands and changing existing commands. Note that because the procedure is very complex, DIGITAL does not recommend modifying the keyboard monitor commands. Instead of modifying KMON, use the CCL (Concise Command Language) or UCL (User Command Linkage) interfaces to create new commands. The procedures for doing this are outlined below.

2.2.4.1 Adding New Commands Through CCL — If KMON does not recognize the first word of an input line as a valid KMON command, it tries to treat the input line as a CCL command by searching for a program of that name on SY: and running the program. If a program is found, KMON passes the remainder of the command line to the program in the CSI input buffer as a CSI command string, followed by a ^C. The general format of a CCL command is:

command <sp> field1<sp>field2

or

command <sp> csistring

If the first form is used, KMON converts it to the second form by reversing the fields and inserting an equal sign:

command <sp> field2 = field1

For example, you might type:

```
PIP A=B
```

or

```
PIP B A
```

Both forms are equivalent to typing:

```
.R SY:PIP
*A=B
*^C
```

If the first word on the line is more than six characters, characters after the first six are ignored. *field1* and *field2* can contain multiple file names, separated by commas. If you have an application program on SY: named EVALUA.SAV to evaluate certain collected data and print a report, you could type:

```
EVALUATE RK3:DATA16.DAT, RK1:DATA03.DAT LP:
```

This is equivalent to:

```
.R SY:EVALUA
LP:=RK3:DATA16.DAT, RK1:DATA03.DAT
*^C
```

2.2.4.2 Adding New Commands Through UCL — RT–11 V5 also supports User Command Linkage (UCL) as a SYSGEN option. The SYSGEN conditional is U$CL. If UCL support is enabled, KMON first checks to see if the first word of the line is one of the defined KMON commands, such as COPY. If not, KMON tries, using the CCL conventions outlined above, to find and run a program of that name. If that also fails, KMON looks for the user program SY:UCL.SAV and runs it if present. KMON passes as ASCII text, in the chain area starting at location 512, the entire command line including the first word, to UCL.SAV. Location 510 contains the number of bytes in the command line. Locations 500 through 507 of the chain area are not used.

The user-written program UCL.SAV can interpret and expand the command line passed to it in any way that it wants. UCL.SAV can perform the operations required by the command. UCL.SAV can reformat and pass the command to another program by doing a .CHAIN, or UCL.SAV can create a new command line and pass the new command to KMON by doing a normal or special chain exit. For example, you could type:

```
BUILD MYPROG
```

UCL.SAV might expand the command into the following series of commands:

```
R MACRO
MYPROG,LP:/C=MYPROG
^C
R LINK
MYPROG=MYPROG
^C
RUN MYPROG
```

These commands could then be passed back to KMON by doing a normal chain exit or a special chain exit. Refer to Section 2.1.2.2 for information about normal and special chain exits.

The default device for UCL.SAV is stored as a Radix–50 word at monitor location ..UCLD; this can be changed to another device name if desired. The default name for the UCL command processor (initially UCL.SAV) is stored as Radix–50 words at monitor location ..UCLF; this may also be changed to another name if desired.

2.3 Sizes of Components

Table 2–8 shows the sizes of some of the components in the distributed RT–11 systems.

Table 2–8: Sizes of Distributed Components in Memory

Monitor	KMON	USR	RMON
BL (base-line)	20000 octal bytes	2K words	1857 decimal words
SJ	20000 octal bytes	2K words	1996 decimal words
FB	20000 octal bytes	2K words	4220 decimal words
XM	21000 octal bytes	2K words	4500 decimal words

If you are not using a distributed system, and you need to know the sizes of the components, you should follow the guidelines in the next few sections.

2.3.1 Size of the USR

For SJ and FB systems, the size of the USR is always 2K words. For XM systems the USR, which is always resident, is somewhat larger. Your running program can determine the exact size of the USR by examining RMON fixed offset 374, USRAREA, which contains the size of the USR in bytes. You can also determine the size of the USR by issuing the monitor commands SET USR NOSWAP and SHOW MEMORY.

2.3.2 Size of KMON

The size of KMON is the same as the size of the p-sect RT11. Examine the link map that resulted from the system generation for your system to obtain this value.

2.3.3 Size of RMON

To determine the size of RMON, issue the SHOW MEMORY monitor command. This command prints the base address of RMON and its size in decimal words.

2.3.4 Size of Device Handlers

The size of each device handler, in bytes, is contained in location 52 of the handler's .SYS file. You can also obtain this value by issuing a .DSTATUS programmed request on the device from a running program or by issuing the SHOW MEMORY monitor command, which reports the sizes of all loaded device handlers.

Chapter 3
Resident Monitor

The main purpose of the *Resident Monitor* (RMON) is to provide services to running programs and to the Keyboard Monitor. The services include fielding traps and interrupts, providing the programmed requests, and acting as the central manager of the device-independent I/O system. In a multi-job system, the monitor also arbitrates the demands of up to eight jobs for processor time.

This chapter describes the functions of the Resident Monitor that are generally common to all RT–11 systems. It provides information on the monitor's terminal service for a single console teminal. (See Chapter 5 for information on multi-terminal systems.) It also describes how clock interrupts are handled and explains how timer support is implemented. The queued I/O system is discussed, scheduling for multi-job systems is described, and the system job feature is introduced. Lastly, information on the Resident Monitor's data structures is provided.

3.1 Terminal Service

RT–11 provides terminal service through the Resident Monitor. Terminal service is always resident, and it is part of RMON itself. Because of the way RT–11 implements terminal service, no handler is involved in the interaction between you at the terminal and the running system. Thus, terminal service is distinct from the services provided through the TT handler. (The TT handler implements .READ and .WRITE programmed requests for the console terminal.) It is designed to be a good interface between a person and the system, rather than an interface between a peripheral device and the system.

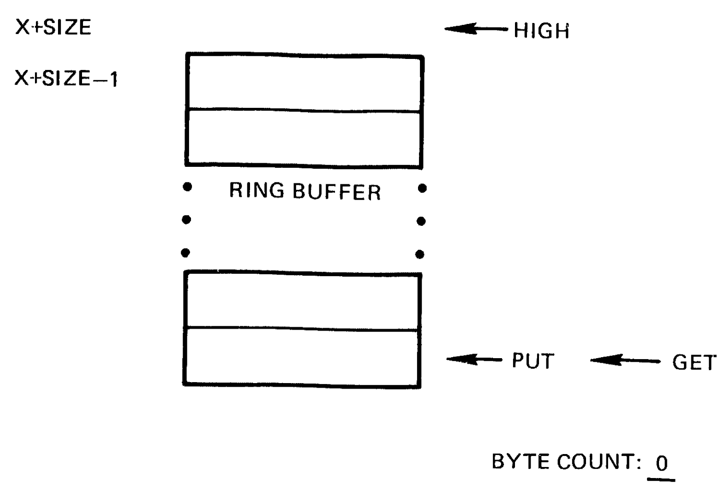
As part of the resident terminal service, RMON provides special programmed requests for terminal I/O. Because it uses ring buffers to implement the terminal service, RMON provides support for line-by-line editing. The terminal input interrupts are always enabled, which means that you can get the system's attention at any time by typing CTRL/C, CTRL/B, CTRL/F, and so on. You can also type ahead to the system without losing characters.

The ring buffers are the heart of the terminal service implementation. In SJ, one input ring buffer and one output ring buffer are located in RMON. For FB and XM systems, each job has its own set of ring buffers located in its impure area. The ring buffers store text in a buffer zone between you at the terminal and a running program in memory. The default size of the input ring buffer is 134 decimal bytes; the default size of the output ring buffer is 40 decimal bytes.

3.1.1 Output Ring Buffer

An output ring buffer consists of the buffer area, three pointers, and a byte count. The buffer, or ring, itself is a block of bytes reserved for storing characters. Two of the three pointers store and retrieve characters. The PUT pointer marks the location where the next character will be stored and is used by the programmed requests that fill the buffer, such as .TTYOUT, .TTOUTR, and .PRINT. The GET pointer marks the next character to be retrieved and is used by the output interrupt service routine that sends characters to the terminal. The third pointer, HIGH, points to the first memory location past the buffer. Lastly, the monitor maintains a byte count for the number of characters currently in the buffer. Figure 3–1 shows an output ring buffer in memory just after the system was bootstrapped.

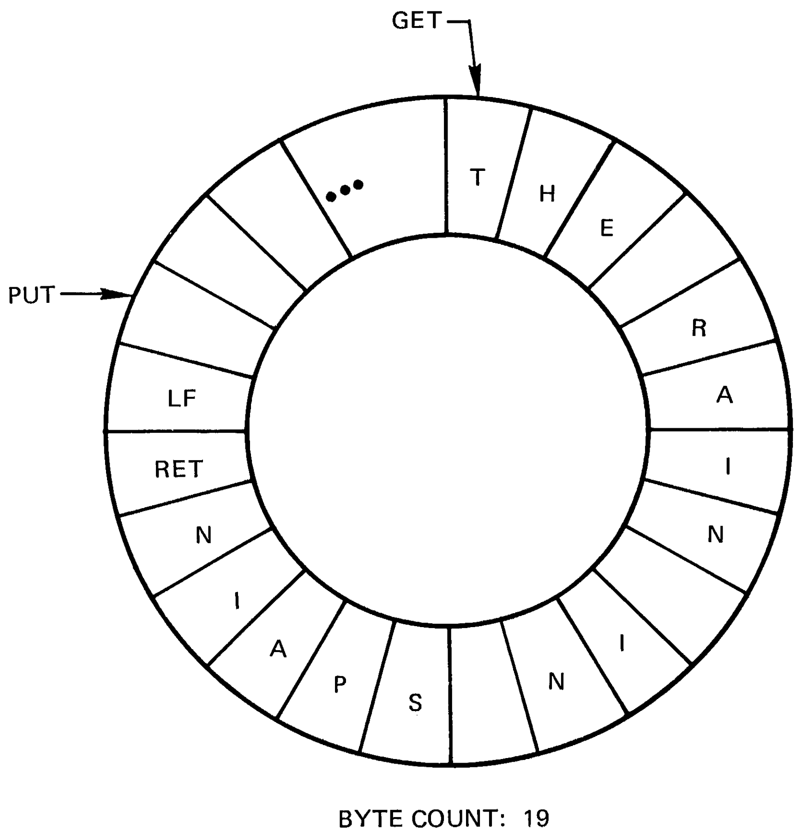
Figure 3–1: Output Ring Buffer



3.1.1.1 Storing a Character in the Output Ring Buffer – The output ring buffer is filled by characters that are passed by .TTYOUT, .TTOUTR, and .PRINT. Characters that echo what you type on the terminal are also stored here, including sets of backslashes to enclose text you rub out with the DELETE key on a hard copy terminal. To store a character in the output ring buffer, the monitor first compares the buffer size to the byte count to check for room. If there is no room, the character cannot be stored. In FB systems, this condition is sufficient to block a job if the job is doing output. (If the output is the result of echoing, it is simply discarded.) If there is enough room, the monitor checks to see if the PUT pointer is equal to the HIGH pointer. This check ensures that the PUT pointer is pointing to a location that is within the buffer. If the PUT and HIGH pointers are the same, the monitor subtracts the size of the buffer from the current PUT pointer to obtain the new PUT pointer. By doing this, the monitor "wraps" around the ring to move from the highest address in the buffer to the lowest one.

Next, the monitor moves a byte into the buffer and it increments both the PUT pointer and the byte count. Figure 3–2 shows how characters are stored in the output ring buffer.

Figure 3–2: Storing Characters in the Output Ring Buffer



3.1.1.2 Removing a Character from the Output Ring Buffer – The terminal output interrupt service routine removes characters from the output ring buffer. If the character count is 0, the routine terminates. The routine checks to see if the GET pointer is equal to the HIGH pointer. If it is, this means it is time to "wrap" around the ring to move from the highest address in the buffer to the lowest one. The wrap routine subtracts the size of the buffer from the current GET pointer to obtain the new value of the GET pointer. This check ensures that the GET pointer is pointing to a location that is within the buffer.

Next, the output interrupt service routine removes one character through the GET pointer and prepares to send it to the terminal. It increments the GET pointer and decrements the byte count.

3.1.2 Input Ring Buffer

The input ring buffer is similar to the output ring buffer except that in addition to the GET, PUT, and HIGH pointers, it has a LOW pointer that points to the first byte of the buffer. This pointer is useful when the pointers are moving backward through the buffer as a result of CTRL/U or DELETE. It indicates when to "wrap" the buffer in the reverse direction, from the lowest address to the highest.

The monitor also keeps a count of the number of lines that are stored in the input ring buffer. A *line* is any sequence of characters terminated by line feed, CTRL/Z, or CTRL/C. (Each time you type a carriage return at the terminal, RT–11 stores two characters in the input ring buffer: a carriage return and a line feed.) In normal mode, the monitor does not pass input characters to a program until an entire line is present. This is why you can use DELETE to rub out a character and CTRL/U to remove an entire line when you are typing at the terminal. Since the monitor provides for line-by-line editing, application programs need not have this overhead themselves.

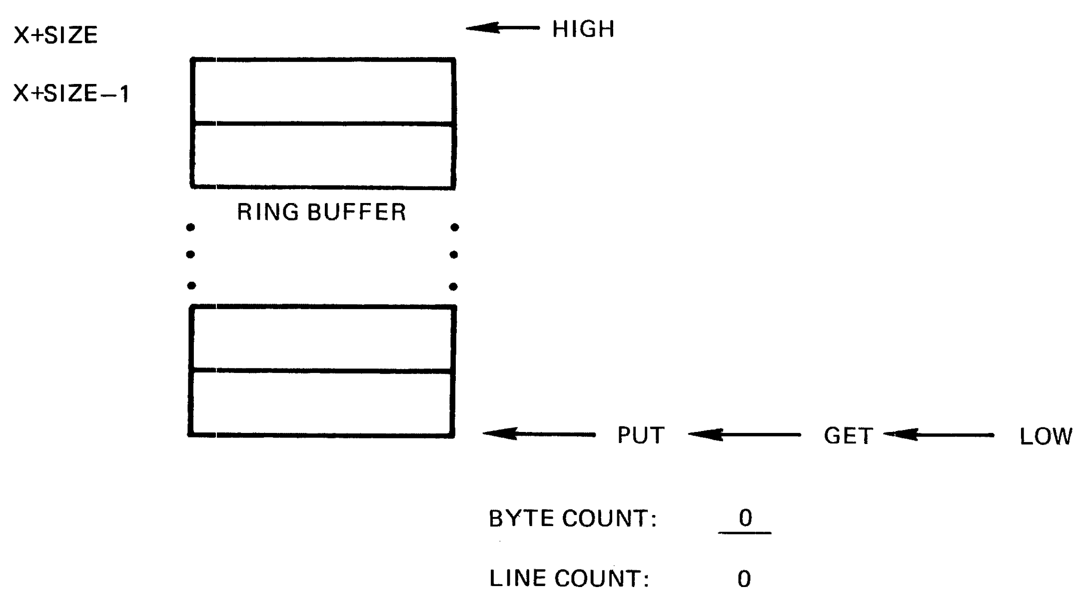
In *special mode*, however, the monitor passes bytes to a program exactly as they are typed on the terminal. In the latter case, the program itself must be able to interpret editing characters such as DELETE and CTRL/U.

NOTE

> Special mode does not provide the complete transparency required to handle devices other than terminals — such as communication lines — through the Resident Monitor terminal service. You can achieve transparency through the multi-terminal feature of RT–11 by using the "read pass-all" and "write pass-all" modes. These are described in Chapter 5.

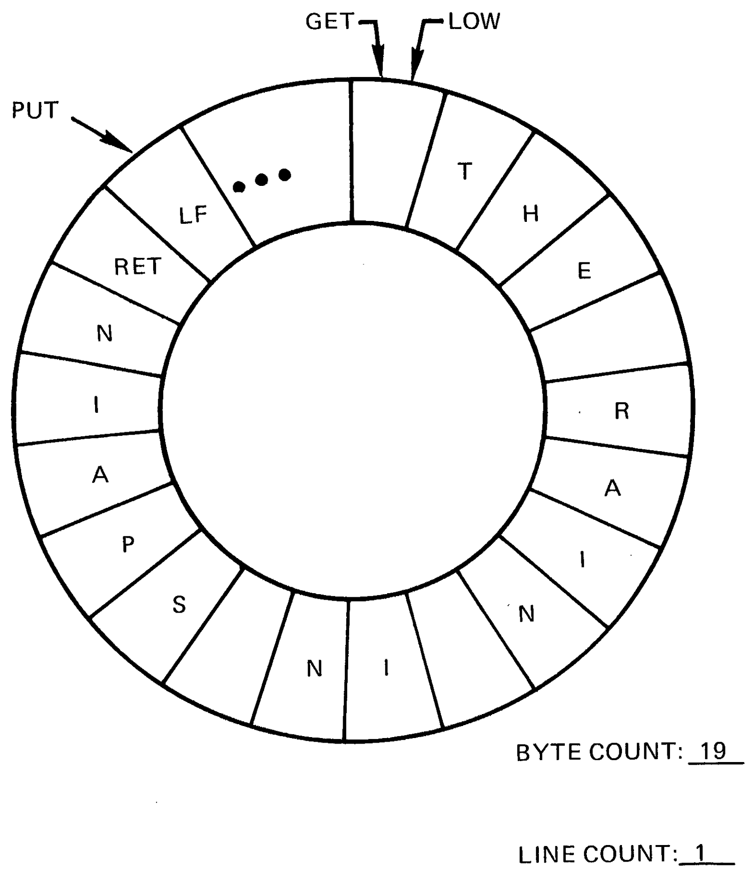
Figure 3–3 shows the input ring buffer just after the system was bootstrapped.

Figure 3–3: Input Ring Buffer



3.1.2.1 Storing a Character in the Input Ring Buffer – When you type characters at the terminal, the keyboard interrupt service routine stores them in the input ring buffer. First, the routine checks to see if there is room in the buffer. If there is no room, it rings the terminal bell (by putting a bell character in the output ring buffer). If there is room, the routine increments the byte count, increments the PUT pointer, wrapping it if necessary, and stores the byte in the ring buffer. It also increments the line counter, if the character typed is a valid line terminator. Figure 3–4 shows how characters are stored in the input ring buffer.

Figure 3–4: Storing Characters in the Input Ring Buffer



3.1.2.2 Removing a Character from the Input Ring Buffer – The monitor removes characters from the input ring buffer when it processes the .TTYIN, .TTINR, .GTLIN, .CSIGEN, and .CSISPC programmed requests. First it increments the GET pointer, wrapping around the ring if necessary. Then it gets a byte from the buffer and decrements the byte count. It decreases the line count as well if the character is a valid line terminator.

3.1.3 High Speed Ring Buffer

RT–11 provides an optional, additional high speed ring buffer that you can enable by setting the conditional HSR$B in SYCND.MAC to 1 and reassembling the monitor. This adds an extra input ring buffer to RMON; it adds an extra output ring buffer only if your system has multiple DL interfaces.

When the high speed ring buffer is present, all character processing and interpretation is performed at fork level. The high speed buffer is used to pass characters from interrupt level to fork level. The advantage of having the high speed buffer is that it allows the monitor to handle short bursts of characters coming in at a very high rate. This is useful for systems with VT100 or other intelligent terminals that report their status by sending a burst of information to the host computer. It is also useful for connecting one computer to another over a serial line.

The disadvantage to using the high speed ring buffer is that a .FORK call is required for each burst of characters, and, thus, overall terminal service may be slower.

3.1.4 Terminal I/O Limitations

Terminal input and output limitations are completely separate; you use different methods to change each of them.

RT–11 accepts terminal input in either of two forms: a line at a time, or a character at a time. In line mode, characters you type at the terminal are stored in the input ring buffer until you type a valid line terminator such as carriage return, line feed, CTRL/Z, or CTRL/C. Only then does a running program receive the line of data. The factor limiting the length of the input line is the size of the input ring buffer. (The setting of the terminal right margin bears no relation to the length of the input line.) In RT–11 V05, the default length is 134 decimal bytes, but you can change this through the system generation process. Any attempt to insert characters beyond this limit causes the terminal bell to ring, and the extra characters are lost. The Command String Interpreter can accept only 81 characters per line. Most utility programs, including PIP and BASIC–11, use the CSI to obtain lines of data from the terminal.

In character mode, a running program receives each character immediately after you type it at the terminal. In this mode, you can enter any number of characters without using a line terminator. KED uses character mode, and can thus accept lines of any length.

The length of terminal output lines is not related to the size of the output ring buffer; instead, it is related to the setting of the terminal right margin. Use the SET TT: WIDTH=*n* command to adjust the right margin. (See the *RT–11 System User's Guide* for details on SET TT: WIDTH and SET TT: CRLF.)

3.1.5 Control Functions

A special aspect of RT–11's terminal service is its response to control characters that you type at the terminal. The monitor handles each character differently, depending on the special function of each one. The following sections describe the different processes involved for the various control characters.

3.1.5.1 CTRL/C – When you type one CTRL/C at the terminal, the terminal interrupt service routine puts it into the input ring buffer, just as it would any other character. The monitor treats it as a line delineator and passes it to the running program.

However, if you type two CTRL/Cs in a row, the monitor processes them entirely differently. Instead of passing them directly to the program, the monitor aborts the running job. A program can use the .SCCA programmed request to intercept CTRL/C and prevent the abort (see the *RT–11 Programmer's Reference Manual* for a description of .SCCA).

3.1.5.2 CTRL/O – When the terminal interrupt service routine detects a CTRL/O, it never places the character in the input ring buffer, even if it is in special mode. The monitor simply toggles a flag in the impure area. (In FB and XM systems, this flag is the sign bit of the output ring buffer byte count.)

The first time you type CTRL/O, the monitor echoes it, then clears the output ring buffer byte count. It empties the ring by setting the GET and PUT pointers equal to each other, and output from a running program is thrown away. In FB and XM systems, this can unblock a job waiting for room in the output buffer. The next time you type CTRL/O or your job issues the .RCTRLO programmed request, normal output resumes.

3.1.5.3 CTRL/S and CTRL/Q – RT–11 implements terminal synchronization through the characters CTRL/S and CTRL/Q. CTRL/S, or XOFF, is a signal that stops a host computer from transmitting data to a terminal. The CTRL/Q, or XON, signal causes the computer to resume the transmission. Although XOFF has many uses, RT–11 supports only the two most common.

In a typical situation, you may be doing program development using a video terminal. When you use the TYPE monitor command to review a file, the text scrolls past faster than you can read. You can type CTRL/S to stop the display so that you can read it, and then type CTRL/Q to resume the scrolling. You initiate the XOFF yourself, in this case.

In another situation, the computer may send characters to a terminal faster than the terminal can display them. So, the terminal itself sends the XOFF signal to the computer, empties its internal silo, and sends XON when it is ready to accept more data. This procedure is transparent to you.

A flag in RMON, called XEDOFF, indicates the XOFF/XON status. Typing CTRL/S sets the flag; typing CTRL/Q clears it. When XEDOFF is set, the monitor disables terminal output interrupts and stops emptying the output ring buffer. See the *RT–11 System User's Guide* for a description of the SET TT: NOPAGE command, which disables CTRL/S and CTRL/Q processing for FB and XM systems, and for those SJ systems with the multi-terminal special feature.

3.1.5.4 CTRL/B, CTRL/F, and CTRL/X – In FB and XM systems CTRL/B and CTRL/F direct terminal I/O to the correct job. (In SJ systems these characters have no special meaning.) CTRL/X performs the same function for systems with system jobs. (See Section 3.5.9 for more information on communicating with system jobs.) The CTRL/B, CTRL/F, and CTRL/X characters are not put into the input ring buffer. Instead, they are recognized by the input interrupt service routine (unless SET TT: NOFB is in effect, in which case the characters have no special meaning) and the monitor switches the set of ring buffers it is using.

The interrupt service routine uses two control words, TTOUSR and TTIUSR, to point to the impure area of the correct job. The job's identification is stored in a special buffer in the impure area. The foreground job ID is F>; the background job ID is B>; the ID for a system job is its job name. When terminal I/O is directed to a different job, the new job's identification prints on the terminal.

3.1.6 SET Options Status Word

The word TTCNFG in the Resident Monitor is a status word that indicates which terminal SET options are in effect. For multi-terminal systems, each terminal control block has a status word similar to TTCNFG. TTCNFG reflects the status of the SCOPE, PAGE, FB, FORM, CRLF, and TAB options. Table 3–1 shows the meanings of the bits. Unused bits are reserved for future use by DIGITAL.

Table 3–1: SET Options Status Word

Bit	Meaning When Set
0	SET TT: TAB option is in effect.
1	SET TT: CRLF option is in effect.
2	SET TT: FORM option is in effect.
3	SET TT: FB option is in effect.
4–6	Reserved.
7	SET TT: PAGE option is in effect.
8–14	Reserved.
15	SET TT: SCOPE option is in effect.

To get the status word and current width of the terminal (in systems without the multi-terminal special feature), use the following lines of code:

```
MOV     @#30,Rn
MOV     -(Rn),STATUS
MOVB    -6(Rn),WIDTH
```

Use the following additional line to obtain the value of the current carriage or cursor position (a value of 0 means the cursor or carriage is at the left margin):

```
MOVB    -1(Rn),POSIT
```

3.2 Clock Support and Timer Service

You do not need a system clock in order to run RT–11 on a PDP–11 computer. However, if your computer does have a clock, RT–11 can provide basic support for keeping time of day, or it can provide timer service – standard with FB systems, and a system generation special feature for SJ systems.

3.2.1 SJ Systems Without Timer Service

An SJ system without the timer feature (the default condition) provides basic support for a system clock. Essentially, RT–11 keeps track of the time of day, but does not provide a means to implement mark time or timed wait requests.

The bootstrap routine looks for a clock on the system. If it finds one, it sets the clock bit in RMON's configuration word at fixed offset 300. If the clock has a CSR (Control and Status Register), the bootstrap turns the clock on. If the clock does not have a CSR (as is the case with some LSI–11 and PDP–11/23 computers), no executing routine can turn the clock on or off; there may be a switch for the clock on the front panel.

RMON maintains the time of day in a two-word counter. The counter is called $TIME (high-order word) and $TIME 2 (low-order word). RT–11 stores time of day as the number of ticks since midnight if you set the time with the monitor TIME command. If you do not set the time, RT–11 stores the number of ticks since the system was last bootstrapped.

RT–11 supports KW11-L and similar line frequency clocks, and KW11-P programmable clocks. (Support for the programmable clock is a feature that you select through system generation.) The default interrupt frequency for the clocks is the same as the line frequency. That is, the clock interrupts 60 times per second with 60 Hz power, and 50 times per second with 50 Hz power. Each time the clock interrupts, it adds one tick to the two-word time of day counter.

In a simple system with a clock and no timer service you can use the monitor TIME command to set the time of day or get the current time. A running program can use the .GTIM programmed request to obtain the current time, and .SDTTM to set it.

3.2.2 Systems with Timer Service

Timer service is always included in FB and XM systems. It is a system generation special feature for SJ systems. Timer service provides three extra programmed requests: the mark time request (.MRKT), the cancel mark time request (.CMKT), and the timed wait request (.TWAIT, in FB and XM only). In addition, another system generation special feature provides device time-out support through the time-out macro (.TIMIO) and the cancel time-out macro (.CTIMIO), which are described fully in Chapter 7.

Because timer support itself requires the fork queue, selecting this feature in SJ results in real, rather than simulated, fork processing. (Usually in SJ a .FORK request returns immediately to the following instructions.) With a real fork queue in SJ, .FORK requests are serialized and do not interrupt one another. For more information on the .FORK request, see Chapter 6.

To implement timer services, RT–11 uses a timer queue, which is a linked list of queue elements, sorted in order of expiration time. The element that expires soonest is at the head of the queue. The .MRKT, .TWAIT, and .TIMIO requests use the timer queue. They schedule completion routines to be executed after a certain time interval elapses.

The monitor uses the timer queue internally to implement the .TWAIT programmed request, which causes the job that issues it to be suspended. The monitor places a timer request in the timer queue, with the .RSUM programmed request code as its completion routine. The job waits until the specified time interval has elapsed. Execution resumes when the monitor itself issues the .RSUM request as a completion routine.

Figure 3–5 shows the format of a timer queue element. It includes the symbolic names and offsets as well as the contents of each word in the data structure. Note that time is stored as a two-word number — a modified expression of the number of ticks until the timed wait expires.

Figure 3–5: Timer Queue Element Format

NAME	OFFSET	CONTENTS
C.HOT	0	HIGH-ORDER TIME
C.LOT	2	LOW-ORDER TIME
C.LINK	4	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
C.JNUM	6	OWNER'S JOB NUMBER
C.SEQ	10	OWNER'S SEQUENCE NUMBER ID
C.SYS	12	–1 IF SYSTEM TIMER ELEMENT; –3 IF .TWAIT ELEMENT IN XM
C.COMP	14	ADDRESS OF COMPLETION ROUTINE
		THREE ADDITIONAL WORDS ARE PRESENT IN XM SYSTEMS. THEY ARE UNUSED, AND ARE RESERVED FOR FUTURE USE BY DIGITAL.

To store the time of day in all systems with timer support, RT–11 uses a two-word pseudo clock called PSCLOK (low-order word) and PSCLKH (high-order word). In this pseudo clock RMON stores the time, in ticks, that has elapsed since the system was bootstrapped. Each clock interrupt adds one tick to the counter. Two other words — $TIME and $TIME 2 — contain a constant that, when added to the value of the pseudo clock, yields the current time of day.

The monitor uses the pseudo clock to implement timer requests. When a new queue element is put on the queue, the monitor adds the low-order word of the pseudo clock to the two-word time value in the queue element and it stores the resulting value, a modified time, in the queue element time words. Whenever the pseudo clock carries into the high-order word (approximately every 18 minutes), the monitor subtracts 1 from the high-order word of time in each pending timer queue element. The element expires when the high-order time word is 0 and the low-order time word is less than or equal to the pseudo clock low-order word. This method of storing time information means that handling timer requests requires only test and compare instructions, which execute rapidly, and a pass over the queue roughly every 18 minutes to correct the time words.

Every time the system clock interrupts, the monitor increments the pseudo clock. It then checks the first element in the timer queue. If the high-order word of the timer element is 0 and the low-order word is greater than the low-order word of the pseudo clock, the element has expired. The monitor removes it from the timer queue and processes it as a completion routine for the correct job. The monitor continues to check the timer queue until it finds an element that has not yet expired or the queue is empty.

There are several uses for system timer elements. If C.SYS is –1, the element is being used by .TIMIO for device time-out support, or by RMON for multi-terminal device time-out. If C.SYS is –3, the element is being used to implement a .TWAIT request in an XM system. For .MRKT and other .TWAIT requests, C.SYS is 0.

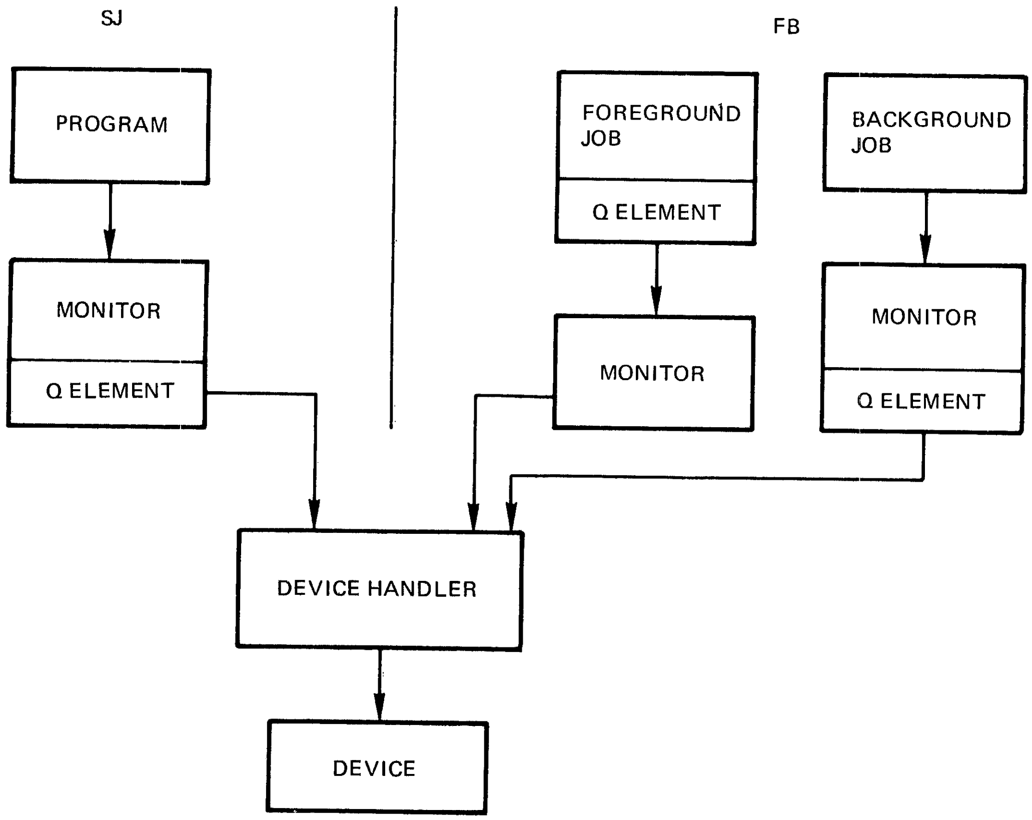
In XM, completion routines that have –1 in C.SYS are run in kernel mode and the queue element is discarded. That is, the queue element is not linked into the list of available elements. If C.SYS is –3, the completion routine is still run in kernel mode. However, the queue element is linked into the available queue when the completion routine is run. (The timer queue element is used as the completion queue element.) In all other cases, the queue element is linked into the available queue and completion routines run in user mode. (Chapter 4 provides more information on extended memory systems.)

3.3 Queued I/O System

RT–11 performs I/O transfers through a queued I/O system. A job can thus have multiple I/O requests outstanding at a given time — that is, it can issue an I/O request and still continue processing.

RT–11 implements queued I/O through the queue elements, the device handlers, and the routines in the Resident Monitor. Once a device handler is in memory and the job has opened a channel, any .READ or .WRITE requests for the corresponding peripheral device are interpreted by the monitor and translated into a call to the handler. Figure 3–6 illustrates the relationship between these components.

Figure 3–6: Components of the Queued I/O System



3.3.1 I/O Queue

The RT–11 I/O queue system consists of a linked list of queue elements for each resident device handler and a queue of available elements for each job. I/O queue elements are seven words long for SJ and FB systems, and 10 decimal words long for XM systems. RT–11 provides one queue element in the Resident Monitor for the SJ environment. For the FB and XM environments, each job has one queue element in its impure area. One queue element is sufficient for a job that uses wait-mode I/O.

Figure 3–7 shows the format of an I/O queue element. It includes the symbolic names and offsets, as well as the contents of each word in the data structure.

Figure 3–7: I/O Queue Element Format

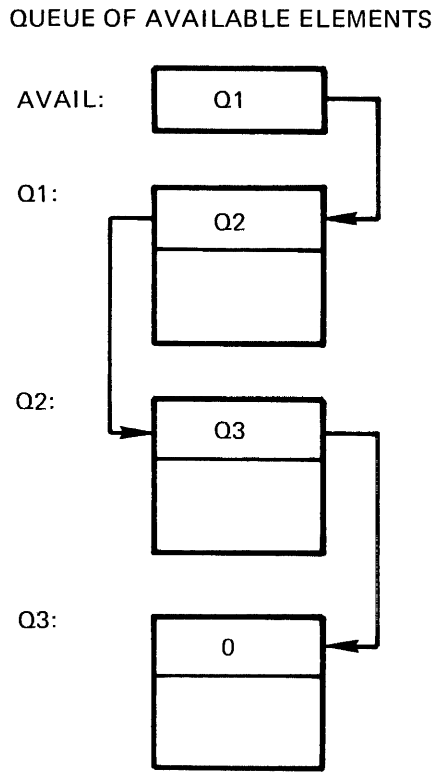
<table>
<tr><th>NAME</th><th>OFFSET</th><th colspan="4">CONTENTS</th></tr>
<tr><td>Q.LINK</td><td>0</td><td colspan="4">LINK TO NEXT QUEUE ELEMENT; 0 IF NONE</td></tr>
<tr><td>Q.CSW</td><td>2</td><td colspan="4">POINTER TO CHANNEL STATUS WORD IN I/O CHANNEL (SEE FIGURE 3-29)</td></tr>
<tr><td>Q.BLKN</td><td>4</td><td colspan="4">PHYSICAL BLOCK NUMBER</td></tr>
<tr><td>Q.FUNC
Q.UNIT
Q.JNUM</td><td>6
7
7</td><td>RESERVED

(1 BIT)</td><td>JOB
NUMBER
(4 BITS)
0 = BG</td><td>DEVICE
UNIT
(3 BITS)</td><td>SPECIAL
FUNCTION
CODE
(8 BITS)</td></tr>
<tr><td>Q.BUFF</td><td>10</td><td colspan="4">USER BUFFER ADDRESS (MAPPED THROUGH PAR1 WITH Q.PAR VALUE, IF XM)</td></tr>
<tr><td>Q.WCNT</td><td>12</td><td colspan="4">WORD COUNT { IF <0, OPERATION IS WRITE
IF =0, OPERATION IS SEEK
IF >0, OPERATION IS READ
THE TRUE WORD COUNT IS THE ABSOLUTE VALUE OF THIS WORD.</td></tr>
<tr><td>Q.COMP</td><td>14</td><td colspan="4">COMPLETION ROUTINE CODE { IF 0, THIS IS WAIT-MODE I/O
IF 1, JUST QUEUE THE REQUEST AND RETURN
IF EVEN, COMPLETION ROUTINE ADDRESS</td></tr>
<tr><td>Q.PAR</td><td>16</td><td colspan="4">PAR1 VALUE (XM ONLY)</td></tr>
<tr><td></td><td></td><td colspan="4">RESERVED (XM ONLY)</td></tr>
<tr><td></td><td></td><td colspan="4">RESERVED (XM ONLY)</td></tr>
</table>

If your program uses asynchronous I/O, you must allocate more queue elements for it by using the .QSET programmed request. Otherwise, if the program initiates an I/O transfer and no queue element is available, RT–11 must wait for a free element before it can queue up the new request. Obviously, this slows processing. The number of queue elements is always sufficient when you allocate *n* new elements, where *n* is the total number of pending requests that can be outstanding at one time for a particular program. This produces a total of $n+1$ available elements, since the original single queue element is added to the list of available elements.

The list header, called AVAIL, is a linked list of free queue elements. It contains a pointer to an available queue element. If AVAIL is 0, no elements are currently available. Figure 3–8 shows an I/O queue with three queue elements, all of which are available. In this diagram, AVAIL points to element 1. The first word in each queue element is a pointer to the next element in the queue. Thus, element 1 is linked to element 2, element 2 is linked to element 3, and element 3 is the last element in the linked list; its link word is 0.

Figure 3–8: I/O Queue with Three Available Elements



When a program initiates a request for an I/O operation, the monitor allocates a queue element for the request by removing it from the list of available elements. The monitor then links the element into the I/O queue for the appropriate device handler. This is accomplished by using two words in the handler header — ddLQE and ddCQE.

The fourth word of the handler is a pointer to the last element in its queue. This pointer is called *ddLQE*, where *dd* is the two-character physical device name. The fifth word of the handler, called *ddCQE*, is a pointer to the current queue element.

Figure 3–9 shows the status of the queue elements when one I/O request is pending. The monitor removes the first queue element from the available list and puts it on the device handler's queue.

When a program requests a second I/O transfer for the same handler before the first transfer completes, the monitor removes another queue element from the available list and adds it to the queue for that handler. Figure 3–10 illustrates this.

Figure 3–9: I/O Queue with Two Available Elements

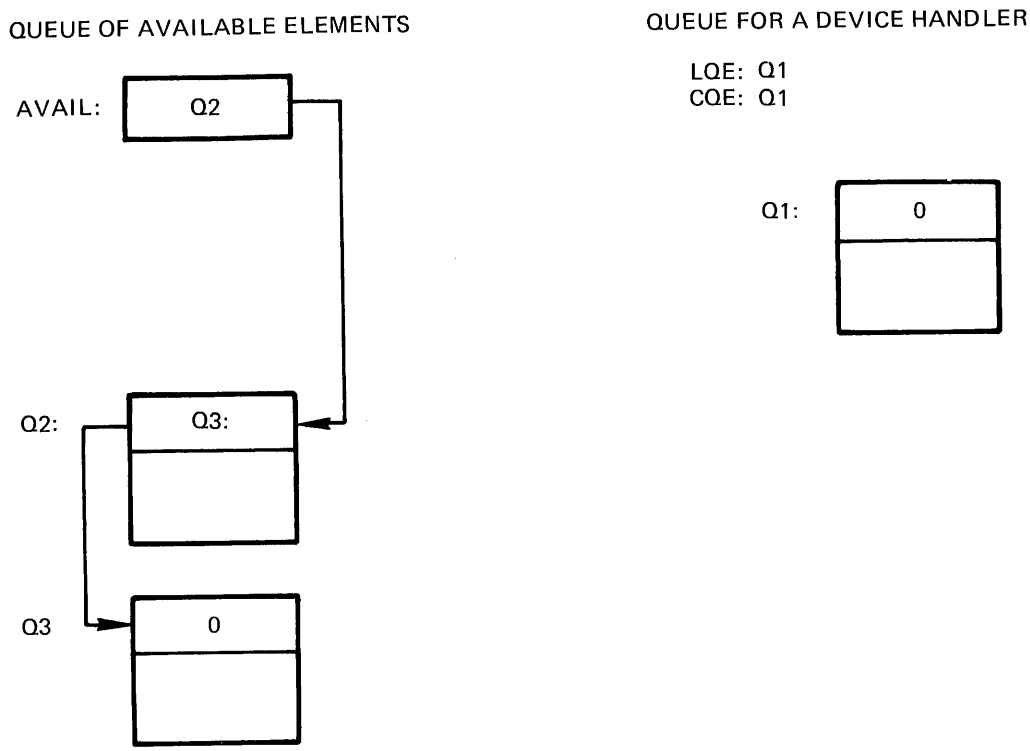


Figure 3–10: I/O Queue with One Available Element

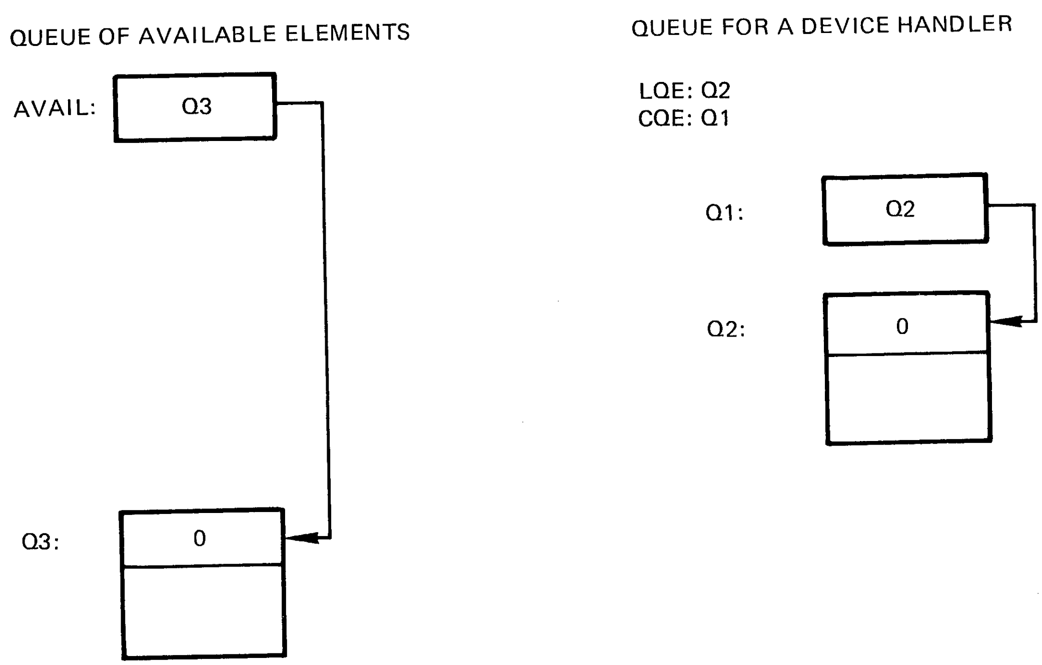


Figure 3–11: I/O Queue When One Element Is Returned

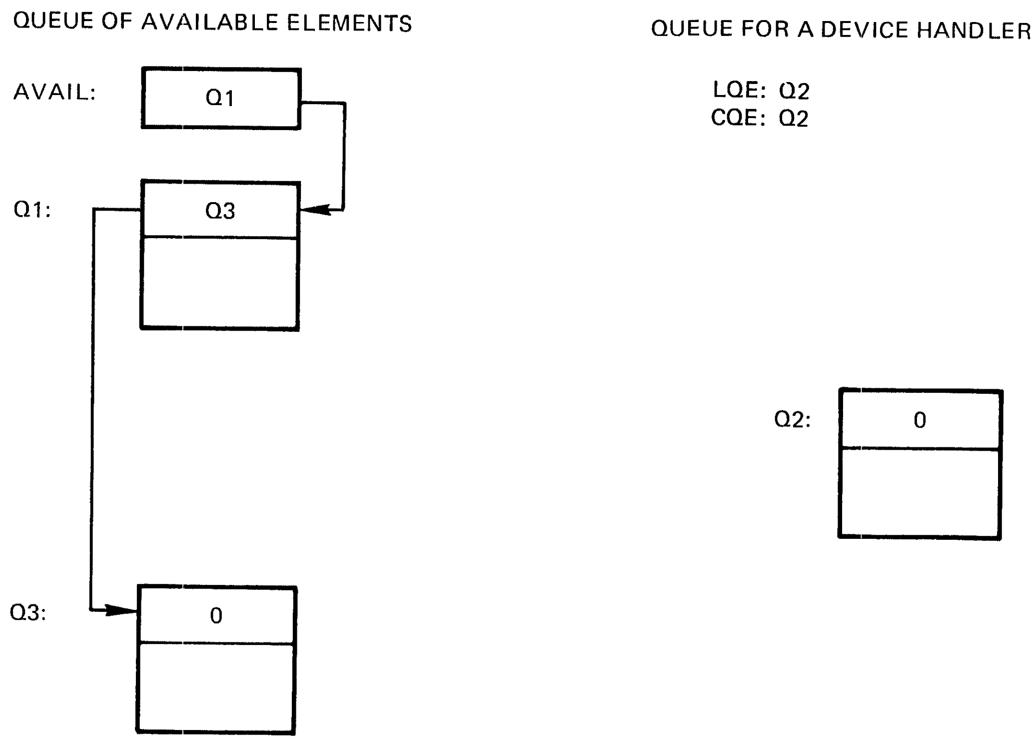
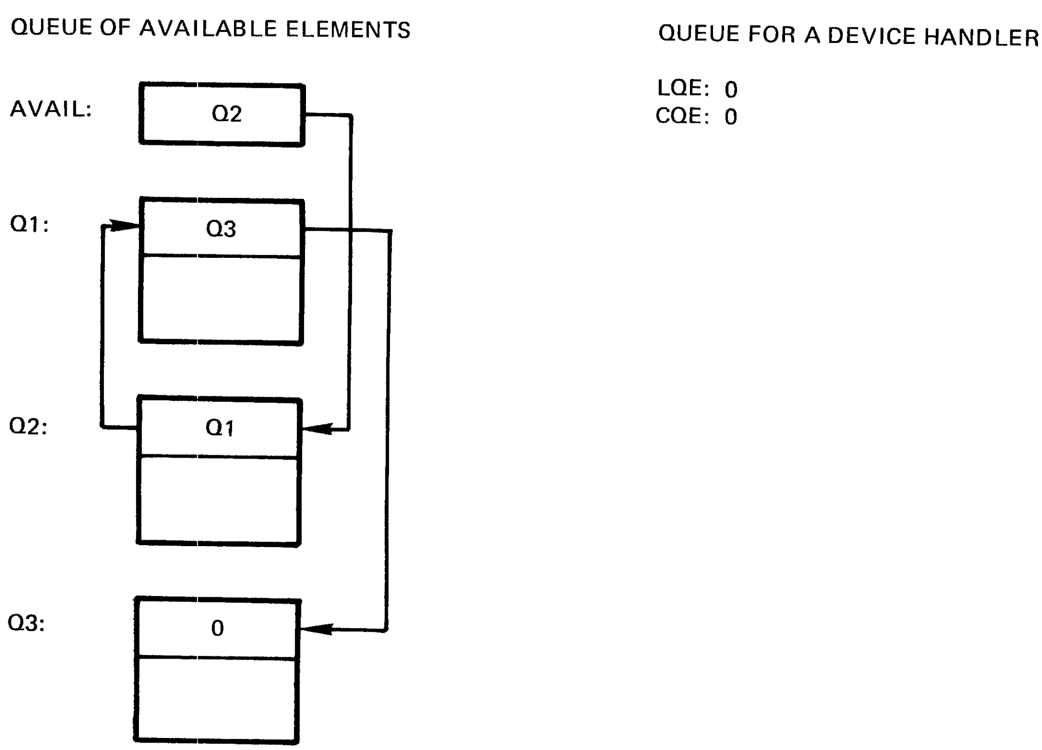


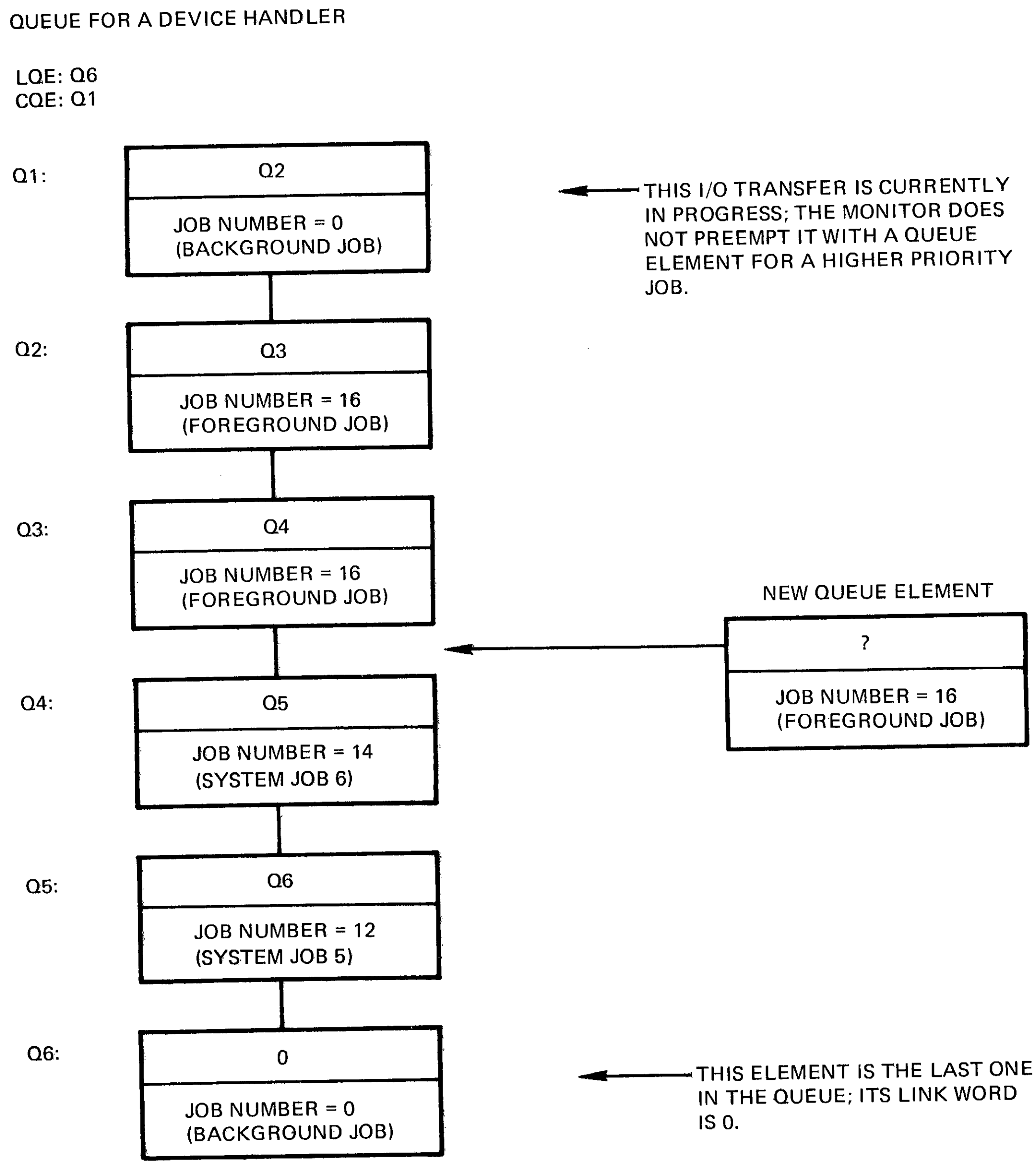
Figure 3–12: I/O Queue When Two Elements Are Returned



When the transfer currently in progress completes, the monitor returns queue element 1 to the available list and initiates the transfer indicated by queue element 2. Figure 3–11 illustrates the queue status when one element is returned.

When the I/O operation indicated by queue element 2 finishes, the monitor returns that element to the available list, as Figure 3–12 indicates. Note that the elements are now linked in a different order from that shown previously in Figure 3–8.

Figure 3–13: Device Handler Queue When a New Element Is Added



In SJ systems, the monitor always puts the new queue element at the end of the device queue. By using ddLQE it can do this quickly. In FB and XM systems, the device queue is sorted in order by job number, with the queue elements belonging to the highest job number appearing at the beginning of the queue and those belonging to the lowest job number at the end. The monitor puts the new element in the queue at the end of the list within a specific job group. Thus, if two requests are queued waiting for a particular handler, the request with the higher job number is honored first. At no time though, does the monitor abort an I/O transfer already in progress to start a higher priority request. The operation in progress always completes before the monitor initiates another transfer.

Figure 3–13 illustrates a large queue for a device handler. The monitor adds the new element, an I/O request from the foreground job, to the queue at the end of the list of other foreground job elements. Note that the monitor does not preempt the current queue element, even though it is a request from the background job.

3.3.2 Completion Queue

In FB and XM systems, the monitor maintains a completion queue for each job, using it to serialize completion routines for each job. The head of the completion queue is called I.CMPL and it is located at offset 6 from the start of the impure area. I.CMPE, at offset 4, points to the end of the completion queue. By using I.CMPE, the monitor can quickly add a new completion queue element to the end of the queue.

A completion routine is a section of code in a program that begins to execute as soon as an asynchronous event occurs. For example, the .READC programmed request starts an I/O transfer and provides the address in the program at which execution is to begin when the I/O transfer completes. See the *RT–11 Programmer's Reference Manual* for a more thorough description of completion routines.

When an I/O transfer completes, the monitor checks Q.COMP at offset 14 octal from the start of the I/O queue element. If the value is greater than 1 it specifies a completion routine address. The monitor then transforms the I/O queue element into a completion queue element and places it on the completion queue for the job whose job number appeared in Q.JNUM at offset 7 from the start of the I/O queue element.

Figure 3–14 shows the format of a completion queue element. It includes the symbolic names and offsets, as well as the contents of each word in the data structure.

Figure 3–14: Completion Queue Element Format

NAME	OFFSET	CONTENTS
Q.LINK	0	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
	2	RESERVED
	4	RESERVED
	6	RESERVED
Q.BUFF	10	CHANNEL STATUS WORD
Q.WCNT	12	OFFSET FROM START OF CHANNEL AREA TO THIS CHANNEL
Q.COMP	14	COMPLETION ROUTINE ADDRESS
		THREE ADDITIONAL WORDS ARE PRESENT IN XM SYSTEMS. THEY ARE UNUSED, AND ARE RESERVED FOR FUTURE USE BY DIGITAL.

3.3.2.1 SJ Considerations — The SJ monitor does not maintain a completion queue. As a result, completion routines in SJ are never serialized. (Whether or not you select timer support at system generation time does not affect the serialization of completion routines.) When you issue a completion-mode programmed request (such as .READC or .WRITC) and the I/O transfer completes, the monitor does not transform the I/O queue element into a completion queue element. Instead, it returns the element to the list of available queue elements. It then moves the Channel Status Word to R0 and the channel number to R1, and begins executing the program's completion routine. Thus, the completion of another I/O transfer could interrupt the current completion routine and begin execution of another one.

3.3.2.2 .SYNCH Considerations — The .SYNCH request also makes use of the completion queue in FB and XM systems but it does not use an I/O queue element. When you issue a .SYNCH call, you supply as an argument the address of a ten-word area in your program, called the synch block. The synch block contains, among other things, the address of the routine to be executed. Figure 3–15 shows the format of a synch block, or synch queue element. When the monitor interprets your .SYNCH request there is no current I/O queue element for it to modify. So, it uses your ten-word area as a completion queue element. The monitor puts the synch block at the head of the appropriate job's completion queue.

Figure 3–15: Synch Queue Element Format

NAME	OFFSET	CONTENTS
Q.LINK	0	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
Q.CSW	2	JOB NUMBER
Q.BLKN	4	RESERVED
Q.FUNC	6	RESERVED
Q.BUFF	10	SYNCH ID
Q.WCNT	12	−1 (CUE THAT THIS IS A SYNCH ELEMENT)
Q.COMP	14	SYNCH ROUTINE ADDRESS
		THREE ADDITIONAL WORDS ARE PRESENT IN XM SYSTEMS. THEY ARE UNUSED, AND ARE RESERVED FOR FUTURE USE BY DIGITAL.

3.3.3 Flow of Events in I/O Processing

As the central manager of the device-independent I/O system, the Resident Monitor supervises the I/O procedure, using a queue element as the communication link between a device handler and a program that requests an I/O transfer. The following sections describe the sequence of events that occur in a simple read or write operation.

3.3.3.1 Issuing the Request — Before a program can request an I/O transfer, it has to open a new file or find an existing file on a device. This procedure sets up a channel containing five words of information about the location and length of the file. A channel number is associated with the five-word block so that you can refer to the block later by specifying this number in a single byte. The monitor uses the channel information when it needs to process an I/O request.

A running program initiates an I/O procedure by issuing a request to read from or write to a particular channel. MACRO–11 programs, for example, can use the .READ, .READW, .READC, .WRITE, .WRITW, .WRITC, and .SPFUN programmed requests. Programs written in other languages use similar statements to read and write data.

When the I/O request executes, the monitor uses the channel number the request specifies to find the corresponding device handler. Then the monitor calls its queue manager routine, which allocates a queue element from the list of available elements and fills in the necessary information.

When a queue element is not available in SJ systems, the monitor executes in a tight loop, waiting for a queue element to appear in the list of available elements. This condition is satisfied when a device interrupts and the handler issues the .DRFIN macro, which indicates that an I/O transfer is complete, and the monitor returns the queue element for that transfer to the available list.

When a queue element is not available in FB and XM systems, the job requests a scheduling pass starting with the job whose priority is immediately below that of the current job. When the original job gets a chance to run again, it first checks the available list for a free queue element. If no element is available, it requests another scheduling pass. In FB systems, there is no blocking bit associated with queue element availability. Therefore, the job that needs a queue element is not officially blocked, even though it cannot run effectively until it gets a queue element.

3.3.3.2 Queuing the Request in SJ — Once a new queue element has been allocated by the queue manager, the element is put on the device handler's queue. In an SJ system the new element always goes at the end of the queue. To prevent interference from a device interrupt (which might remove a different element from the same queue), the SJ monitor goes to priority 7 to manipulate the queue.

If the queue is empty, the monitor makes the new element both the current and the last element in the queue. It increments the count of queue elements on this channel (the byte at offset 10 octal in the channel area), and returns the priority to its previous level. It then jumps to the handler's I/O initiation section to start up the transfer. The handler starts the transfer and returns control to the monitor with an RTS PC instruction.

If the queue is not empty, the handler is busy so the monitor puts the new element at the end of the queue. It increments the count of queue elements on this channel (the byte at offset 10 octal in the channel area), and returns the priority to its previous level.

Whether or not the queue was empty, the queue manager checks to see if this request is for wait-mode I/O. If it is, the system executes in a tight loop until the transfer specified by this queue element finishes. If this request is not for wait-mode I/O, control returns to the program, which executes while I/O occurs simultaneously.

3.3.3.3 Queuing the Request in FB and XM — In FB and XM systems, all jobs (system utility programs, application programs, and language processors) and the Keyboard Monitor run in *user state*. Each job uses its own stack. In user state a low-priority job that is running can be replaced by a higher-priority job that is runnable. Similarly, a higher-priority job that is unable to run for any reason can be replaced by a runnable lower-priority job.

The FB and XM monitors switch to *system state* to modify important data structures and to perform operations that do not run entirely within a job. Stack operations and interrupts in system state use the monitor's stack rather than a job's stack. Jobs cannot run when the monitor is in system state, and switching between lower- and higher-priority jobs is postponed until the monitor returns to user state. In system state, then, the monitor can safely modify critical data structures without the risk that another job could gain control and corrupt the same data structures. (Section 3.4.1 describes system and user state in greater detail.)

Because in SJ systems there is only one execution state, the terms "user state" and "system state" are not meaningful in those systems.

In an FB or XM system, the monitor switches to system state before it puts the new element on the device handler's queue in order to prevent interference from other jobs. It does not raise the priority to 7, as does the SJ monitor, because this would lock out device interrupts for too long a time. However, a device interrupt could remove an element from the queue while the monitor is adding the new element and adjusting the LQE and CQE pointers. To ensure the integrity of the queue, the monitor *holds* the handler while it performs the modification.

Holding a handler prevents any other process or routine from changing the I/O queue. For example, when a device interrupts and an I/O operation completes, the handler issues a .DRFIN call to return to the monitor and remove the current queue element from the I/O queue. Depending on the type of I/O request the program issued, the current element should either go back to the linked list of available queue elements, or it should go onto the completion queue for the appropriate job. However, if the handler is held when it issues the .DRFIN, the monitor does not remove the current queue element from the I/O queue. Instead, it delays this action by setting a flag that it checks later. Similarly, when a job aborts, the abort routine holds a handler while it removes queue elements belonging to the aborted job. This prevents the monitor from starting up the next transfer queued for this device until all elements for the aborted job are gone. After the monitor holds the device handler, it checks to see if the queue is empty.

If the queue is empty, the monitor clears the hold flag for the handler right away, and then makes the new element both the current and the last element in the queue. It increments both the count of queue elements on this channel (the byte at offset 10 octal in the channel area), and the total number of I/O requests for this job. Remaining in system state, the monitor jumps to the device handler's I/O initiation section to start up the transfer. When the handler starts the transfer and returns control with an RTS PC instruction, execution of the program continues in user state within the queue manager. That is, the monitor is executing "for the program".

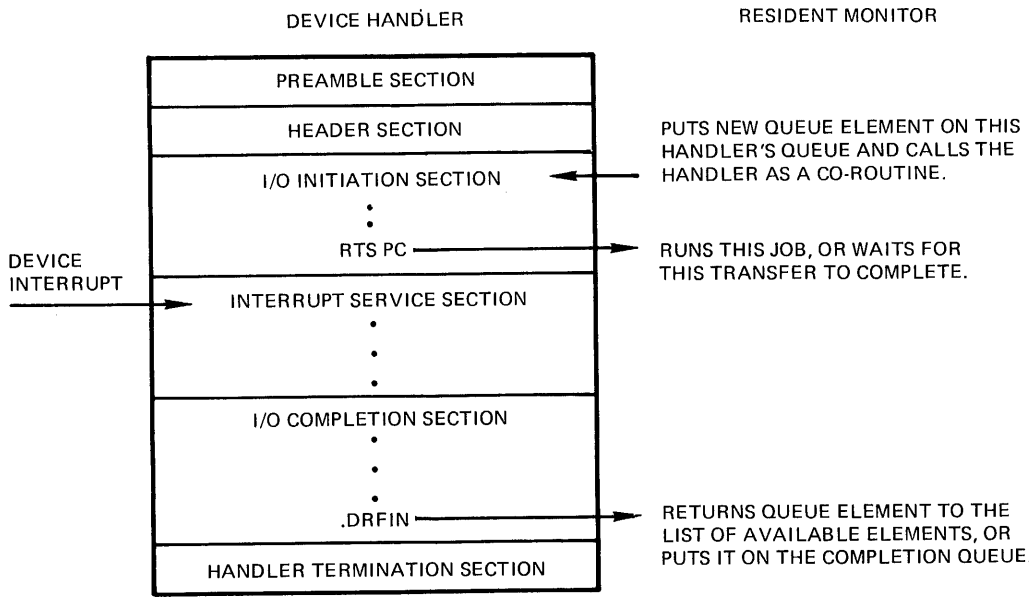
If the queue is not empty, the monitor continues to hold the handler until it finishes modifying the queue. Elements in the queue are sorted by job number, as Section 3.3.1 explains. The monitor searches the queue from front to back, and places the new element at the end of the group of elements belonging to this job. It increments both the count of queue elements on this channel (the byte at offset 10 octal in the channel area), and the total number of I/O requests for this job. Since the device handler is busy, the monitor cannot start up an I/O transfer for this request, so its queue element sits in the queue. The queue manager returns to user state.

Whether or not the queue was empty, the queue manager checks to see if this request is for wait-mode I/O. If so, the program waits for the transfer to complete. If this request is not for wait-mode I/O, execution of the program continues concurrently with the I/O transfer.

3.3.3.4 Performing the I/O Transfer — After the monitor and a device handler have started up an I/O transfer, a peripheral device performs the actual operation and interrupts when it is finished. The interrupt causes control to pass to the device handler's interrupt service section, where the code assesses the results of the I/O operation and restarts it if necessary. When the transfer is done, the handler uses the .DRFIN macro to return to the monitor and remove the current queue element from its I/O queue.

Figure 3–16 summarizes the relationship between the parts of a device handler and the Resident Monitor. Chapter 7 provides a detailed description of the internal operation of a device handler.

Figure 3–16: Device Handler/Resident Monitor Relationship



3.3.3.5 Completing the I/O Request — When a device interrupts, an I/O transfer completes, and the handler issues the .DRFIN call, it is the monitor that must take the appropriate action to complete the I/O procedure. In general, this means that the monitor must remove the current queue element from the handler's I/O queue and put it in the list of available elements or on the completion queue. In an FB or XM system, another I/O request could cause the monitor to hold the handler while it adds an element to the queue. In this case, the monitor simply sets a flag, dismisses the interrupt, and returns to the interrupted process, removing the element later.

In all SJ systems, and in those FB or XM systems in which the handler is not held, the monitor first decrements the count of queue elements on this channel. In an FB system, when the count reaches 0, it makes runnable a job that is waiting for activity on this channel to complete. In FB and XM systems only, the monitor next decrements the total number of I/O requests pending for this job. Again, if this number becomes 0, it makes runnable a job that is waiting for all its I/O to complete. When either count reaches 0, it can cause the scheduler to run.

Next, for all systems, the monitor removes the queue element from the handler's queue. If there is another element in the handler's queue waiting to be processed, the monitor calls the handler again to start the next operation as soon as the final disposition of the current element is resolved. The monitor raises the priority to 7 for a short time as it links the element into either the list of available elements or (except for SJ systems) the job's completion queue. In FB, if the element specifies a completion routine address at offset 14 octal, the monitor transforms the I/O queue element into a completion queue element and puts it at the end of the job's completion queue. Then the monitor returns control to the process or program that was interrupted. In SJ, if the element specifies a completion routine, the monitor merely returns the I/O queue element to the list of available elements. Then it puts the Channel Status Word in R0, puts the channel number in R1, and begins immediate execution of the completion routine.

In all SJ systems, and in those FB and XM systems in which the element does not specify a completion routine address, the monitor simply returns the element to the available list. Control returns to the process or program that was interrupted, or (except in SJ systems) the scheduler can run.

3.4 Scheduling in Foreground/Background Systems

In an FB or XM system the monitor must arbitrate the demands of up to eight jobs for processor time, in addition to performing all its other functions. Clearly then, because of the implications of having more than one job, the FB and XM systems are considerably different from the SJ system. The FB and XM monitors use a number of special tools to implement support for more than one job.

The *scheduler* is the part of the monitor that determines which job is eligible to run and gives control of the processor to it. The scheduler uses a simple algorithm to determine which job should run. It looks at the jobs in order from highest priority to lowest. If a job exists and is runnable, the monitor restores its context and returns to it. Status bits in a flag word (I.BLOK, at offset 36 octal from the start of the impure area) reflect the *blocking conditions* that can prevent a job from running and thereby give a lower-priority job a chance to execute. *Context switching* is the procedure through which the monitor saves a job's *context* — its machine environment and important job-specific information — and begins execution of another job.

All the processes that are job-dependent are kept separate from those that are monitor functions. The monitor functions are, therefore, re-entrant. Data structures that contain job-specific information are located in the *impure area* for each job, and each job has its own stack. Routines that run in a job-dependent environment, including some parts of the monitor, use the job's stack and run as part of the user job in *user state*. Any routines that run outside a job's context, including interrupts, use the monitor's stack and execute in *system state*. This arrangement allows the monitor to "unwind" the stack after a series of interrupts without changing jobs or stacks.

Two or more jobs can share a peripheral device, so the queued I/O system (as Section 3.3 explains) must keep track of the priority of the job requesting an I/O transfer and act accordingly. The USR is serially reentrant — that is, it cannot be shared by two jobs; all jobs must take turns using the USR.

Lastly, *monitor routines* check for blocking conditions, change execution state, interlock parts of the monitor to prevent corruption of important data structures, request a scheduling pass, and so on. The following sections describe the components of FB and XM systems and provide an understanding of the scheduling process in a multi-job environment.

3.4.1 User and System State

In order to isolate job-dependent functions from monitor processes, the FB and XM systems provide two execution states: user state and system state. All jobs and the Keyboard Monitor run in user state. Each job maintains relevant data in its impure area, and each job uses its own stack. Context switching is enabled in user state. That is, a lower-priority job that is running can be replaced by a higher-priority job that is runnable. A higher-priority job that is unable to run for any reason can be replaced by a runnable lower-priority job.

The monitor switches to system state and the system stack for several reasons. Jobs cannot run when the monitor is in system state, and context switching is delayed until the monitor returns to user state. Consequently, the monitor can modify important data structures in system state without interference from other jobs. The monitor uses system state for operations that do not run entirely within a job context. These operations, which must not be interrupted by context switching, include the following:

- Blocking a job
- Starting up an I/O transfer
- Aborting an I/O transfer
- Servicing a timer request
- Executing the .PROTECT programmed request
- Executing the .CHCOPY programmed request
- Interlocking the USR
- Executing any XM mapping programmed request
- Servicing an interrupt
- Executing device handler code (except for .TIMIO completion routines and .SYNCH routines, which run in user state in a specific job's context)

Because it is chiefly system or monitor routines that execute in system state, monitor errors are fatal. Traps to 4 (odd address errors, and illegal or nonexistent memory addressing errors) and traps to 10 (illegal or reserved instruction errors) occurring in system state halt the system.

3.4.1.1 Switching to System State Asynchronously — The monitor switches from user state to system state asynchronously whenever an interrupt occurs. As a result of the interrupt the monitor may modify important data structures. The switch to system state prevents interference from a context switch while the modifications are in progress. In FB the monitor switches from the job's stack to the system stack. In XM the monitor does not perform the stack switch because the hardware does it automatically. Subsequent interrupts that occur in system state put information on the system stack. Note that these subsequent interrupts do not cause another switch to system state.

Interrupt Level Counter

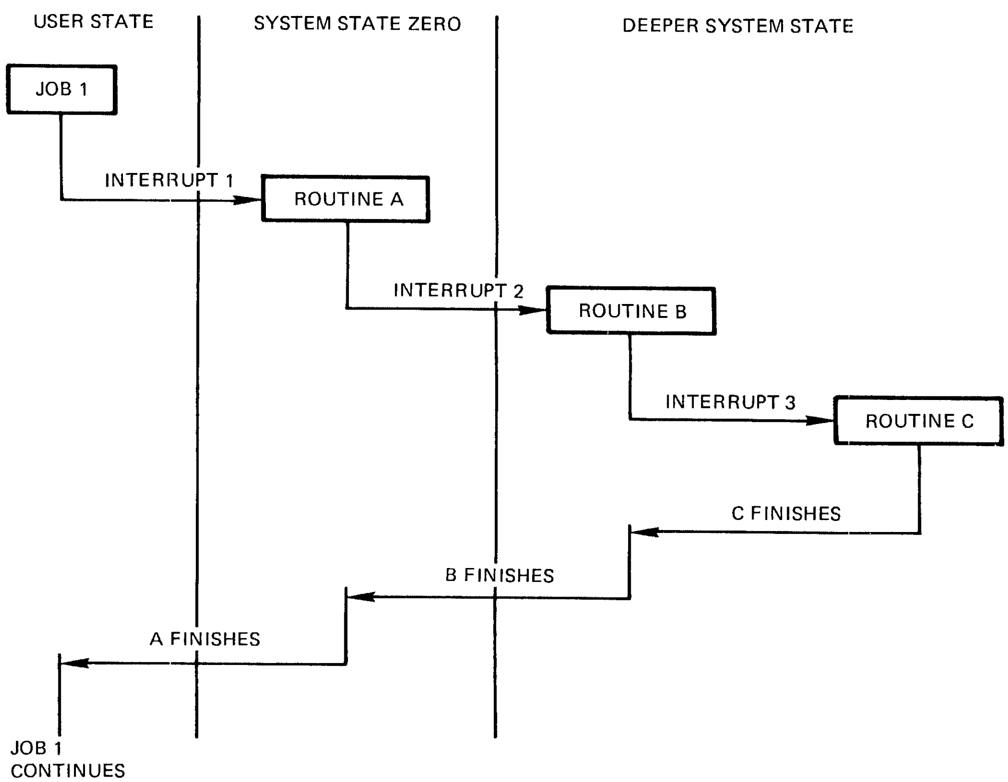
The monitor recognizes three levels of execution state. It uses a counter called INTLVL to distinguish among the three levels. Every interrupt increments this counter. When INTLVL is –1, execution is in user state. When INTLVL is 0, execution is in system state at level zero. When INTLVL is positive, execution is still in system state, but at a deeper interrupt level. Table 3–2 summarizes the relationship between the number of interrupts pending and the execution state.

Table 3–2: Values of the interrupt Level Counter (INTLVL)

Number of Interrupts	Value of INTLVL	Execution State
0	–1	User State
1	0	System State Level Zero
2 or more	1 or greater	Deeper System State

Figure 3–17 shows how interrupts influence the flow of events in a running system.

Figure 3–17: Interrupts and Execution States



$INTEN Monitor Routine

When an interrupt occurs, control passes to the routine specified in the interrupt vector, and the current PS and PC are put on the job's stack. In RT–11, both device handlers and in-line interrupt service routines call the monitor at the common interrupt entry point, $INTEN. Device handlers use the .DRAST macro to call the monitor; in-line interrupt service routines use the .INTEN macro.

$INTEN is the monitor routine that performs the switch to system state. The routine assumes that it was called because an interrupt occurred. Therefore, it expects the old PS and PC to be on the job's stack. The priority should be 7, and the interrupt service routine must not have destroyed any registers between the time the interrupt occurred and the time $INTEN was called. Device handlers generally call the monitor immediately, before they do any processing at all. In-line interrupt service routines sometimes perform crucial operations immediately, at priority 7, then call $INTEN to lower processor priority to device priority.

$INTEN assumes it was called with the following instruction sequence, or its equivalent:

```
JSR         R5,@$INTEN
.WORD       ^C<priority40>&340
```

$INTEN's first action is to save R4 on the job's stack. Since the JSR instruction already saved R5, the job's stack now appears as shown in Table 3–3.

Table 3–3: Job's Stack After $INTEN

Byte Offset	Contents	Agent
0	R4	$INTEN
2	R5	.DRAST macro (JSR R5)
4	PC	interrupt
6	PS	interrupt

Next, $INTEN increments the INTLVL counter from –1 to 0. It saves the job's stack pointer in a memory location and switches to the system stack. $INTEN then lowers processor priority to device priority, and calls the device handler or interrupt service routine back as a co-routine. The interrupt service routine continues to execute in system state.

3.4.1.2 Switching to System State Synchronously — The monitor switches to system state synchronously — that is, without depending on an interrupt — whenever other monitor routines need to go to system state temporarily to ensure the integrity of a certain operation. In these circumstances, the monitor routines can call the $ENSYS routine to switch to system state.

In special circumstances, a routine in a running job (rather than in the monitor) needs to switch to system state. The routine can do this by artificially mimicking an interrupt and using the .INTEN macro to call the $INTEN monitor routine.

$ENSYS Monitor Routine

The $ENSYS routine is voluntarily and synchronously called by any other monitor routine that needs to switch to system state. $ENSYS mimics an interrupt by altering the job's stack so it duplicates the stack condition immediately after an interrupt. Routines call $ENSYS by using the following instructions:

```
JSR     R5,$ENSYS
.WORD   <return address>-.
.WORD   340
```

The instructions following the call to $ENSYS execute in system state. When the routine that must execute in system state completes, it issues an RTS PC instruction. Control then passes in user state to the routine specified in the calling sequence as <return address>.

Table 3–4 shows how $ENSYS manipulates the stack to imitate an interrupt.

Table 3–4: Job's Stack After $ENSYS

Byte Offset	Contents
0	R5
2	return address
4	0

.INTEN Macro

When a routine in a user job needs to switch to system state, it can use a procedure similar to $ENSYS, which is used solely by monitor routines. Essentially, the routine must push the PS and PC onto the stack, and then call the monitor $INTEN routine with a JSR R5 instruction, which puts R5 on the stack as well.

A device handler or a user program subroutine can use the following instructions to switch to 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
```

This routine must be executed with a return address on the top of the stack.

3.4.1.3 Returning to User State — Any routine that is executing in system state issues an RTS PC instruction when it completes. The monitor "unwinds" its stack from one or more interrupts as each RTS PC instruction is issued. As each routine completes, the monitor decrements the INTLVL counter.

When INTLVL is greater than 0, it indicates that the routine that was just interrupted was executing in system state. The monitor defers some special chores until it is just about to return to user state. If it is time to decrement INTLVL after an RTS PC instruction, and the value of INTLVL is currently 0, the monitor knows that it is about to drop back to user state. At this time, there are four special considerations for the monitor:

- Is there an outstanding fork routine? (Fork routines run before jobs or their completion routines.)
- Is a scheduling pass required? (As a result of an interrupt, a job that was previously blocked may now be runnable.)
- Are there outstanding clock ticks? (The monitor may need to normalize its time of day counter and check the timer queue.)
- Is there an outstanding floating-point interrupt?

After taking these considerations into account, the monitor is ready to return to user state. It decrements INTLVL to –1 and switches to the appropriate job's stack. It restores R4 and R5, and then executes the RTI instruction to begin execution in user state.

3.4.2 Context Switching

Context switching occurs as a result of the scheduler's command to run a different job. Its purpose is to restore the context for a job so that it can run. Context switching can occur for one of two reasons:

- The current job becomes blocked and a lower-priority job is runnable.
- A higher-priority job than the current job becomes runnable.

Note that the RT–11 monitor never saves a job's context simply because it switches to system state. For example, if there is only one job running, the monitor does not bother to save or restore its context. A job's context is only significant when there are two or more jobs running. Many other multi-user operating systems switch out the user job every time they leave user state and enter system state. RT–11 avoids the overhead of unnecessary context switching by saving and restoring the context only when it runs a different job. This is a significant saving because there are many situations in which a job is running, an interrupt triggers a switch to system state, and control passes back to the same job once the interrupt is serviced.

When the monitor saves a job's context, it saves the job-dependent information on the job's stack and in the job's impure area. The following information is saved in a context switch:

- PS
- PC
- Stack Pointer (saved in the impure area)
- Registers R0 through R5
- Kernel PAR1 (XM only)
- Memory management fault trap vector (XM only)
- BPT vector (XM only)
- IOT vector (XM only)
- TRAP vector
- System communication area (locations 40–52)
- Location 56 (multi-terminal systems only)
- FPP status word and floating-point registers (if floating-point hardware present)
- All data specified by the program in a .CNTXSW programmed request
- Stack and impure area (which are, of course, part of the job)

When the monitor switches in the new job's context, it tests for a pending completion routine by checking a status bit in I.STATE. If the job's completion queue has a completion queue element on it, the monitor puts a pseudo-interrupt on the job's stack to call the completion queue manager when the scheduler actually starts up the job.

3.4.3 Blocking Conditions

A running job is *blocked* if it cannot proceed until some asynchronous event happens. Table 3–5 lists the blocking conditions, the bits in I.BLOK (at impure area fixed offset 36 octal) that reflect the conditions, and the events that unblock a job. Unused bits are reserved for future use by DIGITAL.

Note that there is no bit that indicates that a job is waiting for a queue element. This is a special case and the monitor handles it by checking the list of available queue elements. If there are none, it requests a scheduling pass to give a lower-priority job a chance to run. The monitor continues to check the available list until a queue element becomes available.

Table 3–5: Blocking Conditions

Blocking Agent	I.BLOK Bit, Name, and Mask	Unblocking Agent
Any request that uses the USR; any monitor command; an exit from a background job.	4 USRWT$ 20	The USR release routine, DEQUSR, when the USR is free and no higher-priority job needs it.
The keyboard monitor SUSPEND command.	6 KSPND$ 100	The Keyboard Monitor, when an operator types the RESUME command.
The .EXIT request; a job that aborts.	8 EXIT$ 400	I/O completion from device handlers, when the job's total I/O count is 0.
Termination of the foreground or system job.	9 NORUN$ 1000	None. Only the Keyboard Monitor can clear this bit by removing the job image from memory.
The .SPND or the .TWAIT programmed request.	10 SPND$ 2000	The monitor's .RSUM processor, when the .RSUM request executes or a .TWAIT completion routine runs.
The .READW, .WRITW, .WAIT, .SDATW, .RCVDW, .MWAIT programmed requests.	11 CHNWT$ 4000	I/O completion from device handlers, when the I/O count for the specified channel is 0.
The .EXIT programmed request issued from a foreground or system job; the .MTSET request issued for a DZ line; .MTDTCH issued for any terminal but a shared console.	12 TTOEM$ 10000	The monitor's terminal service output routine, when the output ring buffer is empty or CTRL/O is typed.
The .TTYOUT, .PRINT, .MTOUT, and .MTPRNT programmed requests.	13 TTOWT$ 20000	The monitor's terminal output interrupt service routine, when there is room in the output ring buffer.
The .TTYIN request (with JSW bit 6 clear); the .CSIGEN, .MTIN, .CSISPC, and .GTLIN programmed requests.	14 TTIWT$ 40000	The monitor's terminal input interrupt service routine, when a line or character is available.
Any request that needs a queue element when none is available.	none	The monitor's queue element return routine, when a queue element becomes free.

3.4.3.1 How the Monitor Blocks a Job — A job becomes blocked when it encounters any of the circumstances listed in Table 3–5. These circumstances are brought about when one of the three following events occurs:

- The job issues one of the programmed requests listed in Table 3–5.

- The monitor SUSPEND command is typed.
- The job aborts.

Typically the job, which is running in user state, issues a programmed request, such as .EXIT. The monitor remains in user state while it processes the programmed request. It then checks to see if the job is waiting because of a blocking condition. The .EXIT request, for example, must wait for all the job's I/O requests to complete before it actually terminates the job. Since waiting for all I/O to complete is a blocking condition, the monitor initiates the appropriate test to see if there are outstanding I/O requests and this job is now blocked.

The monitor calls its $SYSWT routine whenever it needs to determine whether or not a job is blocked. The monitor passes to $SYSWT a bit mask for the bit in I.BLOK corresponding to this particular condition. (Table 3–5 lists the bit masks for I.BLOK; bit 8 corresponds to the .EXIT request condition.) It also passes a *decision subroutine*, which is a routine that determines whether or not a job is blocked for a particular reason. There is a unique decision subroutine for each call to $SYSWT, except the waiting for a queue element condition, which has none. The decision subroutine returns with the carry bit set if the job is indeed blocked. Note that a job can be blocked for only one reason at a time.

When control eventually returns to the job, it executes within the monitor in user state at $SYSWT again. (That is, the monitor runs under the auspices of the job, executing code on its behalf.) The blocking condition must be checked once more in order to reblock a job that may have been unblocked to allow a completion routine to run. (Completion routines are part of a job, but they can run even if the main part of the job is blocked. The monitor unblocks the job to run the completion routine, then runs $SYSWT to reblock the job when the completion routine finishes. Section 3.4.5 discusses the implications of completion routines for scheduling.)

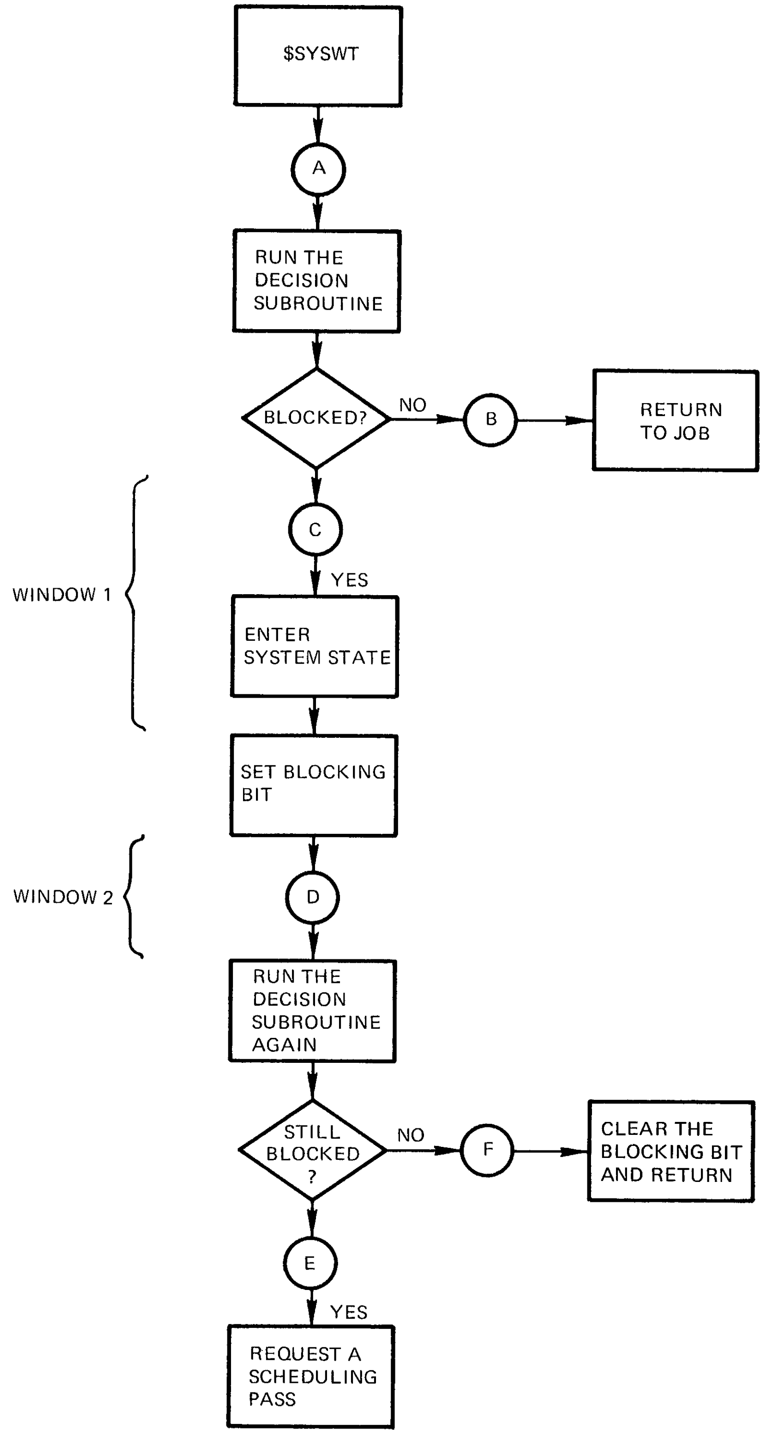
3.4.3.2 $SYSWT Monitor Routine — $SYSWT is the monitor routine that decides whether or not a job is blocked. If a job is blocked, $SYSWT sets the appropriate blocking bit. The flowchart in Figure 3–18 shows how $SYSWT works.

First, $SYSWT runs the decision subroutine passed by the monitor to determine whether or not the job is blocked for a specific reason (point *A* in Figure 3–18). If the job is not blocked, control returns to the job and it continues to run (point *B*). In the .EXIT case, for example, a job is not blocked if there is no pending I/O to delay the exit procedure.

If the job is blocked, $SYSWT calls $ENSYS to enter system state (point *C*). Then it sets the appropriate blocking bit. In the .EXIT example, a job is blocked if there are pending I/O requests; $SYSWT sets the EXIT$ bit, bit 8, in I.BLOK.

Next, $SYSWT runs the decision subroutine again. If the job is still blocked, $SYSWT requests a scheduler pass (point *E*). It does this to give a runnable lower-priority job a chance to execute.

Figure 3–18: $SYSWT Monitor Routine



If the job is no longer blocked, $SYSWT clears the blocking bit and returns (point *F*). When the monitor switches back to user state, the scheduler runs if a scheduling pass is pending. When control finally returns to this job (the one for which $SYSWT originally ran), the monitor continues execution on the job's behalf at the beginning of the $SYSWT routine (point *A*).

$SYSWT runs the decision subroutine twice because interrupts can occur while $SYSWT is running. Since an interrupt can signal the removal of a blocking condition, the job's status can change even as $SYSWT is trying to determine it.

An interrupt can occur after the decision subroutine (point *A*) declares a job to be blocked, but before $SYSWT sets the blocking bit. This time interval is shown as "Window 1" in Figure 3–18. In this situation $SYSWT sets the blocking bit erroneously. But, when it runs the decision subroutine the second time, it discovers that the job is not blocked anymore. $SYSWT clears the bit and returns to the job (point *F*).

"Window 2" in Figure 3–18 indicates the second time interval in which an interrupt can occur. The interrupt can remove the blocking condition immediately after $SYSWT correctly sets the blocking bit. In this case, the monitor's UNBLOK routine clears the blocking bit and requests a scheduling pass because this job became runnable. Control returns to $SYSWT (point *D*), which runs the decision subroutine again. Since the job is no longer blocked, execution leaves $SYSWT (point *F*) and the scheduler runs immediately before the monitor returns to user state.

3.4.3.3 How the Monitor Unblocks a Job — An asynchronous event initiates the monitor's procedure to unblock a job. Table 3–5 lists the significant events that can unblock a job. The completion of all I/O for a specific channel is a significant event, for example, and unblocks a job whose CHNWT$ bit is set.

When an interrupt occurs, control passes to an interrupt service routine. The interrupt routine enters system state by executing the $INTEN monitor routine. Then the interrupt service routine assesses the meaning of the interrupt and takes appropriate action. In a device handler, for example, an interrupt can indicate that an I/O transfer is complete. The handler returns to the monitor to remove the current element from the I/O queue.

In all cases, the monitor clears the blocking bit and requests a scheduling pass if the significant event removes a blocking condition.

3.4.4 Scheduler Operations

The scheduler runs only if there is an outstanding request for a scheduling pass. The monitor checks a flag byte called INTACT each time it is ready to switch from system to user state. If INTACT is not equal to zero, the scheduler runs.

3.4.4.1 How the Monitor Requests a Scheduling Pass — The monitor requests a scheduling pass by calling the $RQTSW monitor routine. It does this whenever a job's ability to run changes. (That is, whenever a running job becomes blocked, or whenever a blocked job becomes runnable.)

3.4.4.2 Characteristics of a Runnable Job — A job that does not have any blocking bit set is runnable. However, there is one circumstance in which a job with a blocking bit set can still be runnable. A job's completion routine can run even though the mainline program is blocked. Section 3.4.5 discusses scheduling implications for completion routines.

3.4.4.3 $RQTSW Monitor Routine — The $RQTSW routine posts a request for a scheduling pass for a specific job by placing a value in the flag byte, INTACT. INTACT holds the job number of the highest-priority job that requested a scheduling pass. $RQTSW ignores a scheduling request for a job if its priority is lower than that of the running job. When a job whose priority is higher than that of the running job requests a scheduling pass, $RQTSW saves the job's number in INTACT, which holds the number in the following format:

$$\text{INTACT} = \frac{\text{Job number}}{2} + 200$$

3.4.4.4 How the Scheduler Works — The scheduler runs just before the monitor returns to a job. Remember that INTLVL, the interrupt level counter, is 0 when it is time to return to user state.

A scheduling pass needed to make a job runnable happens asynchronously, as a result of an interrupt that removed a blocking condition. A scheduling pass needed to make the current job non-runnable happens synchronously, after a job issues a programmed request, after the monitor SUSPEND command is typed, or after a job aborts.

The scheduler runs only if INTACT is not equal to 0. When INTACT is 0, it indicates that no job changed its status, and, therefore, the same job that was interrupted should run again. When INTACT is not 0, it contains the number of the highest-priority job that changed its status. The scheduler runs only if the job number in INTACT is greater than the current number of the current job, which is kept in JOBNUM in the monitor.

The scheduler examines jobs in order of descending priority. It starts with the job whose number is in INTACT, which is not necessarily the highest-priority job in the system. As soon as the scheduler finds a runnable job, the monitor switches context and runs the job. If no jobs at all are runnable, the system idles — that is, it runs the null job briefly, then scans all jobs again for runnability.

3.4.5 Implications for Completion Routines

A job's completion routine can run even though the mainline program is blocked. When an asynchronous event occurs, such as the completion of an I/O request, the interrupt service routine enters system state through the $INTEN monitor routine. The device handler's interrupt service routine returns to the monitor when I/O completes, so the monitor can remove the

I/O queue element from the device handler's queue. If the I/O request specified a completion routine address, the monitor changes the I/O queue element into a completion queue element and puts it on the job's completion queue. The monitor sets bit 7 in the job state word (Iindicate that a completion routine is pending.

As the monitor switches from system to user state, it checks the completion pending bit in I.STATE. If a routine that just ran in system state queued one or more completion routines for this job and the job is not currently running a completion routine, the monitor clears the blocking bit so the scheduler can run the job. This action permits completion routines to execute even though the mainline program is blocked.

When all the completion routines finish, the mainline program begins to execute. However, since it was recently blocked, the monitor executes for the job at the start of the $SYSWT routine. $SYSWT runs the relevant decision subroutine (the routine for the condition that originally blocked this job) and reblocks the job, if necessary.

3.5 System Jobs

Through the system generation process you can create an FB or XM monitor that is capable of running up to six simultaneous jobs in addition to a foreground job and a background job. RT–11 offers the system job feature in order to make two valuable system jobs available: the error logger, called EL, and the file queuing program, called QUEUE. You can run either system job as the foreground job in an RT–11 FB or XM system that does not have the system jobs feature.

Keep in mind that even though RT–11 permits up to eight jobs to run simultaneously, this feature does not mean that RT–11 is a "multi-user" system in any sense of the term. The system jobs RT–11 provides are designed to monitor hardware reliability and to write files to peripheral devices through a queue mechanism. Both jobs are in keeping with the philosophy that RT–11 is essentially a single-user system, and RT–11 still provides no protection for one job from another, or for the operating system software from any job. In the few cases where RT–11 appears to support multiple users, a single application program or language processor that supports multiple terminals is actually running. In Multi-User BASIC–11, for example, the BASIC–11 interpreter is the single user, and it alone is responsible for preserving the integrity of each programmer's work space.

The Resident Monitor in a system job environment is approximately 300 decimal words larger than an equivalent monitor that does not support system jobs. DIGITAL does not encourage customers to write their own system jobs; it reserves the remaining four potential system jobs for future use.

3.5.1 Characteristics

System jobs are similar to ordinary foreground jobs in that, for both kinds of jobs, object code must be stored in relocatable object file format. In addition, system jobs are subject to the same restrictions as foreground jobs — that is, they use restricted arithmetic with global variables.

3.5.2 Logical Names

You reference a system job by its logical name, which, by default, is its file name. However, you can assign a new name when you start the job by using the SRUN monitor command with the /NAME:logical-job-name option. Logical job names must be unique.

The foreground and background jobs have default logical names as well as their actual file names. For the foreground, the default logical name is *F*; for the background, it is *B*. *F* and *B* are permanently assigned; you cannot use them for system jobs. In addition, *EL* is the logical job name permanently assigned to the error logger system job. You can assign another logical name to the foreground job, in addition to *F* by using the FRUN monitor command with the /NAME:logical-job-name option.

The job name is stored in ASCII at offset I.LNAM in the job's impure area.

3.5.3 Job Number

In an FB or XM system without the system job feature the background job number is 0 and the foreground job number is 2. In an environment that supports system jobs, the background job number is still 0, but the foreground job number is always 16 octal. By default, each system job takes the next highest available job number. Job numbers are multiples of 2, and range from 0 to 16 octal. For example, the first system job you start with the SRUN command has a job number of 14, the second system job has a job number of 12, and so on.

3.5.4 Priority

A monitor that supports the system job feature provides the same event-driven, static priority scheduler that ordinary FB and XM systems use. The monitor services jobs according to their priority. The background job always has priority 0, the lowest priority. The foreground job always has the highest priority, which is 7. You cannot change these assignments.

To assign a priority to a system job you can:

1. Use the SRUN command to start the jobs in order of their importance so that the first job you start gets priority 6, the second job gets priority 5, and so on.

2. Explicitly specify the priority when you start the system job. Use the SRUN/LEVEL:priority command to do this. You can specify a priority level for each job in the range 1 through 6, as long as another job is not currently assigned to the level you choose.

The job number is equal to the priority times 2.

NOTE

You can assign a priority only when you start a system job with the SRUN command. The priority levels do not change dynamically, and you cannot change the priority of a job while it is running.

3.5.5 Design Considerations

If you are planning to write or run system jobs, you should keep in mind two major design considerations:

1. RT–11 provides an event-driven, static priority scheduler.
2. Addressing space is at a premium in an RT–11 environment, and certain parts of each job must reside in low (rather than extended) memory.

3.5.5.1 Scheduling Considerations — The RT–11 scheduler arbitrates the demands jobs make for CPU time, awarding the use of system resources to the highest-priority job that is not blocked. Thus, a compute-bound job can lock out all the jobs with a lower priority. On the other hand, an I/O-bound job, such as the RT–11 QUEUE program, is often blocked waiting for I/O transfers to complete. As a result, it does not interfere significantly with lower priority jobs. If you are running a text editor in the background, for example, the fact that the QUEUE program is active is practically transparent to you.

When you design a program to run as a system job, then, consider carefully how often it will require system resources. Keep in mind, too, the fact that RT–11 does not permit parallel use of the USR by two or more jobs. Write the program in such a way that it does not monopolize the system and lock out other jobs.

3.5.5.2 Space Considerations — In an FB system, the main concern is that the number and size of jobs is limited by the amount of space available. As Chapter 2 explains, KMON and the USR slide down in memory each time you load a foreground job, a device handler, or a system job above them. However, KMON cannot slide below location 1000 octal. Since the FB monitor and KMON are about 4K words in size each, this leaves about 20K words of memory for foreground jobs, device handlers, and system jobs. Each job carries a fixed overhead of roughly 220 decimal words for the impure area and channel space.

XM systems have more restrictions that apply to foreground and system jobs. First, the USR is always resident in XM. In addition, the USR cannot slide down in memory into the area mapped by kernel PAR1 (addresses 20000 through 40000). That is, the USR must not slide below location 40000 in low memory. As a result of these two restrictions, about 11K words of memory are available for foreground jobs, device handlers, and system jobs in an XM environment. Each job carries a fixed overhead of approximately 340 decimal words for the impure area and channel space.

However, the XM environment provides other means to load and execute jobs. The only parts of foreground and system jobs that must reside in low memory are the impure area, queue elements, channels, and interrupt service routines. (Like the USR, these four parts of a job cannot reside in the PAR1 area.) The XM system provides three ways to make use of extended memory (memory above the 28K-word boundary) for foreground and system jobs:

1. Use the XM .SETTOP feature in your program.
2. Segment your program and use the /V linker option to make the overlays resident in extended memory.
3. Use the memory management programmed requests in a MACRO program to increase the program's physical address space.

These methods provide the means to circumvent the XM restrictions and execute code in extended memory. They are described in detail in Chapter 4.

3.5.6 Programmed Requests

Two programmed requests — .GTJB and .CHCOPY — have optional arguments that are meaningful only in an FB or XM environment with the system job feature. The .GTJB request obtains job status information for any job in the system. You can reference another job by either logical job name or job number. The .CHCOPY request opens a channel for input, logically connecting it to a file that is currently open for another job for input or output. See the *RT–11 Programmer's Reference Manual* for a detailed explanation of these requests.

3.5.7 Message Handling

In addition to the .SDAT/.RCVD/.MWAIT system through which foreground and background jobs communicate with each other, RT–11 provides an easy way for all jobs, including system jobs, to send and receive messages. The message handling system is implemented through the message queue, or MQ, handler. This handler is a part of the Resident Monitor for all FB and XM systems, whether or not they include the system job feature. The MQ handler is written as an RT–11 device handler for a "special" device. This means that the pseudo-device has a non-RT–11 format. The MQ handler does not accept .SPFUN calls. One advantage of using a device handler in the message system is that you can still debug the send/receive mechanism if one of the jobs involved in the system is not in memory.

For most other purposes, the MQ handler performs like the other RT–11 device handlers except that it communicates with a job, not a device. Essentially, it makes another job appear to be a peripheral device. As a result, you can open a channel to any other job by using a special .LOOKUP programmed request format, described in the *Programmer's Reference Manual*. You can send a message by issuing a .WRITx request. Then you can receive a message to the job by using a .READx request. The first word of the received data buffer contains a count of the words transferred.

A further difference between other RT–11 device handlers and the MQ handler becomes apparent when a job exits (with the .EXIT programmed request) or when it aborts (because of CTRL/C or a fatal monitor error). The monitor allows outstanding I/O requests that are queued for the job to complete, but discards any messages that are queued for the job by examining the queue for the MQ handler and removing queue elements that send messages to the job.

The XM monitor normally uses a special internal macro to transfer message data via the MTPI instruction. This procedure is slow, but safe, since it does not use a PAR to map any buffers. You can use a faster, but more restrictive, transfer procedure by setting the conditional assembly symbol MQH$P2 equal to 1. When the MQ handler is assembled, the assembler will generate code which uses kernel PAR2 to map the user buffers. In this case, all the kernel PAR1 restrictions also apply to PAR2. So, the USR, queue elements, channels, and interrupt service routines cannot reside within locations 20000 through 60000 in a system that actually uses the MQ handler. Note that the QUEUE program uses the MQ handler.

3.5.8 Monitor Commands

The collection of monitor commands has some special features that reflect the system job environment. This section describes them briefly. See Chapter 4 of the *RT–11 System User's Guide* for a complete description.

3.5.8.1 SRUN and FRUN Commands – Use the SRUN command to start execution of a system job. You can also use the FRUN command to begin execution of a system job in the foreground partition.

NOTE

> If you use SRUN or FRUN to start a system job and a job with the same name is already in memory but has finished executing, the monitor unloads the job in memory and brings in a new copy from a peripheral device.

3.5.8.2 LOAD and UNLOAD Commands – Use the LOAD command to bring a device handler into memory and to assign ownership of a peripheral device to a specific job. Different jobs can own different units of a file-structured device. Since a system job must already be in memory before you can assign a device to it, remember to start the job with SRUN before you use the LOAD command. If the job will not run without the handler, use the /PAUSE option with the FRUN or SRUN command. Note that you cannot assign ownership of SY or MQ.

The UNLOAD command removes a device handler or a system job from memory. You should type a colon (:) after the name of the device handler to distinguish it from the name of a system job. If a colon is not included, the UNLOAD command attempts to unload a system job of the specified name. If none is found, the command attempts to unload a device handler with that name. For example, RK could be both the name of a system job and the name of a device handler. To remove the device handler, type:

```
UNLOAD RK:
```

To unload the system job, type:

```
UNLOAD RK
```

3.5.8.3 SUSPEND and RESUME Commands

Use the SUSPEND command to stop execution of a system job.

Use the RESUME command to continue execution of a system job that was stopped by the SUSPEND command or the /PAUSE option for SRUN or FRUN.

3.5.8.4 SHOW JOBS Command

Use the SHOW JOBS command to display status information about all system jobs currently in the system.

3.5.8.5 SET TT: NOFB Command

Use the SET TT: NOFB command to disable the special control keys CTRL/F, CTRL/B, and CTRL/X you use to communicate with foreground, background, and system jobs.

3.5.9 Communicating with a System Job

In a system job environment you use CTRL/X to communicate with a system job in much the same way that you use CTRL/F for a foreground job and CTRL/B for a background job. By directing input to the correct job and by labeling output, this mechanism permits two or more jobs to share one terminal. When you type CTRL/X, the monitor sends a carriage return/line feed combination to the terminal, followed by the *Job?* prompt. While waiting for your response, the monitor simulates a full output ring buffer. This prohibits output from any other job from garbling the CTRL/X dialogue. (This also blocks a job that is waiting for output.)

Respond to the prompt by typing the job's logical name, followed by a line terminator (carriage return, line feed, or CTRL/Z). DELETE (or RUBOUT) and CTRL/U are valid editing commands in a CTRL/X sequence. Remember that the names *F* and *B* are reserved for the foreground and background jobs. If the job you specify is not running, or does not exist, the monitor prints a question mark (?). As a result of the CTRL/X sequence, the monitor directs terminal input characters to the appropriate job's input ring buffer.

To cancel the CTRL/X sequence before you finish typing the job name, type CTRL/C. This does not abort any job. It simply returns to the state of the terminal before you typed CTRL/X. To actually abort a system job, type CTRL/X followed by the job name and a line terminator. Then type two CTRL/Cs.

While terminal input is directed to one job's input ring buffer, other jobs can still send output characters to the terminal. To avoid confusion, the monitor prints an identifying label every time the output user changes. The terminal identity string is stored at I.JID in each job's impure area and it consists of a carriage return/line feed combination, followed by the job name, a right angle bracket (>), and another carriage return/line feed combination.

The following sequence shows how two system jobs can share one terminal. Type a CTRL/X sequence and send a message to the first job:

```
CTRL/X
Job? SY1(RET)
HELLO TO JOB 1(RET)
```

Job 2 sends a message to the terminal:

```
SY2>
HI FROM JOB 2
```

Send another message to job 1. Note that you do not type the *SY1>* label yourself. The monitor prints it when it echoes your input characters.

```
SY1>
HELLO AGAIN TO JOB 1(RET)
```

Job 2 sends two more messages:

```
SY2>
HI AGAIN FROM JOB 2
HI A THIRD TIME FROM JOB 2
```

Finally, job 1 sends a message:

```
SY1>
HI FROM JOB 1
```

3.5.10 How to Queue Files from an Application Program

Usually you queue files that you want to copy to another device by using the monitor PRINT command. If the QUEUE program is running when you issue the PRINT command, the files you specify are queued automatically and the monitor dot prints on your terminal almost immediately.

Your application programs can also copy files to output devices through the QUEUE program. The method your program must use to do this depends on which monitor is currently running. If an FB or XM monitor that includes the system job feature is running, your program must communicate with QUEUE through the message queue (MQ) handler by using .LOOKUP, .WRITW, and .READW programmed requests. Using the MQ handler is beneficial because it frees the monitor for other tasks, and takes advantage of the existing queued I/O system. Note that the MQ handler in an XM system may borrow kernel PAR2 for its own use if the conditional assembly parameter MQH$P2 = 1; see Section 3.5.7 for more information on this topic.

If an FB or XM monitor without the system job feature is running, your program must communicate with QUEUE through the .SDAT and .RCVD programmed requests.

To queue one or more files, follow these steps:

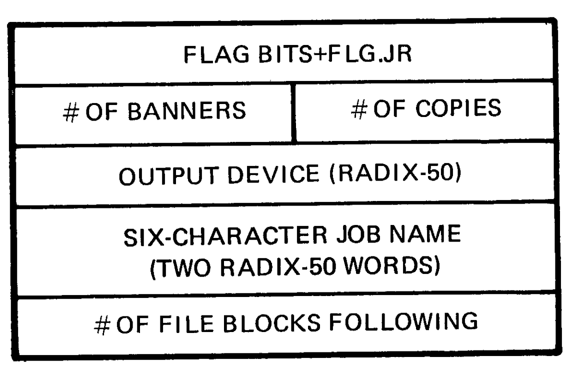
1. Set up a job block in your program for a logical group of files to be queued.
2. Set up a file block for each file to be queued.
3. Issue the .LOOKUP programmed request for the QUEUE program. (Omit this step if your system does not have the system job feature.)
4. Issue the .WRITW request (or the .SDATW request if your system does not have the system job feature) to send the QUEUE request and establish a pointer to the job and file blocks.
5. Issue the .READW request (or the .RCVDW request if your system does not have the system job feature) to receive acknowledgment from QUEUE.

Once QUEUE acknowledges your request, your program is free to continue processing or to exit. Figure 3–23 shows a program that uses .LOOKUP, .READW, and .WRITW to queue one file, then exits.

3.5.10.1 Setting Up the Job Block — Set up a job block in memory for a logical group of files. The job block defines the logical name by which you can later reference the entire group of files.

If you copy files to a file-structured device (rather than to the line printer, for example) all the files belonging to the job are copied and stored in separate files with the input file names and file types. The handler for the device to which you send the job must be made resident in memory through the monitor LOAD command. Figure 3–19 shows the format of the job block.

Figure 3–19: Job Block



The flag word in each job block defines the action QUEUE should take on each file. Table 3–6 lists the definitions of the bits. Bits 4 through 15 are reserved for DIGITAL.

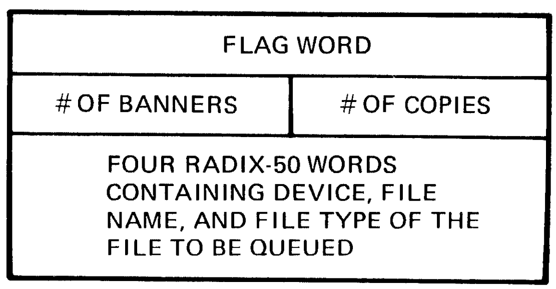
The job block must have bit FLG.JR set. If FLG.CP is set, QUEUE sets the default number of copies to queue for this job from the low byte of the second word in the job block. If FLG.HD is set, QUEUE sets the number of banners to queue for this job from the high byte of the second word in the job block.

Table 3–6: Request Flag Bits

Bit	Name	Mask	Meaning
0	FLG.DE	1	Delete file after copying it.
1	FLG.CP	2	Make multiple copies (get number of copies from second word in block.
2	FLG.HD	4	Create banner pages (get number of pages from second word in block).
3	FLG.JR	10	For initial request and job block.

3.5.10.2 Setting Up the File Block — Immediately after the job block, your program must set up a file block for each file that is part of the job. Arrange the blocks contiguously in memory, with the job block first. Figure 3–20 shows the format of the file block.

Figure 3–20: File Block



In each file block you can specify the number of banner pages and the number of copies for the file by setting flag bits FLG.CP and FLG.HD, and putting values into the second word of the block. If you omit the flag bits, QUEUE ignores the second word of the file block and checks the flag bits of the job block instead. If they are set, QUEUE takes the values from the second word of the file block. Finally, if the flag bits are clear in both the file and the job blocks, QUEUE uses the system default of no banners and one copy of the file, or the current default parameters as set by the QUEMAN /P option.

3.5.10.3 Setting Up the QUEUE Request Block — The last data structure you must establish is called the QUEUE request block. It need not be contiguous in memory with the job and file blocks. Figure 3–21 shows the format of the QUEUE request block. This block contains the information that QUEUE needs to begin processing the files. QUEUE requests can only be issued from a privileged job with kernel mapping. QUEUE request blocks must reside in low memory.

Figure 3–21: QUEUE Request Block

FLG.JR
SIX-CHARACTER FILE NAME OF YOUR PROGRAM (THREE ASCII WORDS)
ADDRESS OF JOB BLOCK
0

3.5.10.4 Issuing the .LOOKUP Request — In the executable section of your program, you must issue a .LOOKUP programmed request to make the first contact with the QUEUE program and establish a communication channel. Issue the .LOOKUP for MQ:QUEUE, following the example provided in Section 3.5.10.7. (Omit this step if your system does not have the system job feature.)

3.5.10.5 Issuing the Request to QUEUE — If the .LOOKUP is successful (or if you omitted it), you next issue the .WRITW programmed request (or the .SDATW request if your system does not have the system job feature) to send your request to QUEUE. The text you send to QUEUE is the QUEUE request block. See the example provided in Section 3.5.10.7.

If your request is valid, QUEUE inserts the request blocks into the queue, which is a workfile on device DK:. The workfile is a first-in/first-out list; it can contain requests for different output devices. QUEUE does not maintain a separate workfile for each device.

3.5.10.6 Receiving Acknowledgment from QUEUE — When QUEUE acknowledges your request, your program can continue execution, or exit, as you desire. You obtain this acknowledgment by issuing the .READW programmed request (or the .RCVDW request if your system does not have the system job feature). QUEUE's response takes the form shown in Figure 3–22.

Your program must wait for this acknowledgment. QUEUE maintains only a limited number of extra queue elements. If QUEUE sends a message to your program that your program is not prepared to accept, a queue element is needlessly kept out of the list of available elements; this could block another job in your system.

Figure 3–22: Request Acknowledgment Block

FLAG BITS
SIX-CHARACTER NAME "QUEUE " (THREE ASCII WORDS)
0
0

If the acknowledgment is positive, the flag word contains 0. If the acknowledgment is negative, the sign bit of the flag word is set in addition to one of the low three bits. Table 3–7 shows the meanings of the acknowledgment flag bits.

Table 3–7: Acknowledgment Flag Bits

Bit	Name	Mask	Meaning
0	FLG.RA	0	Request accepted.
15,0	FLG.IR	100001	Illegal job request.
15,1	FLG.QF	100002	Insufficient room in workfile.
15,2	FLG.NQ	100004	QUEUE being aborted from console.

3.5.10.7 QUEUE Example Program – Figure 3–23 contains a listing of an example program, MYPROG, that uses QUEUE in a system with the system job feature to copy a data file to the line printer.

Figure 3–23: QUEUE Example Program

```
        .TITLE   MYPROG.MAC
        .ENABL   LC
; This example shows how an application program can
; send files through the queue system.

         .MCALL   .READW, .WRITW, .LOOKUP, .EXIT, .PRINT

         ;Flag bits for request

         FLG.DE= 1          ;Delete file after printing
         FLG.CP= 2          ;Multiple copies
         FLG.HD= 4          ;Banner pages
         FLG.JR= 10         ;Job request indicator

         .PSECT   QUETST

;Execution Section
```

(Continued on next page)

Figure 3–23: QUEUE Example Program (Cont.)

```
START:  .LOOKUP #AREA,#16,#LKUP ;.LOOKUP QUEUE
        BCC     1$              ;Error?
        .PRINT  #LUPERR         ;Yes, report it
        .EXIT                   ;and quit

1$:     .WRITW  #AREA,#16,#REQST,#6 ;Send initial
                                ;request to QUEUE
        BCC     2$              ;Error?
11$:    .PRINT  #REQERR         ;Yes, report it
        .EXIT                   ;and quit

2$:     .READW  #AREA,#16,#REPLY,#6 ;Wait for ACK
                                ;from QUEUE.  Word count
                                ;of ACK in REPLY, text
                                ;in REQST.
        BCS     11$             ;Branch on error
        TST     REQST           ;ACK okay?  (First word
                                ;of ACK should be 0)
        BNE     MERR            ;Branch if error
        .PRINT  #ACKMSG         ;Print success message
        .EXIT                   ;End of test, request
                                ;sent to line printer.

MERR:   .PRINT  #NAKMSG         ;Print error message
        .EXIT                   ;and quit

        .PSECT  QUEDTA

;Block for .LOOKUP on QUEUE

LKUP:   .RAD50  /MQ /
        .ASCIZ  /QUEUE/

AREA:   .BLKW   5               ;EMT area

;ACK from QUEUE goes here:
REPLY:  .WORD   0               ;Word count from .READW
REQST:  .WORD   FLG.JR          ;Initial request
        .ASCII  /MYPROG/        ;Calling program
        .WORD   JOBBLK          ;Addr of job block
        .WORD   0               ;End of initial request

;Block for job

JOBBLK: .WORD   <FLG.JR+FLG.HD+FLG.CP> ;Flags for job,
                                ;banners, and copies
        .BYTE   2,3             ;2 copies, 3 banners
        .RAD50  /LP /           ;Send to printer
        .RAD50  /DATA   /       ;Logical job name
        .WORD   1               ;One file follows:
FILBLK: .WORD   0               ;No flags, use defaults
        .BYTE   0,0             ;Default banners, copies
        .RAD50  /DK /           ;Filespec to be queued
        .RAD50  /TSTFIL/
        .RAD50  /DAT/

;Messages

LUPERR: .ASCIZ  /MYPROG-F-QUEUE not running/
REQERR: .ASCIZ  /MYPROG-F-Initial request error/
NAKMSG: .ASCIZ  /MYPROG-W-QUEUE acknowledgment negative/
ACKMSG: .ASCIZ  /MYPROG-I-QUEUE acknowledgment OK/
        kmon.EVEN

        .END    START
```

3.6 Data Structures

The following sections describe some of the data structures in the Resident Monitor.

3.6.1 Fixed Offsets

Some words always have fixed positions relative to the start of the Resident Monitor. These words are called fixed offsets. In general, they contain either status words or pointers to other significant information. The fixed offset area in RMON is located at the start of the RTDATA p-sect.

To access the fixed offsets from a running program, use the .GVAL programmed request, as follows:

```
.GVAL       #area,#offset
```

Here, *area* represents a two-word argument block, and *offset* is a byte offset from Table 3–8. Your programs should never modify the contents of the fixed offsets.

Table 3–8: Resident Monitor Fixed Offsets

Offset	Symbol	Byte Length (Octal)	Description
0	$RMON	4	Common interrupt entry point; contains the instruction JMP $INTEN. The .INTEN macro uses it.
4	$CSW	240	Background job channel area (16 decimal channels; each is five words long).
244	$SYSCH	12	Internal channel used for system functions; the Keyboard Monitor uses this channel.
246		2	SJ only: Reserved.
250		2	SJ only: Reserved.
252	I.SERR/ I.SPLS	2	SJ only: An indicator for hard or soft errors.
254	I.SPLS	2	SJ only.
256	BLKEY	2	Segment number of the directory now in memory. A value of 0 implies that no directory is there. See Section 2.2.3.2 for a method of inhibiting directory caching.
260	CHKEY	2	Device index and unit number of the device whose directory is in memory. The low byte contains the device index into the monitor tables; the high byte is the unit number.
262	$DATE	2	Current date value.

(Continued on next page)

Table 3–8: Resident Monitor Fixed Offsets (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
264	DFLG	2	"Directory operation in progress" flag. This is nonzero to inhibit CTRL/C from aborting a job while a directory operation is in progress.
266	$USRLC	2	Address of the normal USR area. This is where the USR resides when it is called into memory by the background job and location 46 is 0. In other words, the foreground job must provide space for the USR to swap. (Note: if the foreground job calls in the USR and location 46 is 0, the foreground job aborts.) See Chapter 2 for information on USR swapping.
270	QCOMP	2	Address of the I/O exit routine for all devices. The exit routine is an internal queue management routine through which all device handlers exit once the I/O transfer is complete. Any new device handlers you add to RT–11 must also use this exit location; use the .DRFIN macro in your handler to generate the exit code automatically.
272	SPUSR	2	Special device error word. Non RT–11 file-structured devices, such as magtape, use this word to report errors to the monitor.
274	SYUNIT	2	The high byte contains the unit number of the system device. This is the unit number of the device from which the system was bootstrapped.
276	SYSVER	1	Monitor version number. You can always access the version number in this fixed offset to determine if you are using the most recent version of the software. For RT–11 Version V5, this value is 5.
277	SYSUPD	1	Monitor release level. This number identifies the release level of the monitor version specified in byte 276. For RT–11 Version V5, this value is 0.
300	CONFIG	2	Configuration word. These 16 bits indicate information about either the hardware configuration of the system or a software condition. Another configuration word located at fixed offset 370 contains additional data. See Section 3.6.1.1 for the meaning of each bit.
302	SCROLL	2	Address of the VT11 scroller.
304	TTKS	2	Address of the console keyboard status register. The default value is 177560. See Chapter 5 for details on changing the hardware console interface to another terminal.
306	TTKB	2	Address of the console keyboard buffer register. The default value is 177562.

(Continued on next page)

Table 3–8: Resident Monitor Fixed Offsets (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
310	TTPS	2	Address of the console printer status register. The default value is 177564.
312	TTPB	2	Address of the console printer buffer register. The default value is 177566.
314	MAXBLK	2	The maximum file size allowed in a 0 length .ENTER programmed request. The default value is 177777 octal blocks, allowing an essentially unlimited file size. You can change this value from within a running program (although this is not recommended), or by using SIPP to patch this location.
316	E16LST	2	Offset from the start of RMON to the dispatch table for EMTs 340 through 357. The BATCH processor uses this.
320	CNTXT	2	FB and XM only: A pointer to the impure area for the current executing job.
322	JOBNUM	2	FB and XM only: The executing job's number.
320	$TIME	4	SJ only: Two words of time of day.
322	$TIME 2	2	
324	SYNCH	2	Address of monitor routine to handle .SYNCH requests. Your interrupt routines can issue the .SYNCH programmed request, which enters the monitor through this address to synchronize with the job they are servicing.
326	LOWMAP	24	Start of the low-memory protection map. This map protects vectors at locations 0 through 476. See Section 3.6.1.2 for more information on the low-memory bitmap.
352	USRLOC	2	A pointer to the current entry point of the USR. This may be 0, if the USR is not in memory; it may be the relocation code in USRBUF, if the USR was just brought into memory; it is the processing code, in all other cases.
354	GTVECT	2	Address of VT11 or VS60 display processor display stop interrupt vector (default is 320).
356	ERRCNT	2	Low byte is the error count byte for use by system utility programs. The high byte is reserved.
360	$MTPS	2	Entry point of the move to PS routine. The .MTPS macro calls this routine to perform processor independent moves to the Processor Status word.
362	$MFPS	2	Entry point of the move from PS routine. The .MFPS macro calls this routine to do processor independent moves from the Processor Status word.

(Continued on next page)

Table 3–8: Resident Monitor Fixed Offsets (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
364	SYINDX	2	Index into the monitor device tables for the system device. See Section 3.6.5 for information on the device tables.
366	STATWD	2	Indirect file and monitor command state word.
370	CONFG2	2	Extension configuration word. This is a string of 16 bits indicating the presence of an additional set of hardware options on the system. See Section 3.6.1.3 for the meaning of each bit.
372	SYSGEN	2	System generation features word. The bits in this word indicate the presence or absence of some system generation special features. See Section 3.6.1.4 for the meaning of each bit.
374	USRARE	2	Size of the USR in bytes. Your program can use this information to dynamically determine the size of the region you need in order to swap the USR. (The USR is always resident in XM systems.)
376	ERRLEV	1	Error severity at which to abort indirect files. You can change this level with the SET ERROR command. The default setting is ERROR. See Chapter 2 for more information.
377	IFMXNS	1	Depth of nesting of indirect files. The default nesting level is 3. You can change this value by using SIPP to patch this location. Be sure to refer to offset 377 as a byte, not as a word.
400	EMTRTN	2	Internal offset for use by BATCH only.
402	FORK	2	Offset to fork processor from the start of the Resident Monitor. (Location 54 contains the starting address of RMON.) Use the .DREND macro in your device handler to automatically set up a pointer to the fork processor.
404	PNPTR	2	Offset to the $PNAME table from the start of the Resident Monitor.
406	MONAME	4	Two words of Radix–50 containing the name of the current monitor file.
412	SUFFIX	2	One word of Radix–50 containing the suffix used by the current monitor to name device handlers. For SJ and FB systems, this word is normally blank. For XM, it is normally *X*, right-justified. This word is set up by the bootstrap; you can modify it there (see the *RT–11 System Generation Guide* for details).
414	DECNET	2	Reserved.
416	EXTIND	1	IND stored error byte.

(Continued on next page)

Table 3–8: Resident Monitor Fixed Offsets (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
417	INDSTA	1	IND control status byte. The following bits are defined: 40 LN$IND Set if current line passed by IND 100 IN$RUN Set if KMON issued RUN of IND 200 IN$IND Set if IND active
420	$MEMSZ	2	Total physical memory available, in 32-word blocks.
422		2	Reserved.
424	$TCFIG	2	Address of terminal SET option status word.
426	$INDDV	2	Pointer to ASCII device name and unit number of IND.SAV.
430	MEMPTR	2	Offset to memory control block pointers.
432	P1EXT	2	Pointer to $P1EXT routine (refer to Section 7.9.7 for details).

3.6.1.1 Configuration Word — The configuration word, CONFIG, indicates information about either the hardware configuration of the system or a software condition. Table 3–9 lists the bits and their meanings. Unused bits are reserved for future use by DIGITAL.

Table 3–9: The Configuration Word, Offset 300

Bit	Meaning
0	0 = SJ Monitor. 1 = (If bit 12 = 0): FB Monitor. (If bit 12 = 1): XM Monitor.
1	1 = KMON fetches SL handler and uses single-line editor.
2	1 = VT11 or VS60 graphics display hardware exists.
3	1 = BATCH is in control of the background.
4	1 = Single-line editor is available to user programs.
5	0 = 60-cycle clock. 1 = 50-cycle clock. The value of bit 5 is patchable to indicate the current line frequency.
6	1 = FP11 floating-point hardware exists.
7	0 = No foreground or system job is in memory. 1 = A foreground or system job is in memory.

(Continued on next page)

Table 3–9: The Configuration Word, Offset 300 (Cont.)

Bit	Meaning
8	1 = User is linked to the graphics scroller.
9	1 = USR is permanently resident, via SET USR NOSWAP. (USR is always resident in XM and this bit is always set.)
10	1 = The QUEUE program is running.
11	1 = Processor is a PDP–11/03. The Processor Status word on this system cannot be accessed by means of an address in the I/O page.
12	1 = A mapped system is running under the XM monitor.
13	1 = The system clock has a status register.
14	1 = A KW11-P clock exists and programs can use it.
15	1 = There is a system clock (L clock, P clock, or 11/03–11/23 line-frequency clock).

3.6.1.2 Low-Memory Protection Bitmap — RT–11 maintains a bitmap that reflects the protection status of low memory, locations 0 through 477. This map is required in order to avoid conflicts in the use of the vectors. In FB and XM, the .PROTECT programmed request allows a program to gain exclusive control of a vector or a set of vectors. When a vector is protected, RMON updates the bitmap to indicate which words are protected. If a word in low memory is not protected, it is loaded from block 0 of the executable file. If a word in low memory is protected, it is not loaded from block 0 of the file. In addition, if the word is protected by a foreground job, it is not destroyed when you run a new background program.

The bitmap is a 20-byte decimal table that starts 326 octal bytes from the beginning of the Resident Monitor. Table 3–10 lists the offset from RMON and the corresponding locations represented by that byte.

Table 3–10: Low-Memory Bitmap

Offset	Locations (Octal)	Offset	Locations (Octal)
326	0–17	340	240–257
327	20–37	341	260–277
330	40–57	342	300–317
331	60–77	343	320–337
332	100–117	344	340–357
333	120–137	345	360–377
334	140–157	346	400–417
335	160–177	347	420–437
336	200–217	350	440–457
337	220–237	351	460–477

Each byte in the table reflects the status of eight words of memory. The first byte in the table controls locations 0 through 17, the second byte controls locations 20 through 37, and so on. The bytes are read from left to right. Thus, if locations 0 through 3 are protected, the first byte of the table contains 11000000.

NOTE

> Only words are protected, not individual bytes. Thus, protecting word 0 means that bytes 0 and 1 are both protected.

If locations 24 through 27 are protected, the second byte of the table contains 00110000.

The leftmost bit of each byte represents lower memory locations; the rightmost bit represents higher memory locations. For example, to protect locations 300 through 307, the leftmost four bits of the byte at offset 342 must be set to result in a value of 360 for that byte: 11110000.

The SJ monitor does not support the .PROTECT programmed request. If you need to protect vectors in SJ, either use SIPP to manually modify the bitmap or dynamically modify the bitmap from within a running program.

For example, the following instructions protect locations 300 through 306 dynamically:

```
MOV @#54,R0
BISB #^B11110000,342(R0)
```

Protecting locations with SIPP means that the vector is permanently protected, even if you rebootstrap the system. The dynamic method provides a temporary measure and does not remain effective across bootstraps. Be aware that the dynamic method involves storing data directly into the monitor. For this reason, DIGITAL recommends that you use SIPP to protect vectors in SJ.

The RT–11 monitor uses the low-memory bitmap to automatically protect some locations in low memory. The locations it protects are as follows:

0–16
24–32
50–66
100–102 (line-frequency clock)
104–106 (if KW11-P selected as system clock)
114–116
244–246
250–252 (for XM systems only)
The system device handler interrupt vector
Interrupt vectors for loaded device handlers
Vectors for all interfaces supported in a multi-terminal system

NOTE

Vectors of device handlers that you load with the LOAD command are protected; vectors of device handlers that you bring into memory with the .FETCH programmed request are not protected.

3.6.1.3 Extension Configuration Word — The extension configuration word, CONFG2, indicates the presence of an additional set of hardware options on the system. Table 3–11 lists the bits and their meanings. Unused bits are reserved for future use by DIGITAL.

Table 3–11: Extension Configuration Word, Offset 370

Bit	Meaning
0	1 = Cache memory is present.
1	1 = Parity memory is present.
2	1 = A readable switch register is present.
3	1 = A writeable console display register is present.
4	1 = A handler used by LD may have been unloaded.
5	1 = Do not swap user code or exit.
6	Reserved.
7	1 = The Commercial Instruction Set (CIS) option is present.
8	1 = The Extended Instruction Set (EIS) option is present.
9	0 = VT11 display hardware exists if bit 2 at offset 300 is set. 1 = VS60 display hardware exists if bit 2 at offset 300 is set.
10–13	Reserved.
14	1 = The processor is a PDP–11/70.
15	1 = The processor is a PDP–11/60.

3.6.1.4 System Generation Features Word — The system generation features word, SYSGEN, indicates which major system generation features are present. Table 3–12 lists the meaning of each bit. Unused bits are reserved for future use by DIGITAL. In addition, do not set or clear any bits in this word yourself.

Note that the values of the first three bits must correspond to the conditional variables you use when you assemble your device handler files. Attempts to use handlers that are not compatible with the monitor cause the *?KMON-F-Conflicting SYSGEN options* error message to appear.

Table 3–12: System Generation Features Word, Offset 372

Bit	Meaning
0	1 = The error logging feature is present.
1	1 = The memory management feature is present.
2	1 = The device I/O time-out feature is present.
3	1 = This is an RTEM–11 system.
4–8	Reserved.
9	1 = The memory parity feature is present.
10	1 = The SJ mark time feature is present.
11–12	Reserved.
13	1 = The multi-terminal feature is present.
14	1 = The system job feature is present.
15	Reserved.

3.6.2 Impure Area

The impure area is an area of memory where the monitor stores all job-dependent data. For each job, the impure area contains job-specific information, such as terminal ring buffers and I/O channels. The monitor sets up the impure area and maintains its contents.

3.6.2.1 Single-Job Monitor Impure Area – In the SJ system, there is no distinct impure area for the single job. Instead, information relating to the job is stored in various places throughout the Resident Monitor.

3.6.2.2 Foreground/Background Monitor Impure Area – In an FB system, the impure areas contain all the information the monitor requires to run two or more independent jobs. The information stored in the impure area is job-specific. The impure area for the background job is located at the start of the p-sect RMON in the Resident Monitor and it is permanently resident. The impure area for a foreground or system job is located in memory below the start of the job itself. The size of the impure area is the value in the global symbol FMPUR, which you can find by looking at your monitor's link map.

The monitor maintains a table of one-word pointers to the impure areas of all jobs in the system. This table is located at $IMPUR, and is either eight or two words long, depending on whether the system job feature is present or not.

In RT–11, a background job is always present. It is the Keyboard Monitor if no other background job exists. The foreground or system job impure area pointer may be 0 if no such job is in memory. When you issue an FRUN command, the monitor creates an impure area for the foreground job. Similarly, the SRUN command creates an impure area for a system job. In both cases, the monitor also updates the job's $IMPUR entry to point to the impure area.

The contents of the impure area are the same for both the background and the foreground jobs, as shown in Table 3–13. The offset in the table is the offset from the start of the impure area itself. In some cases, the contents of the impure area depend on which system generation features you select. These cases are indicated by a "Feature only:" phrase in the "Description" column.

Table 3–13: Impure Area

Offset	Symbol	Byte Length (Octal)	Description
0	I.STATE	2	Job state word bits. See Table 3-14 for the meaning of each bit.
2	I.QHDR	2	Head of available queue element linked list.
4	I.CMPE	2	Last entry in the completion queue.
6	I.CMPL	2	Head of the completion queue.
10	I.CHWT	2	Pointer to channel during I/O wait. When a job is waiting for I/O on a channel to complete, the address of that channel area is stored here.
12	I.PCHW	2	Saved I.CHWT during execution of a completion routine.
14	I.PERR	2	Error bytes 52 and 53 saved during completion routines.
16	I.TTLC	2	Terminal input ring buffer line count (for non-multi-terminal systems).
20	I.PTTI	2	Previous terminal input character (for non-multi-terminal systems).
16	I.CNSL	2	Multi-terminals only: Pointer to terminal control block (TCB) for this job's console terminal.
20	unused	2	Multi-terminals only: Unused.
22	I.TID	2	Pointer to job ID area, later in impure area.
24	I.JNUM	2	Job number of the job that owns this impure area.
26	I.CNUM	2	Number of I/O channels defined. The default is 16 decimal; you can use .CDFN to define more.
30	I.CSW	2	Pointer to the job's channel area.
32	I.IOCT	2	Total number of I/O operations outstanding.
34	I.SCTR	2	Suspension counter. A value less than 0 means the job is suspended.
36	I.BLOK	2	Job blocking bits. See Table 3–15 for the meaning of each bit.

(Continued on next page)

Table 3–13: Impure Area (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
The following offsets are not guaranteed to remain constant from release to release. In fact, since the pointers and status words can vary depending on the special features you select through system generation, you should consult the link map from the monitor assembly to find the correct offsets for your system. Note that some items, such as the input and output ring buffers, have a variable length.			
-	I.JID	10	Job's terminal prompt string. If the system job feature is present, the length of I.JID is 14 octal.
-	I.LNAM	6	System jobs only: Logical job name in ASCII.
-	I.NAME	10	File name and file type, in Radix–50, of the running job.
-	I.SPLS	2	Pointer to nonlinked .DEVICE list.
-	I.TRAP	2	Address of trap to 4 and 10 routine defined via .TRPSET.
-	I.FPP	2	FPU only: Address of FPP exception routine defined via .SFPA.
-	I.SPSV	2	XM only: Bottom of saved SP data.
-	I.SWAP	4	Pointer to extra swap information specified in the .CNTXSW programmed request.
-	I.SP	2	Saved stack pointer.
-	I.BITM	24	Bitmap for protection.
-	I.CLUN	2	Multi-terminals only: LUN of job's console.
-	I.TTLC	2	Multi-terminals only: Terminal input ring buffer line count.
-	I.IRNG	2	Input ring buffer low limit.
-	I.IPUT	2	Input PUT pointer for interrupts.
-	I.ICTR	2	Input character count.
-	I.IGET	2	Input GET pointer for .TTYIN.
-	I.ITOP	2	Input ring buffer high limit.
-	–––	TTYIN	Input ring buffer.
-	I.OPUT	2	Output PUT pointer for .TTYOUT.
-	I.OCTR	2	Output character count.
-	I.OGET	2	Output GET pointer for interrupts.
-	I.OTOP	2	Output ring buffer high limit.

(Continued on next page)

Table 3–13: Impure Area (Cont.)

Offset	Symbol	Byte Length (Octal)	Description
-	---	TTYOUT	Output ring buffer.
-	I.QUE	QWDSIZ	The initial queue element; 16 octal bytes (24 bytes if XM).
-	I.MSG	4	The internal message channel.
-	I.SERR	6	The third word of the message channel is used as the hard/soft error flag.
-	I.TERM	2	Terminal status word.
-	I.TRM2	2	Terminal status word 2.
-	I.SCCA	2	CTRL/C terminal status word set via .SCCA.
-	I.SCCI	2	XM only: PAR1 value of I.SCCA for XM.
-	I.DEVL	2	Pointer to linked .DEVICE list.
-	I.FPSA	2	XM and FPU only: Pointer to FPU save area, later in impure area.
-	I.SCOM	36	XM only: System communication save area (for non-multi-terminal systems).
-	I.SCOM	40	XM and multi-terminals only: System communication save area.
-	I.RSAV	20	XM only: Register save area.
-	I.WPTR	2	XM only: Pointer to window control blocks, at I.WNUM later in impure area.
-	I.RGN	RGWDSZ	XM only: Region control blocks.
-	I.WNUM	2	XM only: Number of window blocks.
-	---	WNWDSZ	XM only: Window control blocks.
-	I.FSAV	62	XM and FPU only: FPU save area.
-	I.VHI	2	XM only: Virtual high limit of job; nonzero if linker /V option used.
-	I.SCHP	2	Pointer to the job's system channel. The monitor uses this channel for its own calls, such as .DSTATUS.
-	I.SYCH	14	The job's system channel, for all foreground and system jobs. The background job's channel is in the fixed offset area of the Resident Monitor.

Job State Word Bits

The job state word, Iindicates status information about a job. Table 3–14 shows the meaning of each bit. Unused bits are reserved for future use by DIGITAL.

Table 3–14: Job State Word Bits, Offset 0

Mnemonic	Bit	Meaning When Set
ABPND$	0	An abort has been requested for this job.
BATRN$	1	BATCH is running for this job.
CSIRN$	2	The CSI is running for this job.
USRRN$	3	The USR is running for this job.
	4	Reserved.
ABORT$	5	The job is being aborted.
	6	Reserved.
CPEND$	7	This job has a completion routine pending.
	8–11	Reserved.
WINDW$	12	This is a virtual job.
	13–14	Reserved.
CMPLT$	15	A completion routine is running for this job.

Job Blocking Bits

The job blocking word, I.BLOK, indicates which condition is blocking a job. Unused bits are reserved for future use by DIGITAL. Table 3–15 shows the meaning of each bit.

Table 3–15: Job Blocking Bits, Offset 36

Mnemonic	Bit	Meaning When Set
	0–3	Reserved.
USRWT$	4	The job is waiting for the USR.
	5	Reserved.
KSPND$	6	The job is suspended as a result of the monitor SUSPEND command.
	7	Reserved.
EXIT$	8	The job is waiting for all I/O to complete.
NORUN$	9	The job is not running (that is, it is a foreground or system job that has completed).
SPND$	10	The job is suspended.

(Continued on next page)

Table 3–15: Job Blocking Bits, Offset 36 (Cont.)

Mnemonic	Bit	Meaning When Set
CHNWT$	11	The job is waiting for I/O on a channel to complete.
TTOEM$	12	The job is waiting for the output ring buffer to be empty.
TTOWT$	13	The job is waiting for room in the output ring buffer.
TTIWT$	14	The job is waiting for terminal input.
	15	Reserved.

3.4.1 Queue Element Format Summary

This section summarizes the formats of the various types of queue elements. For detailed information on clock support and timer service, see Section 3.2, which also describes the timer queue element. Section 3.3 contains more information on the queued I/O system and includes descriptions of the I/O queue element, the completion queue element, and the synch queue element.

3.4.1.1 I/O Queue Element — Figure 3–24 shows the format of an I/O queue element.

Figure 3–24: I/O Queue Element Format

<table>
<tr><th>NAME</th><th>OFFSET</th><th colspan="4">CONTENTS</th></tr>
<tr><td>Q.LINK</td><td>0</td><td colspan="4">LINK TO NEXT QUEUE ELEMENT; 0 IF NONE</td></tr>
<tr><td>Q.CSW</td><td>2</td><td colspan="4">POINTER TO CHANNEL STATUS WORD IN I/O CHANNEL (SEE FIGURE 3-29)</td></tr>
<tr><td>Q.BLKN</td><td>4</td><td colspan="4">PHYSICAL BLOCK NUMBER</td></tr>
<tr><td>Q.FUNC
Q.UNIT
Q.JNUM</td><td>6
7
7</td><td>RESERVED

(1 BIT)</td><td>JOB NUMBER (4 BITS) 0 = BG</td><td>DEVICE UNIT (3 BITS)</td><td>SPECIAL FUNCTION CODE (8 BITS)</td></tr>
<tr><td>Q.BUFF</td><td>10</td><td colspan="4">USER BUFFER ADDRESS (MAPPED THROUGH PAR1 WITH Q.PAR VALUE, IF XM)</td></tr>
<tr><td>Q.WCNT</td><td>12</td><td colspan="4">WORD COUNT { IF <0, OPERATION IS WRITE; IF =0, OPERATION IS SEEK; IF >0, OPERATION IS READ }
THE TRUE WORD COUNT IS THE ABSOLUTE VALUE OF THIS WORD.</td></tr>
<tr><td>Q.COMP</td><td>14</td><td colspan="4">COMPLETION ROUTINE CODE { IF 0, THIS IS WAIT-MODE I/O; IF 1, JUST QUEUE THE REQUEST AND RETURN; IF EVEN, COMPLETION ROUTINE ADDRESS }</td></tr>
<tr><td>Q.PAR</td><td>16</td><td colspan="4">PAR1 VALUE (XM ONLY)</td></tr>
<tr><td></td><td></td><td colspan="4">RESERVED (XM ONLY)</td></tr>
<tr><td></td><td></td><td colspan="4">RESERVED (XM ONLY)</td></tr>
</table>

3.4.1.2 Completion Queue Element – Figure 3–25 shows the format of a completion queue element.

Figure 3–25: Completion Queue Element Format

NAME	OFFSET	CONTENTS
Q.LINK	0	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
	2	RESERVED
	4	RESERVED
	6	RESERVED
Q.BUFF	10	CHANNEL STATUS WORD
Q.WCNT	12	OFFSET FROM START OF CHANNEL AREA TO THIS CHANNEL
Q.COMP	14	COMPLETION ROUTINE ADDRESS
		THREE ADDITIONAL WORDS ARE PRESENT IN XM SYSTEMS. THEY ARE UNUSED, AND ARE RESERVED FOR FUTURE USE BY DIGITAL.

3.4.1.3 Synch Queue Element – Figure 3–26 shows the format of a synch queue element.

Figure 3–26: Synch Queue Element Format

NAME	OFFSET	CONTENTS
Q.LINK	0	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
Q.CSW	2	JOB NUMBER
Q.BLKN	4	RESERVED
Q.FUNC	6	RESERVED
Q.BUFF	10	SYNCH ID
Q.WCNT	12	–1 (CUE THAT THIS IS A SYNCH ELEMENT)
Q.COMP	14	SYNCH ROUTINE ADDRESS

3.4.1.4 Fork Queue Element – Figure 3–27 shows the format of a fork queue element.

Figure 3–27: Fork Queue Element Format

NAME	OFFSET	CONTENTS
F.BLNK	0	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
F.BADR	2	FORK ROUTINE ADDRESS
F.BR5	4	R5 SAVE AREA
F.BR4	6	R4 SAVE AREA

3.4.1.5 Timer Queue Element — Figure 3–28 shows the format of a timer queue element.

Figure 3–28: Timer Queue Element Format

NAME	OFFSET	CONTENTS
C.HOT	0	HIGH-ORDER TIME
C.LOT	2	LOW-ORDER TIME
C.LINK	4	LINK TO NEXT QUEUE ELEMENT; 0 IF NONE
C.JNUM	6	OWNER'S JOB NUMBER
C.SEQ	10	OWNER'S SEQUENCE NUMBER ID
C.SYS	12	–1 IF SYSTEM TIMER ELEMENT; –3 IF .TWAIT ELEMENT IN XM
C.COMP	14	ADDRESS OF COMPLETION ROUTINE
		THREE ADDITIONAL WORDS ARE PRESENT IN XM SYSTEMS. THEY ARE UNUSED, AND ARE RESERVED FOR FUTURE USE BY DIGITAL.

3.4.2 I/O Channel Format

Figure 3–29 shows the format of an I/O channel. Since each channel uses five words, the size of the monitor's channel area is five times the number of channels. RT–11 allocates 16 channels for each job. The channel area is 80 decimal words long. For SJ, a single channel area is located in RMON. For FB and XM, one channel area for each job is located in the job's impure area. The .CDFN programmed request can provide more channels. Table 3–16 shows the significant bits in the Channel Status Word.

Figure 3–29: I/O Channel Description

NAME	OFFSET	CONTENTS	
	0	CHANNEL STATUS WORD	
C.SBLK	2	STARTING BLOCK NUMBER OF THIS FILE (0 IF NON-FILE-STRUCTURED)	
C.LENG	4	LENGTH OF FILE (IF OPENED BY .LOOKUP) SIZE OF EMPTY AREA (IF OPENED BY .ENTER)	
C.USED	6	HIGHEST BLOCK WRITTEN	
C.DEVQ	10	DEVICE UNIT NUMBER	NUMBER OF REQUESTS PENDING ON THIS CHANNEL

Table 3–16: Channel Status Word (CSW)

Bit	Meaning
0	Hard error bit. 0 = No error. 1 = Hard error.
1–5	Index into the $PNAME table and other device tables.
6	RENAME flag. 0 = No RENAME is in progress. 1 = A RENAME operation is in progress.
7	0 = The file was opened with a .LOOKUP. The monitor does not modify the directory when the file is closed. 1 = The file was opened with an .ENTER. The monitor modifies the directory when the file is closed.
8–12	The number of the directory segment containing this entry.
13	End-of-file (EOF) bit. 0 = No end-of-file. 1 = End-of-file was found on this channel.
14	Reserved.
15	0 = The channel is free. 1 = The channel is active.

3.4.3 Device Tables

Tables in the Resident Monitor keep track of the devices on the RT–11 system. These tables are contained in the module SYSTBL.MAC, which is created by system generation and assembled separately from the module RMON. SYSTBL is linked with RMON and other modules to form the Resident Monitor. The symbol $SLOT in SYSTBL, which is defined at system generation time, defines the maximum number of devices the system can have. The value of $SLOT is greater than or equal to 3, and less than or equal to 31 decimal.

3.4.3.1 $PNAME Table — The permanent name table is called $PNAME. It is the central table around which all the others are constructed. The total number of entries is fixed at assembly time; you can allocate extra slots then. Entries are made in $PNAME at monitor assembly time for each device that is built into the system.

Each table entry consists of a single word that contains the Radix–50 code for the two-character physical device name. (For example, the entry for DECtape is .RAD50 /DT/.) The TT device must be first in the table; the system device is always second. After that, the position of a device in this table is not critical. Once the entries are made into this table, their relative position (that is, their order in the table) determines the general device index used in various places in the monitor. Thus, the other tables are organized in the same order as $PNAME. The offset of a device name entry in $PNAME serves as the index into the other tables for a given device.

The bootstrap checks the system generation parameters of a handler with those of the current monitor (by inspecting the low three bits of SYSGEN at RMON fixed offset 372), and zeroes the $PNAME entry for that device if the parameters do not match. The INSTALL monitor command cannot install a handler whose conditional parameters do not match those of the monitor.

3.4.3.2 $STAT Table – The device status table is called $STAT. Entries to this table are made at assembly time for those devices that are permanently resident in the RT–11 system, such as TT and MQ in FB and XM systems. When the system is bootstrapped, the entries for all other devices are filled in when the handler is installed by the bootstrap or the INSTALL monitor command. Each device in the system has a status entry in its corresponding slot in $STAT. The device status word identifies each physical device and provides information about it, such as whether it is random or sequential access. The device status word is part of the information returned to a running program by the .DSTATUS programmed request. See Chapter 7 for details on the status word.

3.4.3.3 $DVREC Table – The device handler block number table is called $DVREC. Entries to this table are made at bootstrap time for devices that are built into the system, and at INSTALL time for additional devices. The entries are the absolute block numbers where each of the device handlers resides on the system device. Since handlers are treated as files, their positions on the system device are not necessarily fixed. Thus, each time the system is bootstrapped, the handlers are located and $DVREC is updated with their locations on the system device. The pointer in $DVREC points to block 1 of the file. (Because handlers are linked at 1000, the actual handler code starts in the second block of the file.) A zero entry in the $DVREC table indicates that no handler for the device in that slot was necessary (such as TT or MQ in FB and XM systems). (Note that if block 0 of the handler file resides on a bad block on the system device, RT–11 cannot install or fetch the handler.) Note also that 0 is a valid $DVREC entry for permanently resident devices.

3.4.3.4 $ENTRY Table – The handler entry point table is called $ENTRY. Entries in this table are made whenever a handler is loaded into memory by either the .FETCH programmed request or by the LOAD keyboard monitor command. The entry for each device is a pointer to the fourth word of the device handler in memory. The entry is zeroed when the handler is removed by the .RELEASE programmed request or by the UNLOAD keyboard monitor command.

Some device handlers are permanently resident. These include the system device handler and, for FB and XM systems, the TT handler. The $ENTRY values for such devices are fixed at boot time.

3.4.3.5 $DVSIZ Table — Each entry in the $DVSIZ table contains the size of a device, in blocks. The value is 0 for a non-file-structured device. For devices that accept multi-size volumes, the entry contains the size of the smallest possible volume.

3.4.3.6 $HSIZE Table — Each entry in the $HSIZE table contains the size of a device handler, in bytes. This value indicates the amount of memory needed to load each handler.

3.4.3.7 $UNAM1 and $UNAM2 Tables — The tables that keep track of logical device names and the physical names that are assigned to them are called $UNAM1 and $UNAM2. Entries are made in these tables when the ASSIGN monitor command is issued. The physical device name is stored in $UNAM1 and the logical name associated with it is stored in the corresponding slot in $UNAM2. When the system is first bootstrapped, there are two assignments already in effect that associate the logical names DK and SY with the device from which the system was booted. The value of $SLOT, which is determined at system generation time, limits the total number of logical name assignments. Thus, you can issue one ASSIGN command for each device in your system. (The initial SY and DK assignments at bootstrap time do not come out of your total.)

The $UNAM1 and $UNAM2 tables are not indexed by the $PNAME table offset. The fact that the tables are the same size is interesting, but not significant.

3.4.3.8 $OWNER Table — The device ownership table is called $OWNER and it is used in the FB and XM environments to arbitrate device ownership. The table is ($SLOT*2) words in length and is divided into two-word entries for each device. Entries are made into this table when the LOAD keyboard monitor command is issued. Each two-word entry is in turn divided into eight four-bit fields capable of holding a job number. The low four bits of the first byte correspond to unit 0, and the high four bits correspond to unit 1. The low four bits of the next byte correspond to unit 2, and so on (see Figure 3–30). Thus, each device is presumed to have up to eight units, each assigned independently of the others. However, if the device is non-file-structured, units are not assigned independently; the monitor ASSIGN code ensures that ownership of all units is assigned to one job.

Figure 3–30: $OWNER Entry

DEVICE UNIT #	3	2	1	0
	OWNER #	OWNER #	OWNER #	OWNER #
	OWNER #	OWNER #	OWNER #	OWNER #
DEVICE UNIT #	7	6	5	4

When a background job, a foreground job, or a system job attempts to access a particular unit of a device, the monitor checks to be sure the unit being accessed is either public or belongs to the requesting job. If another job owns the unit, a fatal error is generated.

The device is public if the four-bit field is 0. If the device is not public, the field contains a code equal to the job number plus 1. Since job numbers are always even, the ownership code is odd. For example, in a distributed foreground/background system, the owner field value for the background job is 1; for the foreground job it is 3. In a foreground/background system with the system job feature the owner field value for the background job is still 1; for the foreground job it is 17. The owner field value for a system job is 1 plus the job number.

3.4.3.9 Adding a Device to the Tables — You can create free slots in the tables by deleting or renaming one or more of the device handler files from the system device and rebooting the system, or by issuing the REMOVE monitor command. The INSTALL monitor command can install a different device handler into the table after the system has been booted. However, INSTALL does not make a device entry permanent. For more information on installation, the DEV macro, and the bootstrap, see Chapter 7.

Chapter 4
Extended Memory Feature

After introducing RT–11's extended memory feature, this chapter provides an overview of the hardware components that are the basis of the extended memory system. (The term **extended memory** refers to physical memory above the 28K word boundary that can be accessed only by using special hardware. **Low memory** is the physical memory between 0 and 28K words. In some systems with an additional 2K words of low memory, low memory extends to 30K words and there is no extended memory.) It then shows how RT–11 implements support for extended memory, and explains how to design, code, and execute a program in an extended memory environment. Following these demonstrations is a discussion of the implications of extended memory support for other system software components and a description of all the restrictions you must observe when working with extended memory. Lastly, this chapter describes how to debug an extended memory application program and provides a sample program that uses double buffering in extended memory.

4.1 Introduction

The following sections present a brief overview of the circumstances that led to the RT–11 extended memory implementation. Read it to gain an understanding of the limitations of 28K-word systems and the means by which RT–11 circumvents these limitations.

4.1.1 16-Bit Addressing

Each computer in the PDP–11 family can directly address 32K words. A PDP–11 computer can never address more than this amount of memory directly because its architecture provides only 16-bit addresses. Figure 4–1 illustrates this addressing limitation. Since the PDP–11 computer can address bytes individually, you can see from the illustration why its address space is limited to 32K words.

Remember that one K equals 1024 decimal, or 2 raised to the 10th power. The *RT–11 Mini-Reference Manual* provides a convenient reference chart of K-words and their equivalent octal numbers.

Figure 4–1: 16-Bit Word Addressing Space Limitation

A 16-BIT WORD WITH THE HIGHEST POSSIBLE VALUE, EXPRESSED IN BINARY:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

THE SAME VALUE EXPRESSED IN OCTAL IS 177777.

THE SAME VALUE EXPRESSED IN DECIMAL IS 65535.

SINCE 0 IS A VALID LOCATION, THE PDP-11 CAN ADDRESS 65536 UNIQUE BYTE LOCATIONS. THUS, THE PDP-11 (WHICH IS A BYTE-ADDRESSABLE COMPUTER) ADDRESSES 64K BYTES OF MEMORY, OR 32K WORDS OF MEMORY.

In unmapped PDP–11 systems (those not using extended memory), the highest 4K words of address space, called the I/O page, are reserved for device registers, general registers, and so on. Thus, only 28K words of address space are left for use by the operating system software and programs. On a system with 28K words of memory, all 28K words are available.

4.1.2 Virtual and Physical Addresses in a 28K-Word System

A **virtual address** is a value in the range 0 through 177777. It is a 16-bit address within a program's 32K-word address space.

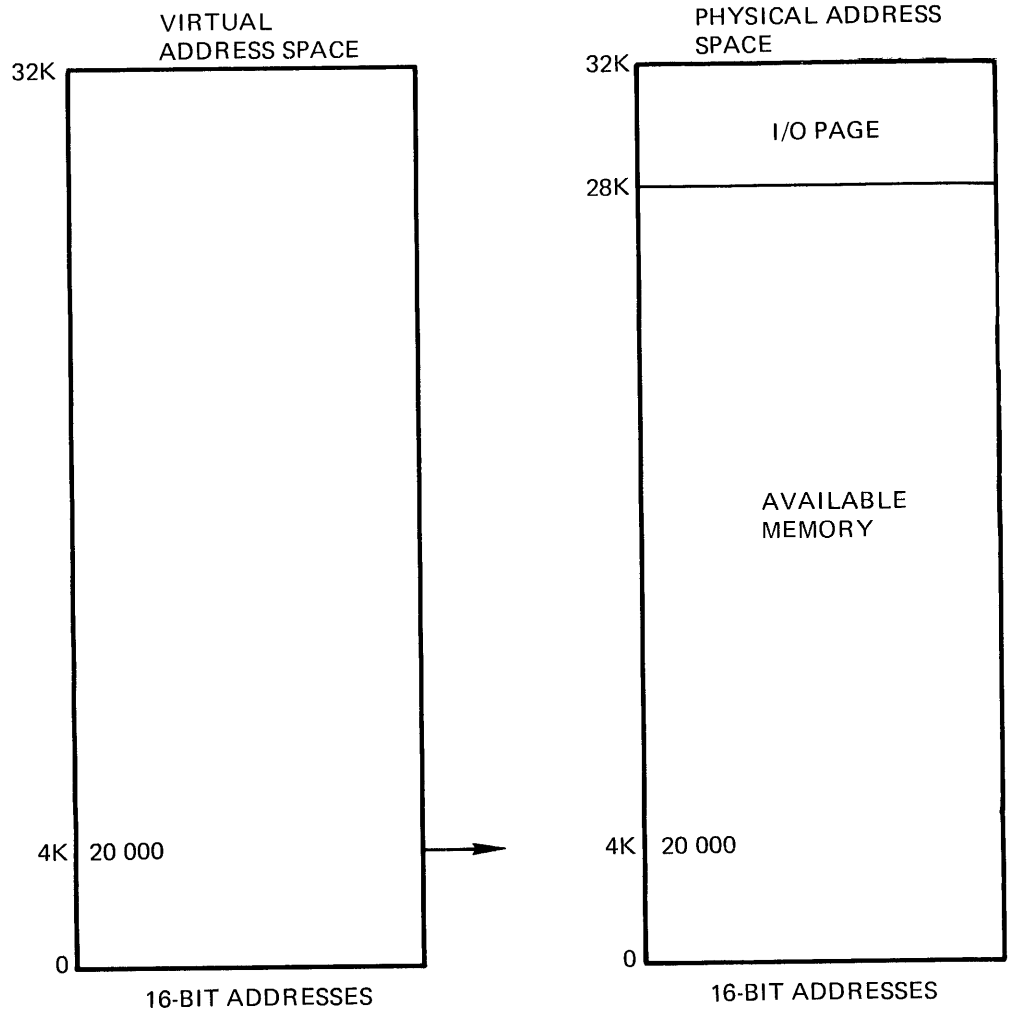
A **physical address** is the actual hardware address of a specific memory location. Physical addresses are not limited to 16 bits.

Figure 4–2 shows the relationship between virtual address space and physical address space in an RT–11 system with 28K words of memory. Note that in this system, which could be running either the SJ or FB monitor, there is a one-to-one correspondence between virtual and physical addresses. For example, virtual address 20000 corresponds directly to physical address 020000.

4.1.3 Circumventing the 28K-Word Memory Limitation

Before RT–11 provided support for extended memory, systems were limited to using 28K words of memory. Programmers have traditionally used two mechanisms to circumvent the 28K-word available memory limitation. One of the mechanisms is called **chaining**: one program calls a second program at exit time; the second program provides additional processing for the data the original program passes to it. The MACRO–11 assembler, for example, assembles a MACRO–11 source file and chains to CREF, which produces the cross-reference listing. One way, then, to run a program that is larger than the amount of memory available is to divide the program into two or more

Figure 4–2: Virtual and Physical Addresses in a 28K-Word System

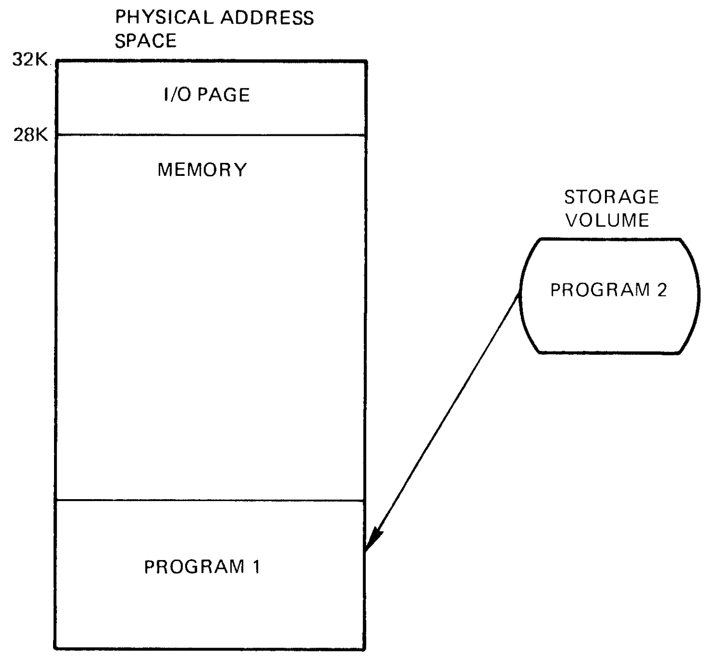


functionally distinct parts. Then, when the first program finishes, it can start up the second program by chaining to it.

Another way to run a program that is larger than the amount of memory available is to divide the program into overlay segments. Separate segments can then take turns residing in the same place in physical memory. By using overlays you can run a very large program in a much smaller amount of physical memory.

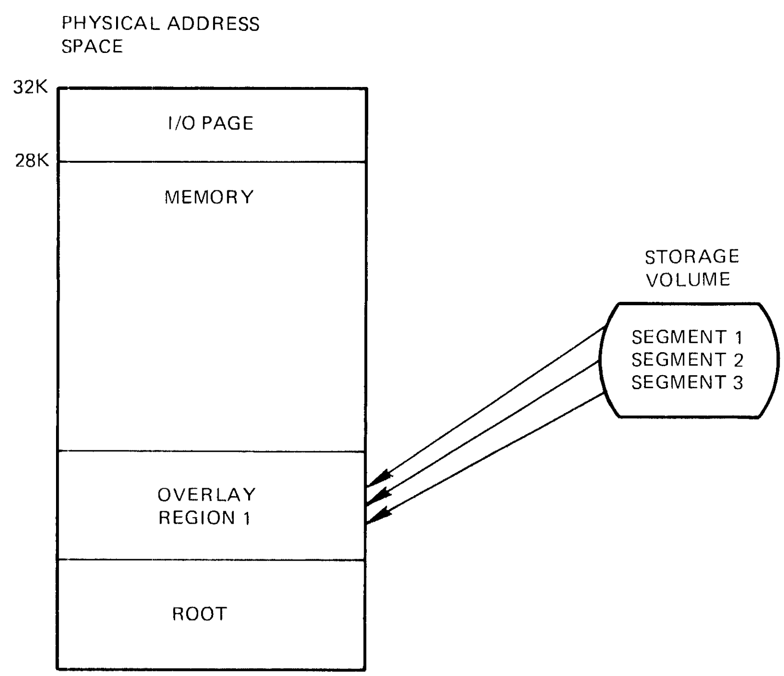
In both chaining and overlaying, instructions and data in the separate programs or segments use both the **same virtual addresses** and the **same locations** in physical memory. Programs or segments not currently in memory reside on an auxiliary storage volume. Figure 4–3 illustrates chaining; Figure 4–4 shows overlaying.

Figure 4–3: Chaining



AS PROGRAM 1 EXITS, IT CALLS PROGRAM 2. PROGRAM 2 USES THE SAME VIRTUAL ADDRESSES AND PHYSICAL MEMORY LOCATIONS AS PROGRAM 1.

Figure 4–4: Overlaying



AS THE PROGRAM RUNS, SEGMENTS 1, 2, AND 3 TAKE TURNS RESIDING IN OVERLAY REGION 1. THE SEGMENTS ALL USE THE SAME VIRTUAL ADDRESSES AND PHYSICAL MEMORY LOCATIONS.

4.1.4 18- and 22-Bit Addressing

Although PDP–11 software uses 16-bit words, it is possible to access more than 32K words of memory by using special memory management hardware. With memory management, RT–11 can use up to 18-bit addresses on a Unibus machine, or up to 22-bit addresses on a Q-bus machine. This means that you can address up to 124K words plus a 4K-word I/O page on a Unibus machine, or up to 2044K words plus a 4K-word I/O page on a Q-bus machine. Figure 4–5 shows the addressing range for 18- and 22-bit addresses.

Figure 4–5: 18- and 22-Bit Word Addressing Range

AN 18-BIT WORD WITH THE HIGHEST POSSIBLE VALUE, EXPRESSED IN BINARY:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

THE SAME VALUE EXPRESSED IN OCTAL IS 777777.

THE SAME VALUE EXPRESSED IN DECIMAL IS 262143.

A 22-BIT WORD WITH HIGHEST POSSIBLE VALUE, EXPRESSED IN BINARY:

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

THE SAME VALUE EXPRESSED IN OCTAL IS 17777777

THE SAME VALUE EXPRESSED IN DECIMAL IS 2097151

SINCE 0 IS A VALID LOCATION, 18 BITS CAN ADDRESS 262,144 UNIQUE BYTE LOCATIONS, OR 128K WORDS OF PHYSICAL ADDRESS SPACE. 22 BITS CAN ADDRESS 2,097,151 UNIQUE BYTE LOCATIONS, OR 1024K WORDS OF PHYSICAL ADDRESS SPACE.

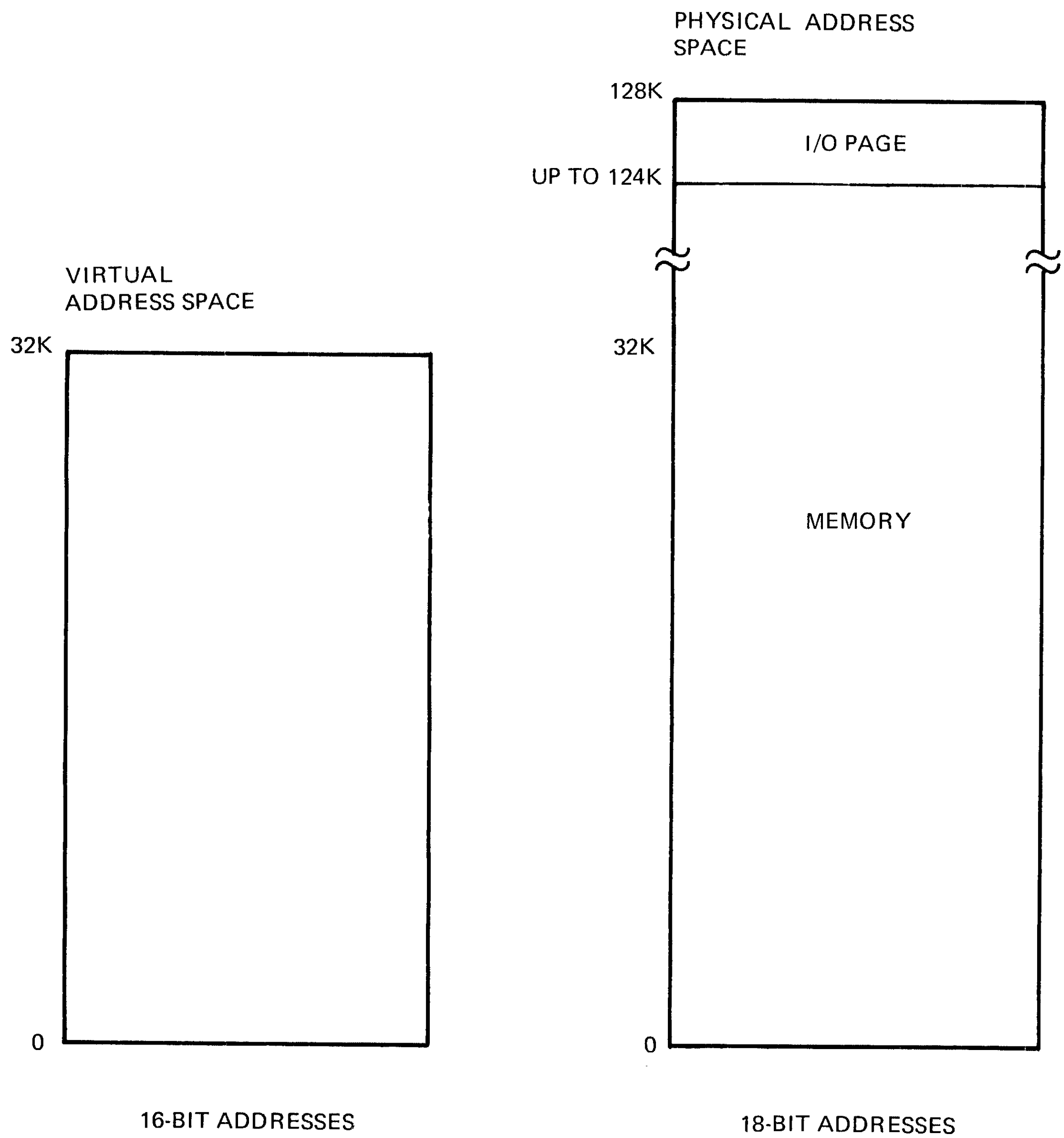
4.1.5 Virtual and Physical Addresses with Extended Memory Hardware

The virtual addresses your program uses are always limited to 16 bits so that your program's virtual address space is always limited to 32K words.

However, an 18-bit address can reference any location between 0 and 128K words; a 22-bit address can reference any location between 0 and 2048K words. RT–11 systems with more than 28K words of memory, physical locations are referenced by the hardware as 18- or 22-bit addresses.

As Figure 4–6 shows, there can no longer be a direct one-to-one correspondence between virtual and physical addresses.

Figure 4–6: Virtual and Physical Addresses with Extended Memory Hardware



4.1.6 Circumventing the 32K-Word Address Limitation

As memory technology improves, it becomes more and more feasible to provide PDP–11 systems with more than 28K words of memory. Since the UNIBUS and Q-bus already have the ability to use addresses longer than 16 bits, it remains the task of the hardware — the Memory Management Unit — and the operating system software to set up a correspondence between a program's virtual addresses and physical memory locations so that programs can access all of memory.

If you select extended memory as a special feature at system generation time, you can take advantage of the 18- or 22-bit addresses. The extended memory feature permits programs, which are still restricted to using 16-bit words, to access 2044K words of physical memory. RT–11 implements support for extended memory through a combination of hardware and software components.

Through its extended memory (XM) monitor, RT–11 provides a mechanism to associate a virtual address with a physical address. This process is called **mapping**. RT–11 permits programs to access extended memory by mapping their virtual addresses to physical locations in memory. In summary:

- Every location in memory has an 18- or 22-bit physical address; there are more physical addresses than virtual addresses.
- A program cannot access specific physical addresses unless its virtual addresses are mapped to those physical locations.
- Programs can access all the available physical memory by using their virtual addresses over and over again, but with different mapping each time.

Section 4.3 presents more material on mapping. Be sure you understand the hardware concepts discussed in the next section before you proceed to 4.3.

In an extended memory system, programs are no longer limited to using 28K words of memory. However, they must still deal with the 32K-word addressing limitation. Typically, large programs are still divided into smaller segments, as in the 28K-word systems. While the instructions and data in separate segments of a program share the same virtual addresses, they can have unique physical addresses. Figure 4–7 shows a program that is divided into three overlay segments. The three segments are resident simultaneously in extended memory, but they share the virtual addresses in overlay region 1.

4.2 Hardware Concepts

There are three hardware requirements for an RT–11 extended memory system:

- At least 32K words of memory
- The Extended Instruction Set (EIS) option
- A Memory Management Unit

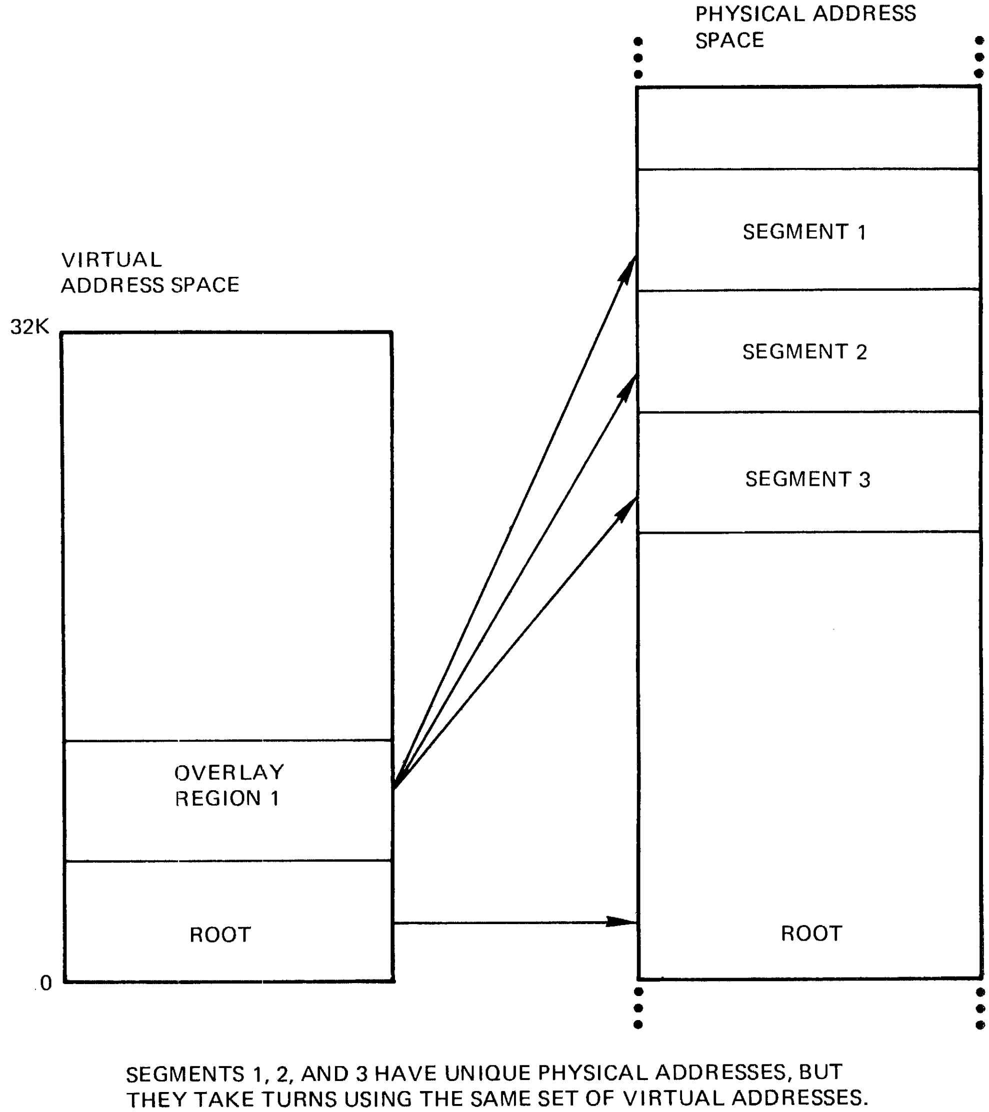
This manual provides an overview of the memory management hardware and its functions. The best sources of detailed information on the memory management hardware are the hardware manuals for the KT11-C, -CD, and -D Memory Management Units. Their full titles and order numbers are:

KT11-C, CD Memory Management Unit User's Manual: EK-KT11C-OP–001

KT11-D Memory Management Option Manual: EK-KT11D-TM–002

KT11-D Memory Management Option User's Manual: EK-KT11D-OP–001

Figure 4–7: Program Segments Sharing Virtual Address Space



Two sources of information on the memory management hardware are Chapter 10 of the *Microcomputers and Memories Handbook* (order number EB–20912–20) and the *PDP–11 Processor Handbook*.

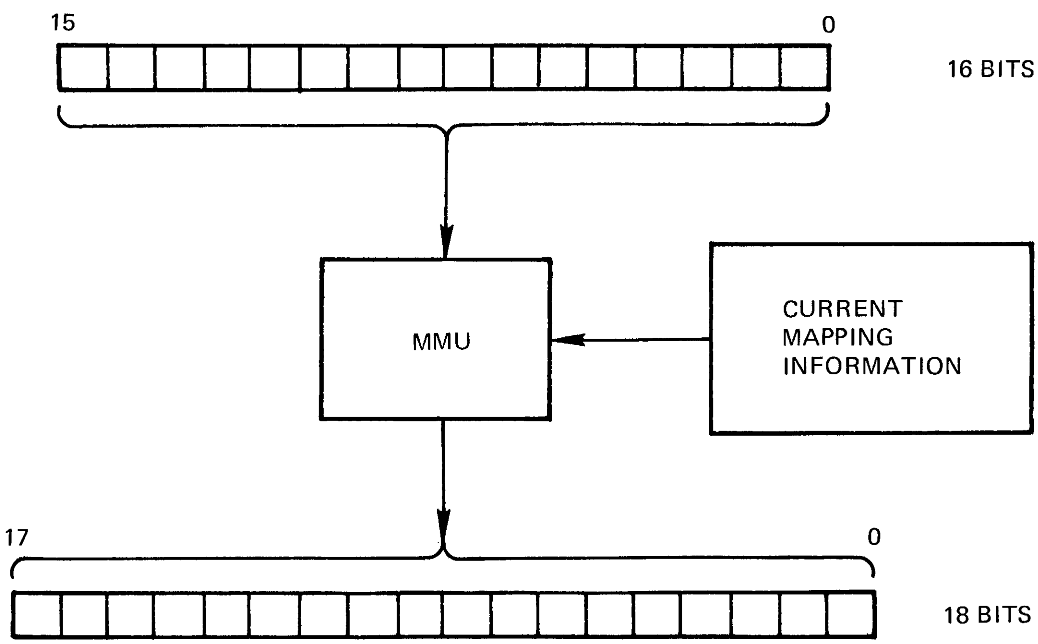
Note that it is not necessary to learn the details of how the Memory Management Units function in order to understand and use the RT–11 extended memory system. These manual references are provided for your convenience should you choose to do some further background reading.

4.2.1 Memory Management Unit

The central component of an XM system is a hardware option referred to generally as the Memory Management Unit, or MMU. DIGITAL manufactures several types of Memory Management Units, including the KT–11A, the KT11-C, the KT11-D, and the KT11-CD. RT–11 supports the minimal set of functions common to all the memory management units.

The function of the Memory Management Unit is to intercept a 16-bit virtual address generated by the processor and convert it to an 18- or 22-bit physical address. Figure 4–8 illustrates this process for 18 bits.

Figure 4–8: MMU Address Conversion



4.2.2 Concept of Pages

In an extended memory system the 32K-word virtual address space is divided into eight sections called **pages**. Each page begins on a 4K word boundary, and the pages are numbered from 0 through 7. A page is made up of units of 32 decimal words each. Since there can be as many as 128 of these units, a page can vary in size from 0 words to 4096 words, in 32-word increments. Figure 4–9 shows the virtual address space divided into eight 4K-word pages.

Figure 4–10 shows the virtual address space divided into five pages of varying lengths. The shaded areas in the virtual address space are not part of the pages, and are therefore inaccessible. Thus, short pages cause gaps in the virtual address space.

4.2.3 Relocation

When the Memory Management Unit converts a 16-bit virtual address to an 18- or 22-bit physical address, it *relocates* the virtual address. This means that two or more programs can have the same virtual addresses but different

Figure 4–9: 4K-Word Pages

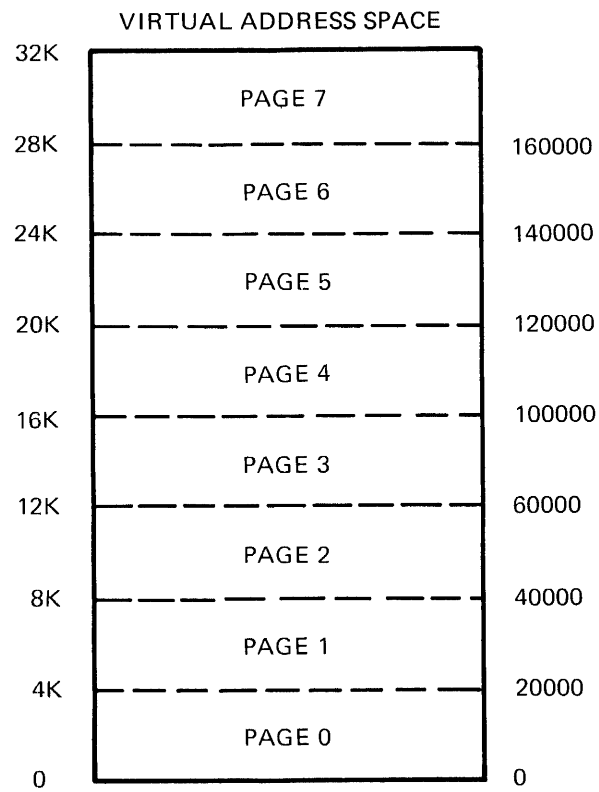
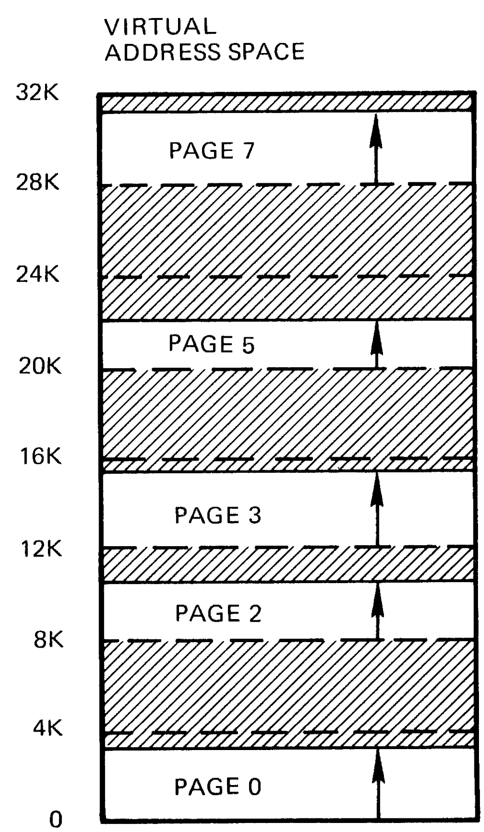
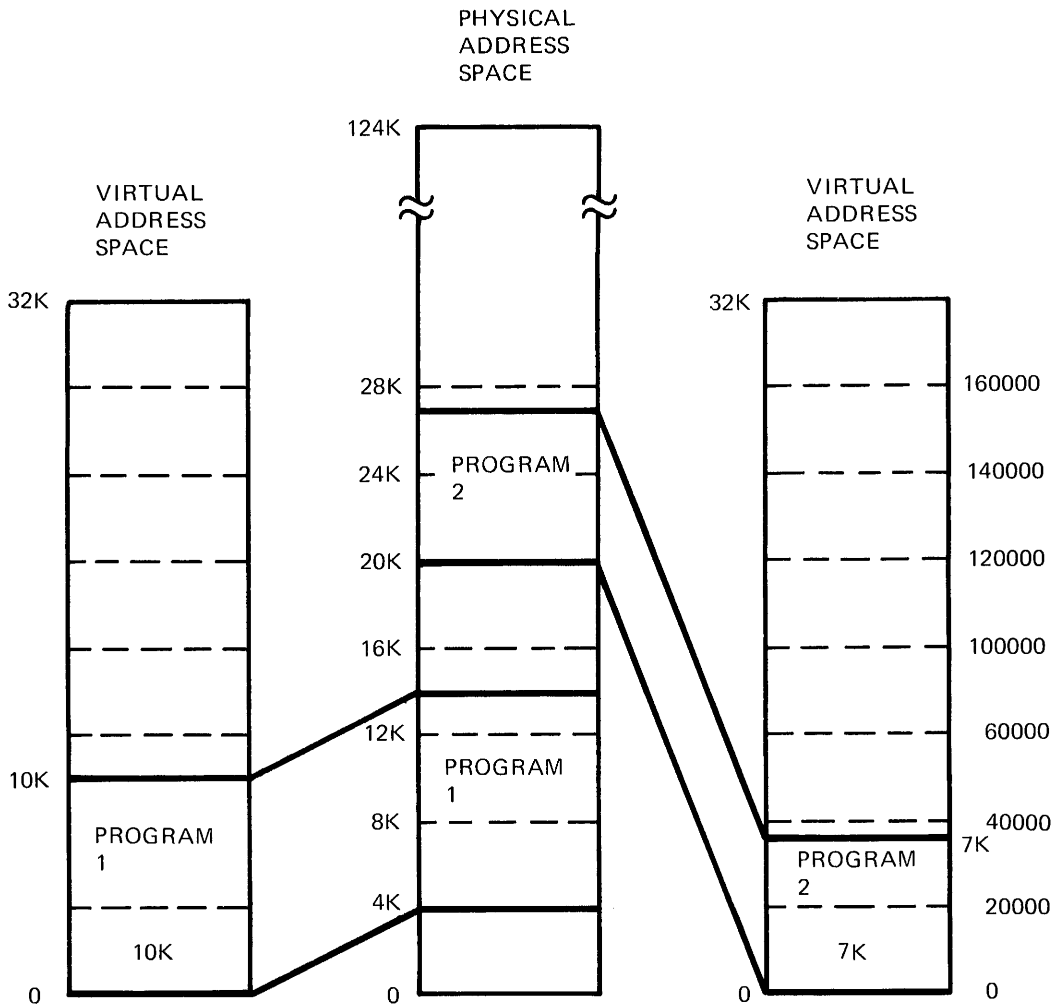


Figure 4–10: Smaller Pages



physical addresses. The Memory Management Unit relocates virtual addresses in units of pages. It assigns a page to a section of physical memory that starts on a 32-word decimal boundary. Figure 4–11 shows how the Memory Management Unit can relocate the virtual addresses of two different programs in a 124K-word memory.

Figure 4–11: Relocation by Program



Program 1 in Figure 4–11 is relocated by 20000 octal. So, when program 1 references virtual address 0, for example, it actually accesses memory location 20000.

Since the Memory Management Unit relocates each page of virtual address space separately, a program can reside in disjoint sections of memory, as Figure 4–12 shows.

4.2.4 Active Page Register (APR)

The RT–11 monitor communicates with the Memory Management Unit through the Active Page Registers, which are located in the I/O page. Each Active Page Register consists of two 16-bit words, as Figure 4–13 shows: a Page Address Register (PAR), and a Page Descriptor Register (PDR).

Figure 4–12: Relocation by Page

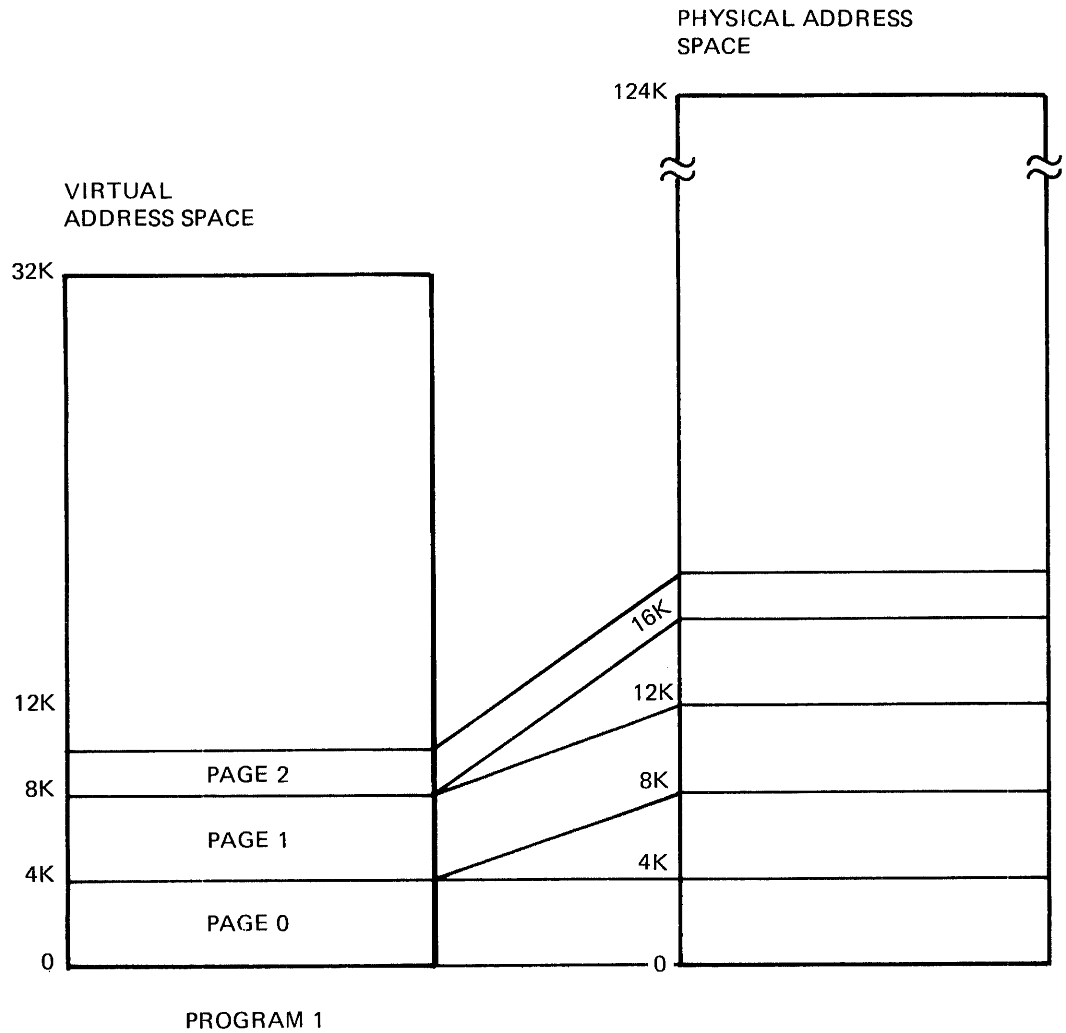
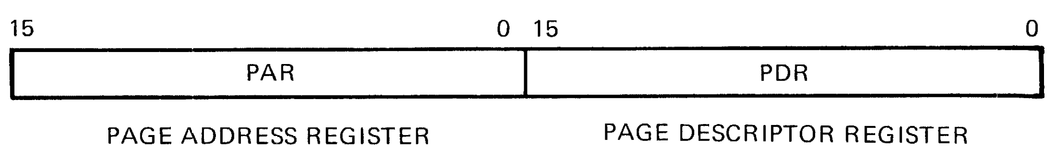


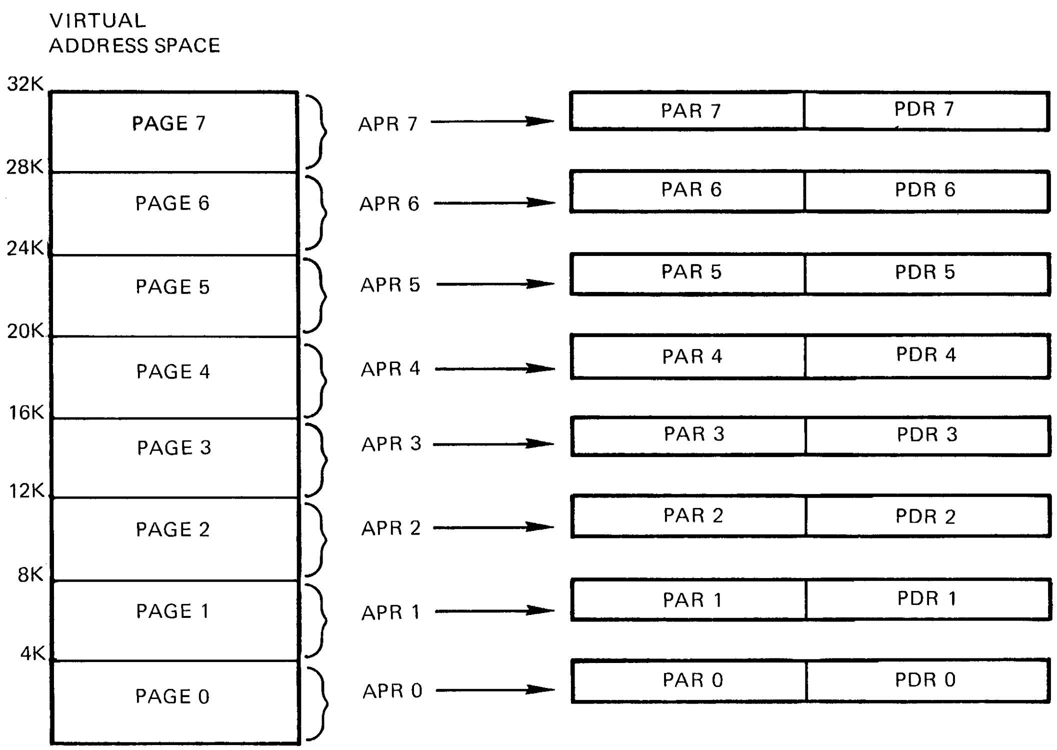
Figure 4–13: Active Page Register (APR)



The Page Address Register and the Page Descriptor Register always act as a pair. A set of eight Active Page Registers contains all the information necessary to describe and relocate the eight virtual address pages. The Page Descriptor Register describes how much of a virtual page to map to memory. The Page Address Register describes where in memory to put the virtual page.

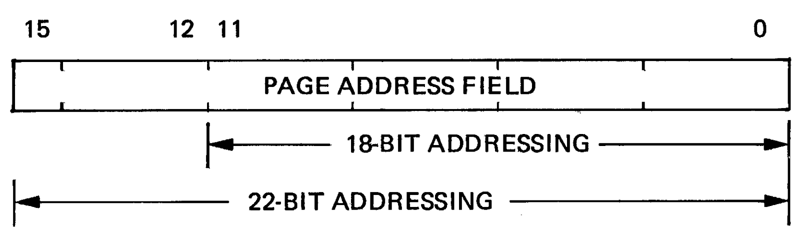
The eight Active Page Registers are numbered from 0 through 7. There is one Active Page Register for each page in the 32K-word virtual address space, as Figure 4–14 shows.

Figure 4–14: Correspondence Between Pages and Active Page Registers



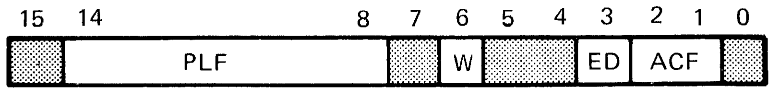
4.2.4.1 Page Address Register (PAR) — The eight Page Address Registers correspond directly to the eight virtual address pages. The Page Address Register contains the physical memory address in 32-word decimal units, or Page Address Field, for a particular virtual address page. Figure 4–15 shows the contents of the Page Address Register. Bits 0 through 11 are used for 18-bit addressing; bits 0 through 15 are used for 22-bit addressing.

Figure 4–15: Page Address Register (PAR)



4.2.4.2 Page Descriptor Register (PDR) — The Page Descriptor Register contains information about page expansion, page length, and access control for a particular page. Like the Page Address Registers, the Page Descriptor Registers correspond directly to the virtual address pages, as Figure 4–14 shows. Figure 4–16 shows the contents of the Page Descriptor Register. Unused bits are reserved for future use by DIGITAL.

Figure 4–16: Page Descriptor Register (PDR)



In Figure 4–16, the field marked **ACF** represents the **Access Control** field. This field describes how a particular page can be accessed, and whether or not a particular access should cause an abort of the current operation. The values in this field are as follows:

Value	Meaning
00	Nonresident page. Abort any attempt to access it.
01	Resident read-only page. Abort any attempt to write into it. (RT–11 does not use this value.)
10	Unused code. Abort all attempts to access this page. (RT–11 does not use this value.)
11	Resident read/write page. All accesses are valid.

The field marked **ED** is the **Expansion Direction** field. This bit indicates the direction in which a page can expand. The codes and their meanings are as follows:

Value	Meaning
0	The page expands to higher addresses. (In RT–11, this field is always 0.)
1	The page expands to lower addresses. (RT–11 does not use this value.)

The field marked **W** is the **Written Into** field. It indicates whether the page has been modified since it was loaded into memory. (RT–11 does not use this field.)

Some PDP–11 processors, instead of using bit 6 to indicate the page's modification status, use one or more of the reserved bits in the Page Descriptor Register. RT–11 ignores these other bits.

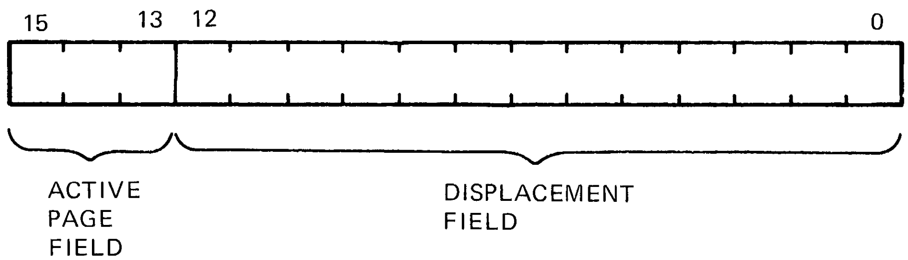
The field marked **PLF** is the **Page Length field**. It indicates the length of a page, in 32-word decimal units.

4.2.5 Converting a 16-Bit Address to an 18- or 22-Bit Address

The information necessary for the Memory Management Unit to convert a 16-bit virtual address to an 18- or 22-bit physical address is contained in the virtual address and in its corresponding Active Page Register set. Figure 4–17 shows the meanings of the fields in the virtual address. These fields represent a breakdown of the virtual address that is convenient for RT–11 and the MMU to use.

Bits 13 through 15 of the virtual address constitute the *Active Page Field.* This field determines which Active Page Register the Memory Management Unit will use to create the physical address.

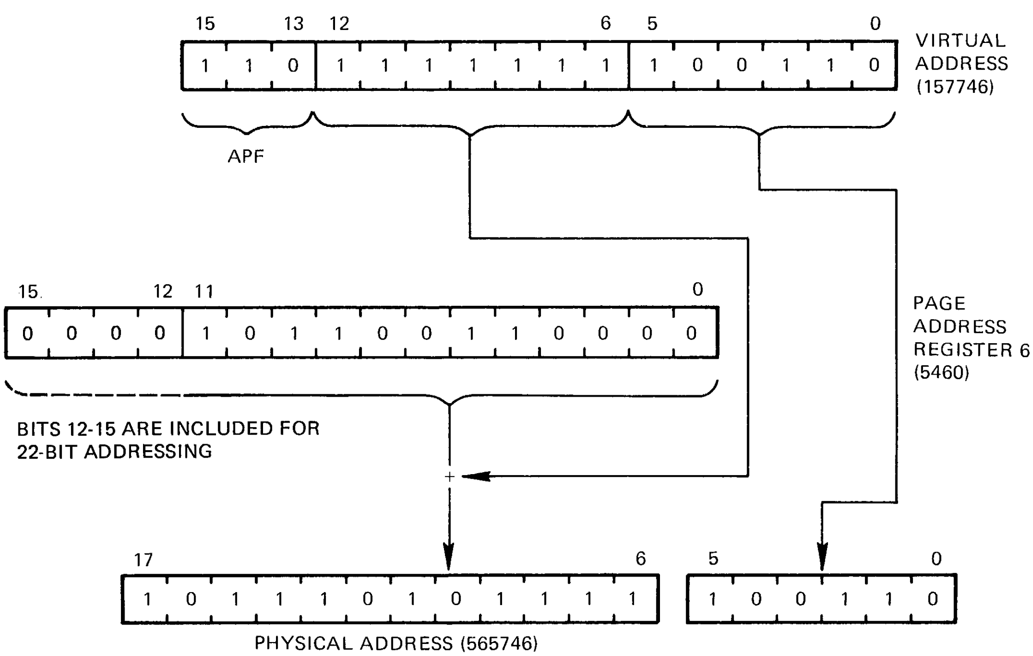
Figure 4–17: Virtual Address



Bits 0 through 12 of the virtual address are the *Displacement Field*, which contains an address relative to the beginning of a page.

The rest of the information necessary to create a physical address is contained in the Page Address field of the appropriate Page Address Register. Figure 4–18 shows how the Memory Management Unit converts a 16-bit virtual address to an 18- or 22-bit physical address. In this example, Page Address Register 6 contains 5460 octal, so virtual address 157746 converts to physical address 565746. Bits 12–15 of the Page Address Register are included for 22-bit addressing.

Figure 4–18: MMU Address Conversion (Detail)



As you can see from Figure 4–18, bits 13, 14, and 15 of the virtual address specify which Active Page Register to use. The Memory Management Unit adds the value in bits 6 through 12 of the virtual address to the corresponding Page Address Register. The Memory Management Unit places the result of this addition in bits 6 through 17 or 6 through 21 of the physical address.

The Memory Management Unit copies the value in bits 0 through 5 of the virtual address into bits 0 through 5 of the physical address to form the final 18- or 22-bit physical address.

4.2.6 Status Registers

The Memory Management Unit also communicates with the RT–11 monitor through two status registers. Status Register 0, located at 777572 in the I/O page contains abort error flags, the memory management enable bit, and other essential information required by RT–11 to recover from an abort or to service a memory management trap. Status Register 2, located at 777576, is a read-only register containing the 16-bit virtual address that the Memory Management Unit is currently converting to an 18- or 22-bit physical address. (RT–11 does not use Status Register 2. However, if a memory management unit fault occurs in your system, you can examine this register yourself.) RT–11 also uses Memory Management Register 3 (MMSR3), located at 772516, to enable 22-bit addressing.

4.2.7 Kernel and User Processor Modes

In addition to its primary function of managing the address space, the memory management system must provide some kind of protection for the monitor. To implement protection, the processor provides two modes of operation: **kernel mode** and **user mode**. The two modes provide a mechanism for separating system-level functions (kernel mode) from application-level functions (user mode).

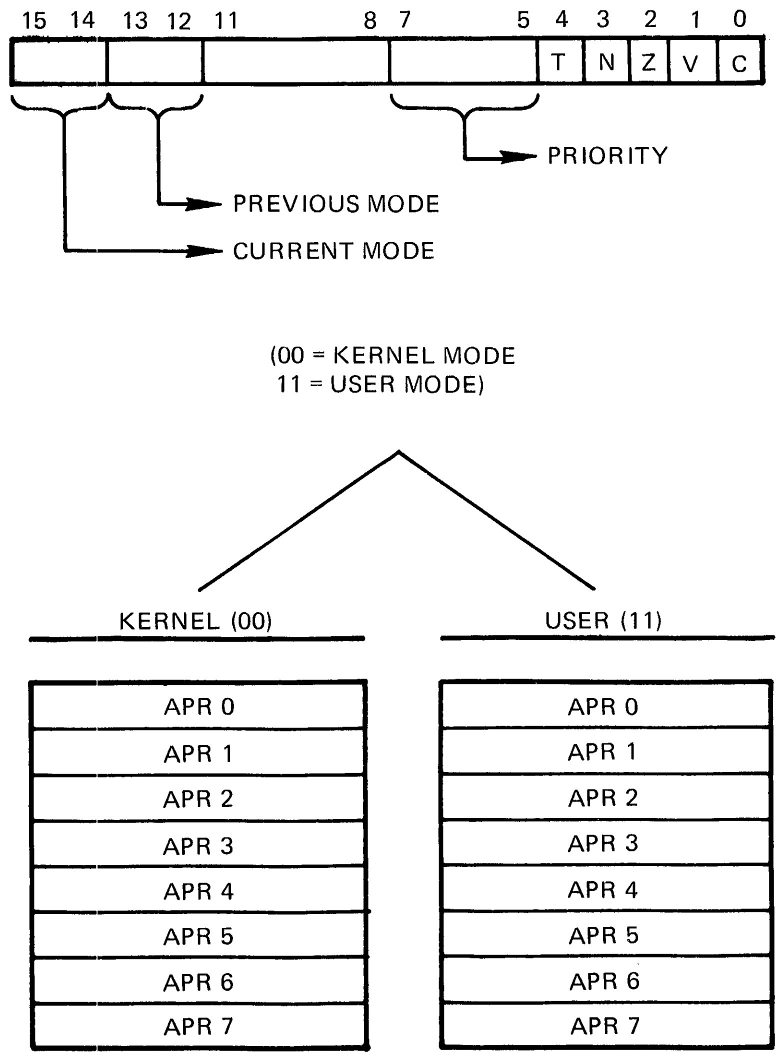
Each mode has its own set of eight Active Page Registers and its own stack pointer. Therefore, each processor mode also makes its own assignments of virtual addresses to physical locations: each mode has its own mapping. Figure 4–19 shows how the value in bits 14 and 15 of the Processor Status word determine in which processor mode execution takes place.

Routines that run in **kernel mode** are generally part of the run-time operating system software and must not be corrupted by other programs. RT–11 uses the processor's kernel mode for the Resident Monitor and the USR, for interrupt service routines, and for device handlers, including .SYNCH and .FORK routines. Interrupts and traps vector through kernel mapping and cause execution to continue in kernel mode.

Routines that run in **user mode** are generally part of application programs. They are prevented from executing instructions that could corrupt the monitor or halt the computer. For example, a RESET instruction acts as a NOP instruction in user mode, and a HALT instruction generates a trap to 10. RT–11 uses the processor's user mode for the Keyboard Monitor, for system utility programs, and for application programs and their completion routines.

Since each processor mode uses its own set of Active Page Registers, kernel mapping is not necessarily identical to user mapping. For example, if user virtual address 20010 is associated with physical address 40210, it does not

Figure 4–19: Processor Status Word and Active Page Registers

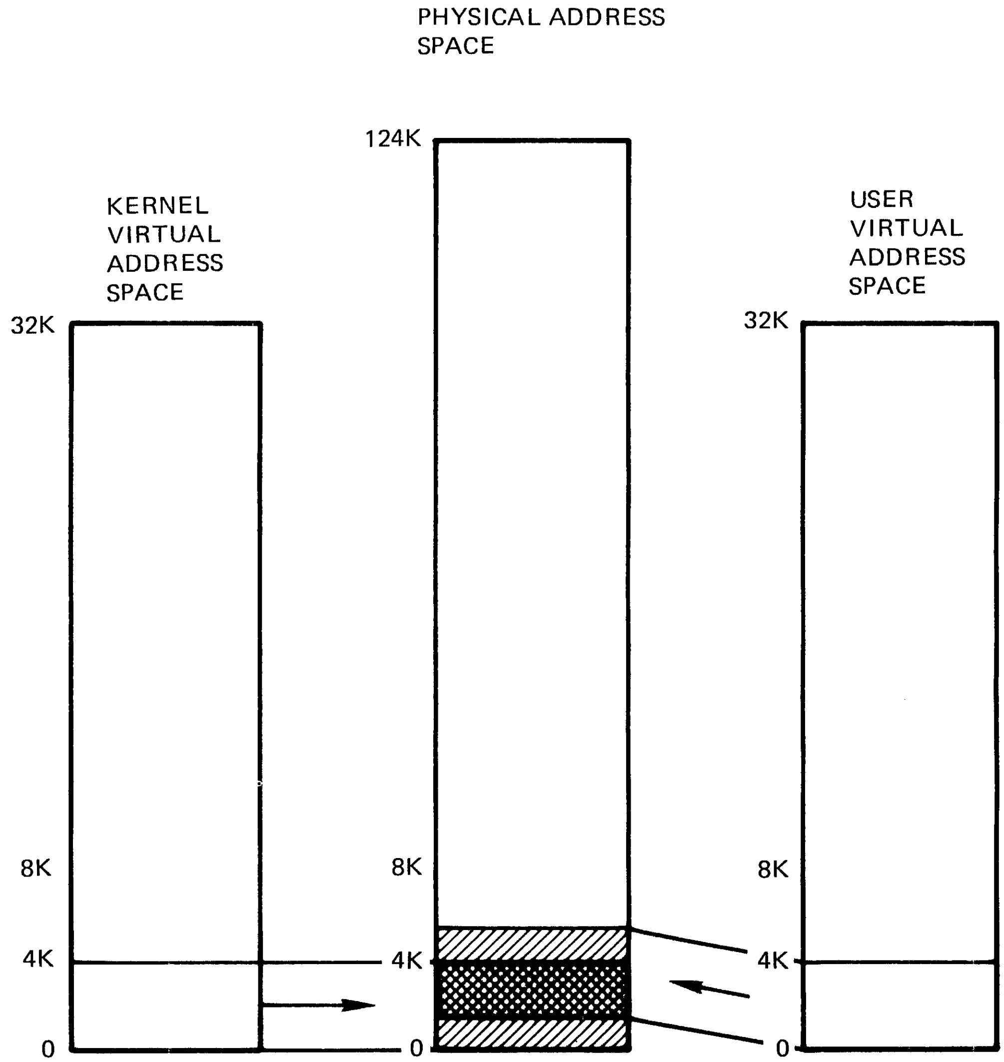


necessarily mean that kernel virtual address 20010 is also mapped to physical address 40210. In fact, kernel virtual addresses are often mapped to different sections of physical memory from user virtual addresses. The mapping depends entirely on the contents of the Active Page Registers. Thus, changing from user to kernel processor mode has some interesting implications: referencing the same virtual addresses in different modes can cause a program to access different physical locations. Figure 4–20 shows an example in which virtual address 0 in kernel mode maps to physical location 0; in user mode, virtual address 0 maps to physical location 500. This is the mapping scheme RT–11 uses for a virtual job at load time.

4.2.8 Default Mapping

Mapping is the process of associating virtual addresses with physical locations (see Section 4.1.6). The RT–11 XM monitor manages the virtual address space by controlling the way the virtual addresses map to physical

Figure 4–20: Mapping the Same Virtual Addresses to Different Physical Locations



locations. The monitor does this by putting values into the Active Page Registers, thereby controlling the Memory Management Unit.

When you first bootstrap an RT–11 extended memory system, kernel and user mapping are identical. That is, the monitor puts the same values into both the kernel and user sets of Active Page Registers. Table 4–1 shows the initial values of the Active Page Registers. Figure 4–21 shows the default mapping that results from these values. Table 4–2 shows the default mapping for a typical 4K virtual background job that has no extended memory overlays and no extra regions.

Table 4–1: Initial Contents of Kernel and User APRs

Page and APR No.	Kernel PAR	Kernel PDR	User PAR	User PDR
7	177600	77406	177600	77406
6	1400	77406	1400	77406
5	1200	77406	1200	77406
4	1000	77406	1000	77406
3	600	77406	600	77406
2	400	77406	400	77406
1	200	77406	200	77406
0	0	77406	0	77406

Figure 4–21: Default Mapping at Bootstrap Time

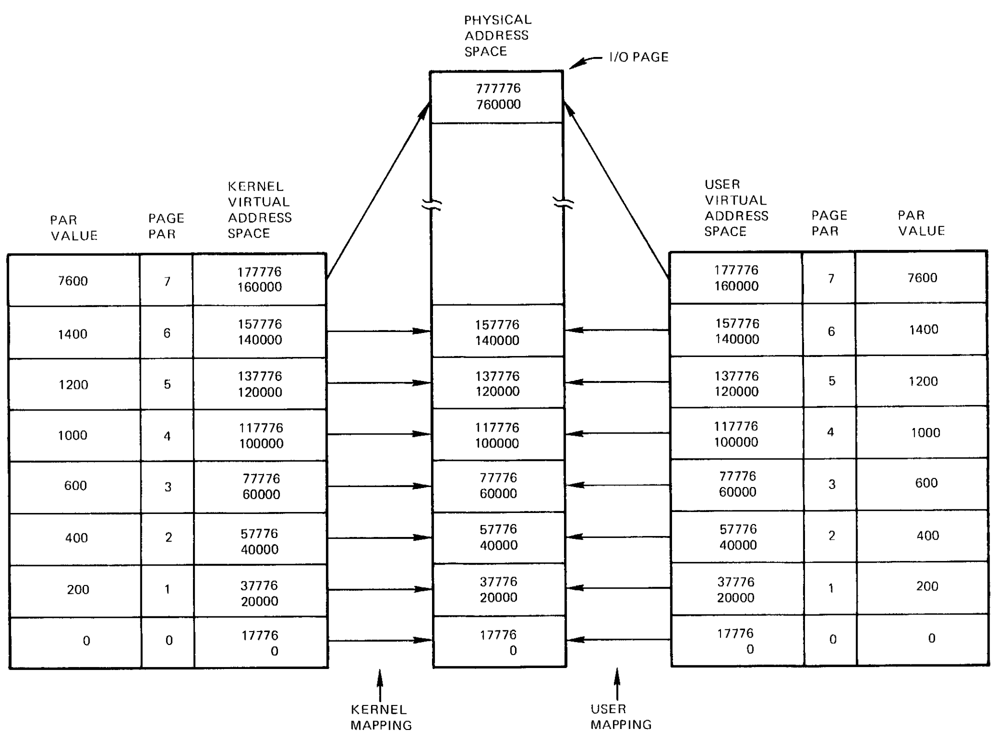


Table 4–2: Initial Register Contents for Virtual Job

Page and APR No.	User PAR	User PDR
7	?	0
6	?	0
5	?	0
4	?	0
3	?	0
2	?	0
1	?	0
0	5	77406

4.3 Software Concepts

RT–11 implements support for extended memory through the extended memory, or XM, monitor. You must perform the system generation process to obtain an XM monitor, since it results from assembling the FB monitor source files with the conditional MMG$T set to 1. One of the major design considerations for RT–11's extended memory support was that the XM monitor should closely resemble the FB monitor.

In addition, you must use a special set of device handlers that can communicate between a peripheral device and extended memory. It is part of the extended memory system design that the USR must be permanently resident.

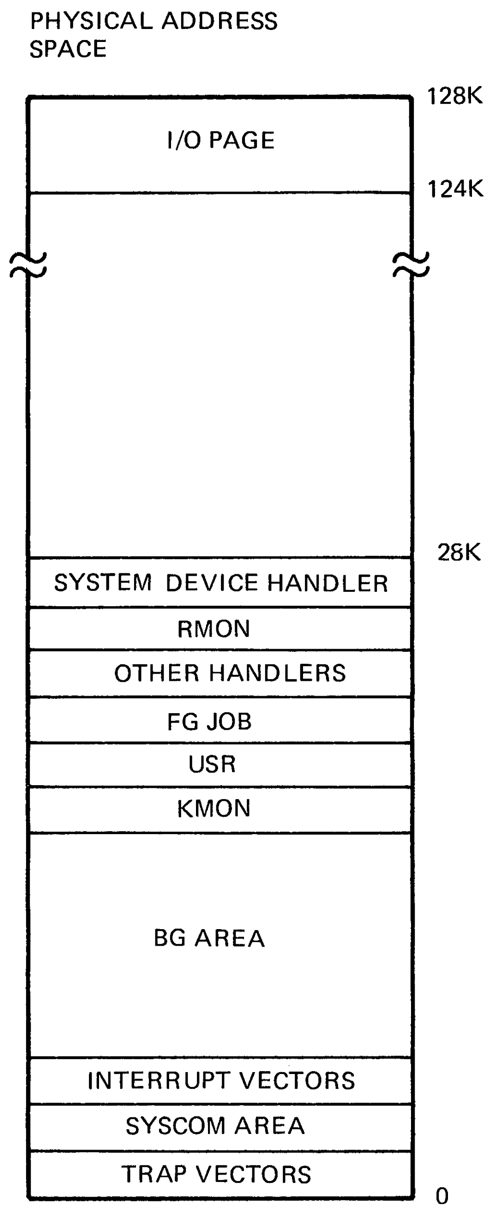
The following sections describe the software concepts RT–11 uses in its extended memory system.

4.3.1 XM System Memory Layout

Figure 4–22 illustrates the locations of the XM system components in physical memory in a 128K-word system. (Notice that this layout closely resembles the FB system arrangement described in Chapter 2.) When you first bootstrap an XM system, the system device handler and the Resident Monitor use the available memory just below the 28K-word boundary so that extended memory — the locations between 28K and 124K — is not used. Other loaded device handlers occupy the space below the Resident Monitor, followed by foreground and system jobs, if any, and the USR.

The Resident Monitor executes in processor kernel mode and can access the low 28K words of memory and the I/O page. The USR also executes in kernel mode and is always memory resident in an XM system. The Keyboard Monitor executes in processor user mode, but since it is a privileged background job, it uses the same mapping as the Resident Monitor. (Privileged jobs are described in Section 4.3.3.2.) Physical locations 0 through 500 contain the vectors.

Figure 4–22: XM System Memory Layout



4.3.2 How Programs Control Mapping

Mapping — associating virtual addresses with physical locations — is the heart of the extended memory system. The XM monitor controls mapping by putting values into the Active Page Registers, thus controlling the Memory Management Unit. Obviously, this level of control is elementary and requires the monitor to keep close watch over the mapping situation.

Fortunately, the monitor provides the means by which system and application programs can direct mapping operations and experience the benefits of accessing extended memory without concern for the specifics of the Memory Management Unit operations. In fact, your programs should never access the Active Page Registers or the Memory Management Unit Status Registers directly. Programs communicate their extended memory requirements to the monitor through a collection of programmed requests. These requests store or modify information in data structures within the programs. Based on the contents of these data structures, the monitor modifies its own internal control blocks and puts the correct values into the Active Page Registers to perform the appropriate mapping action.

In order to access extended memory, a program must:

- Tell the monitor how much physical address space it needs.
- Describe the virtual addresses it needs to the monitor.
- Direct the monitor to associate the virtual addresses with the physical locations. That is, it must **map** the virtual addresses to the physical locations.

Background, foreground, and system jobs can all access extended memory by following the three steps described above. Note, however, that none of the jobs can share physical address space with another job.

The monitor and the programs use certain software concepts to describe the virtual addresses and the physical memory locations. The following sections describe the concepts of **physical address regions**, **virtual address windows**, and the **program's logical address space**.

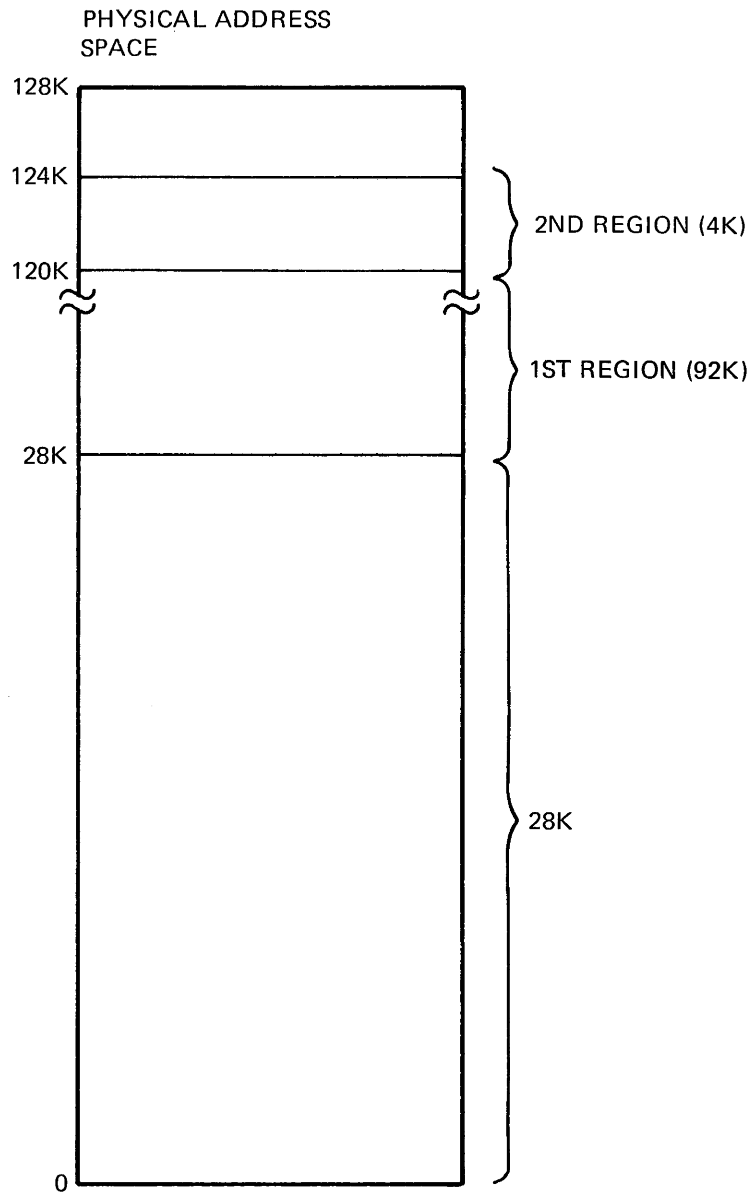
4.3.2.1 Physical Address Regions — A program that needs to access extended memory must communicate to the monitor a description of the physical memory locations it plans to use. The program does this by defining one or more **regions** in extended memory.

A **physical address region** is a segment of physical memory consisting of contiguous 32-word decimal units. A region must begin on a 32-word boundary; it can be as large as 96K words. A job can have as many as four regions at any time, but their total combined size cannot exceed 128K words. The monitor assigns identification numbers to the regions when it creates them. A region identification is actually a pointer within your job's impure area to the start of the region's control block. (You will read more about region control blocks later.)

The purpose of a region is to describe a portion of the physical address space, thus making it available for mapping and permitting a program to use those physical addresses. Sections of physical address space, if any, that are not part of a region are unavailable to a program. Figure 4–23 shows how memory can be divided into regions. Note that two jobs cannot share a region in extended memory.

Information about a physical address region is contained in a three-word data structure in your program, called a **region definition block**. The monitor collects information from the region definition block and stores it in a different internal data structure, called the **region control block**. The

Figure 4–23: Physical Address Space and Two Regions



region control block is located in your program's impure area. Section 4.6 provides more detailed information on the region definition and control blocks.

The Static Region

The first region, called the static region, is created for a **virtual** job by the monitor at run time. (Section 4.3.3 describes the differences between virtual programs and privileged programs.) The size of the static region varies, depending on the size of the program and whether the program is a foreground or background job, but it is always within the low 28K words of memory. You can refer to the static region by using an identification of 0.

Your program cannot eliminate the static region or change it in any way. (You cannot use the first region in privileged jobs, either; its data structures are reserved and currently unused.)

The Dynamic Regions

If your program needs to access more memory than the amount allocated at run time, it can create one to three dynamic regions and map virtual address windows to them. A dynamic region is a portion of physical memory above the 28K-word boundary. The static region is created by the monitor and a program can create up to three more regions. A program can create and eliminate any of the dynamic regions.

4.3.2.2 Virtual Address Windows — A program that needs to access extended memory must also communicate to the monitor a description of the virtual addresses it plans to use. While the monitor uses the concept of **pages** to describe virtual addresses to the Memory Management Unit, programs describe the virtual address space to the monitor by using the software concept of **virtual address windows**.

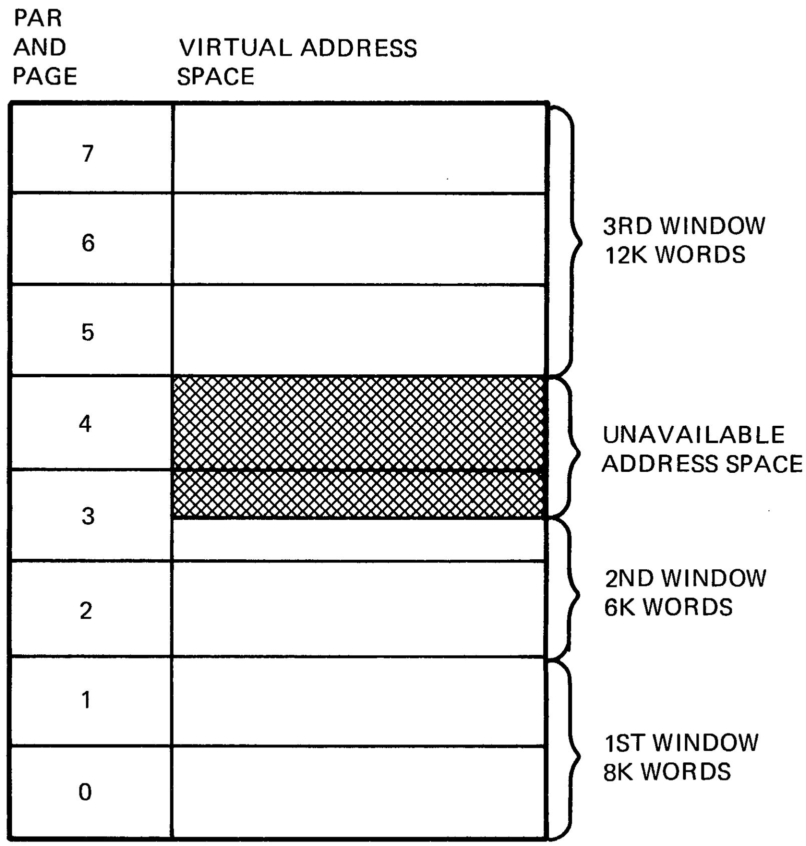
A virtual address window is a section of the 32K-word virtual address space consisting of contiguous 32-word decimal units. A window, like a page, must begin on a 4K-word boundary. However, unlike a page, whose maximum size is 4K words, a window can be as large as 32K words and can encompass one or more pages. There can be as many as eight virtual address windows or as few as one. The monitor assigns identification numbers to the windows when your program creates them.

The purpose of a window is to describe a section of virtual address space to the monitor, and thus permit a program to use those virtual addresses. Windows cannot overlap each other. (While a job can describe a new window that overlaps an existing one, the old one is eliminated when the new one is created.) And, sections of virtual address space, if any, that are not part of a window are not available for a program to use, unless the job is privileged. Each window that is less than 4K words causes a discontinuity in the program's virtual address space. A memory management fault results if the program tries to access a virtual address that does not fall within a mapped window. (A window is not useful until it is also mapped.)

The monitor can assign physical addresses to the virtual addresses encompassed by windows by calculating the number and size of the pages involved and putting values into the corresponding Active Page Registers for those pages. Figure 4–24 shows how virtual address space can be divided into windows.

Information about a virtual address window is contained in a seven-word data structure in your program, called a *window definition block*. The monitor collects information from the window definition block and stores it in a different internal data structure, called the *window control block*. The window control block is located in your program's impure area. Section 4.6 provides more detailed information on the window definition and control blocks.

Figure 4–24: Virtual Address Space and Three Windows



The Static Window

The first window, called the static window, is created for a **virtual** job by the monitor at run time. (Section 4.3.3 describes the differences between virtual jobs and privileged jobs.) The static window begins at virtual address 0, and its size is equal to the size of your program's base segment, up to the program's high limit. The static window contains your program's root, stack, virtual vectors, overlay handler, and low memory overlays. Instructions, data, and buffers can appear in extended memory overlays or in extended memory .SETTOP buffers; they are contained in a different window and region. You can refer to the static window by using an identification of 0. Your program cannot eliminate the static window or change its mapping. (You cannot use the first window in privileged jobs, either; its data structures are reserved and currently unused.)

The Dynamic Windows

If your program needs to access more memory than the amount allocated at run time, it can create one or more dynamic windows and map their virtual addresses to physical locations. The static window is created by the monitor and a program can create up to seven more windows. A program can create, eliminate, map, and remap any of the dynamic windows.

4.3.2.3 Program's Logical Address Space (PLAS) – A program's logical address space is the range of physical address space effectively available to the program as a result of mapping operations. That is, all physical locations that are part of a region can be accessed by the program through mapping operations, and are thus part of its logical address space. The Program's Logical Address Space is abbreviated as PLAS, a term often used to refer to extended memory support in general.

4.3.3 Two Kinds of Mapping

RT–11 provides two kinds of mapping for jobs that run in an extended memory environment: **virtual mapping** and **privileged mapping**. The following sections describe virtual jobs – those that run with virtual mapping – and privileged jobs – those that run with privileged mapping.

4.3.3.1 Virtual Jobs – Jobs that run with virtual mapping execute in the processor's user mode. Virtual jobs do not use kernel mapping; virtual background jobs load into memory at an offset of 500. Virtual jobs cannot load over the USR, the Resident Monitor, or the I/O page. Virtual mapping is the better mapping mode to use for a job that does not require privileged access to the vector area, the monitor, or the I/O page, since it protects these system areas from virtual jobs.

The first 500 bytes of each virtual job image are its virtual vector and system communication areas. The static window includes the virtual addresses between the program's virtual address 0 and its high limit. The size of the static region varies depending on whether the virtual job is a foreground or a background job and on the size of the job.

When you first run a virtual job, it can access only those virtual addresses that are within its own program bounds and that are also mapped to physical memory. However, a virtual job can use any remaining virtual address space between its own high limit and the 32K-word address boundary. It can create one or more regions in extended memory, and one or more virtual address windows. It can then map a window to a region, thus accessing extended memory. If a virtual job unmaps a window, it cannot use the virtual addresses encompassed by the window unless it remaps the window. A virtual job can also use the extended memory .SETTOP feature and extended memory overlays.

Selecting Virtual Mapping

You indicate that a job is to use virtual mapping by setting bit 10 of the Job Status Word before you run the program. If a particular job is always virtual, set bit 10 at assembly time. Use the following instructions to do this:

```
.ASECT
.=44
.WORD   2000
.PSECT
```

Or, if you prefer, select the program's mapping by running SIPP and patching location 44 in the job's .SAV or .REL file before you run the program.

NOTE

> Do not change the value of bit 10 of the JSW when the program is running. Doing so interferes with accurate processing of I/O requests and can cause unpredictable results.

A Virtual Background Job

Use the monitor R or RUN command to start a virtual background job. You can also start the job through CCL by typing only the program name. The file should have the .SAV file type. A virtual background job loads into memory starting at physical location 500. Its highest physical address is equal to the size of the program in octal plus 500.

The static region for a virtual background job begins at physical location 500 and extends to the lowest address used by the USR. This prevents a virtual background job from ever accessing the physical vector area between locations 0 and 500. As a result, the vectors are protected from virtual jobs. Figure 4–25 illustrates the mapping for a virtual background job in a 128K-word system. Figure 4–26 shows how a virtual background job can map a window into the static region to use the available memory just below the USR in a 128K-word system.

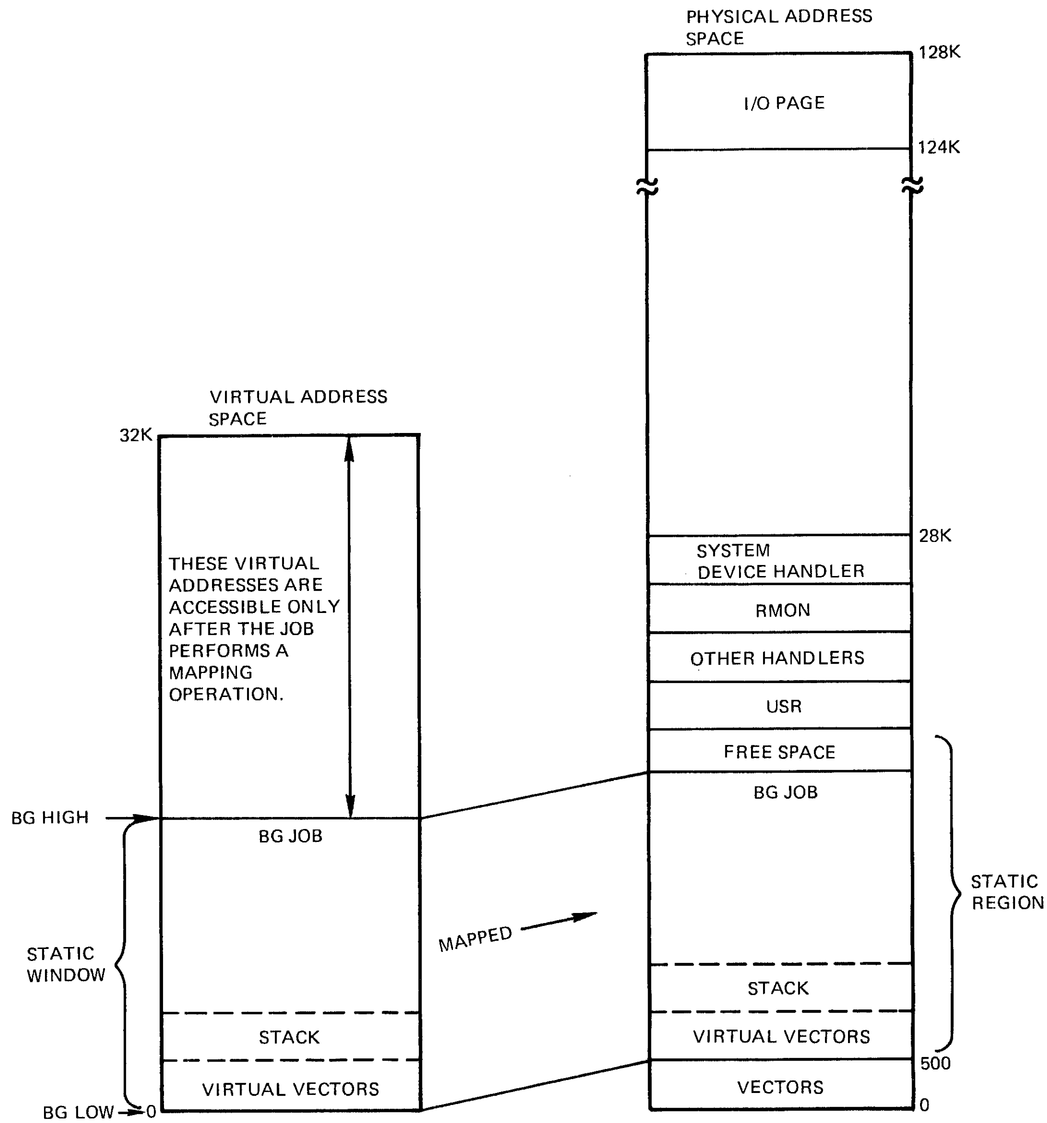
A Virtual Foreground or System Job

Use the FRUN monitor command to start a virtual foreground job and the SRUN command to start a virtual system job. You should link these jobs as background jobs with the .SAV file type, rather than as foreground or system jobs with the .REL file type. You can FRUN or SRUN a virtual .SAV image because virtual foreground jobs require no relocation information. Thus, the .SAV files are smaller on disk than .REL files, and they load into memory faster.

When a foreground job is loaded, it uses the physical locations just below the lowest loaded handler or previously loaded system job. The USR slides down in memory, if necessary, to accommodate the foreground job. The foreground job is linked with a default base address of 1000 (unless it is a .SAV image); its virtual addresses between 0 and 500 represent the virtual vector and system communication areas. As with the background virtual job, the static window starts at virtual address 0 and extends to this foreground program's high limit, rounded up to a 32-word multiple.

The static region begins at physical location 0 and extends to the program's physical high limit. The foreground impure area is located in physical memory just below the program. However, no virtual addresses are mapped to the impure area, so a virtual foreground job cannot access the contents of the impure area. As a result, the impure area is protected from a virtual foreground job. Figure 4–27 illustrates the mapping for a virtual foreground or system job.

Figure 4–25: Virtual Background Job

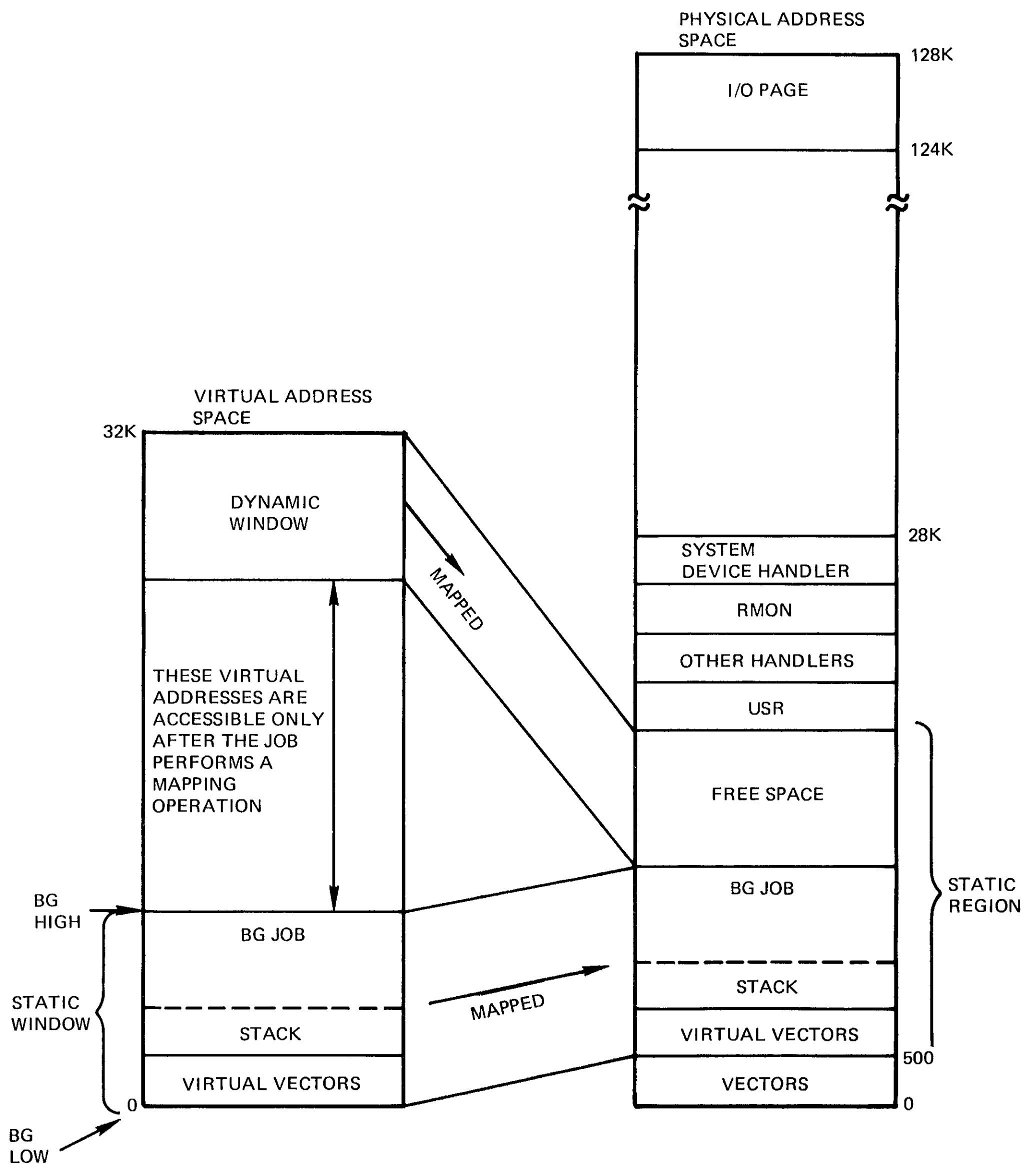


4.3.3.2 Privileged Jobs — The default mapping in an extended memory system is privileged. To indicate a privileged job, bit 10 of the Job Status Word remains 0. The XM environment appears to a privileged job to be very similar to an SJ or FB environment. A privileged job can access the low 28K words of memory as well as the I/O page. All the RT–11 utility programs run as privileged jobs in an extended memory environment.

Privileged jobs, like virtual jobs, run in user processor mode. However, the monitor copies the contents of the kernel Active Page Registers into the user Active Page Registers. The default mapping for privileged jobs is thus the same as the default kernel mapping.

Privileged jobs do have all 32K words of virtual address space available to them. But much of that virtual address space is already mapped to operating system software, the I/O page, and — in the case of a privileged foreground or

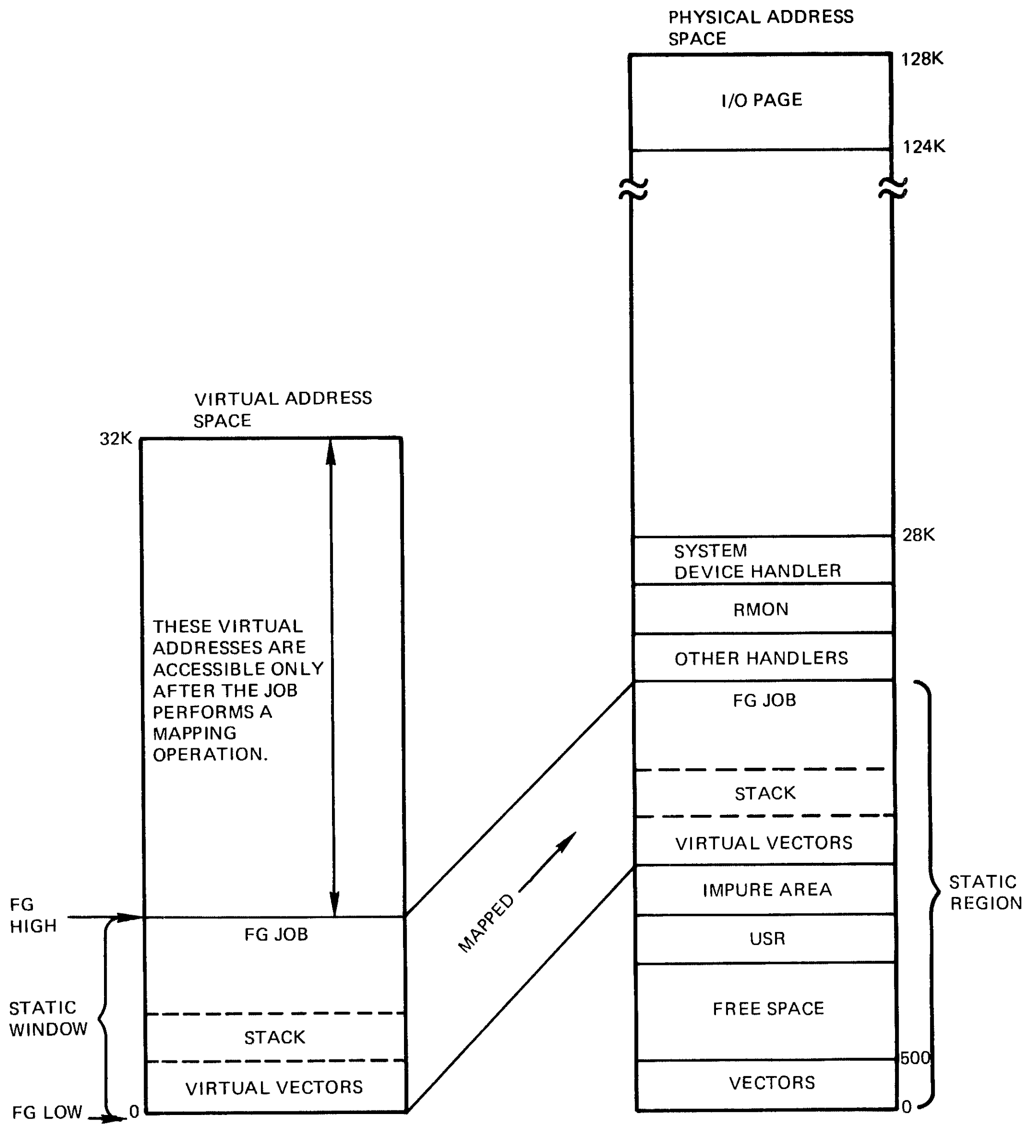
Figure 4–26: Virtual Background Job Mapping into the Static Region



system job — to a background job or the Keyboard Monitor. A privileged job can alter its default mapping through the use of extended memory overlays or programmed requests. It can map away all or part of the operating system to obtain a full 32K words of addressable memory for itself. For example, a program that needs to access the I/O page for only a limited time can explicitly map away from the I/O page when it is done using it.

Note that the static window and static region concept does not apply to privileged jobs. However, one window and one region are reserved by the monitor. Thus, privileged jobs have seven dynamic windows and three dynamic regions available to them, just as virtual jobs do.

Figure 4–27: Virtual Foreground or System Job



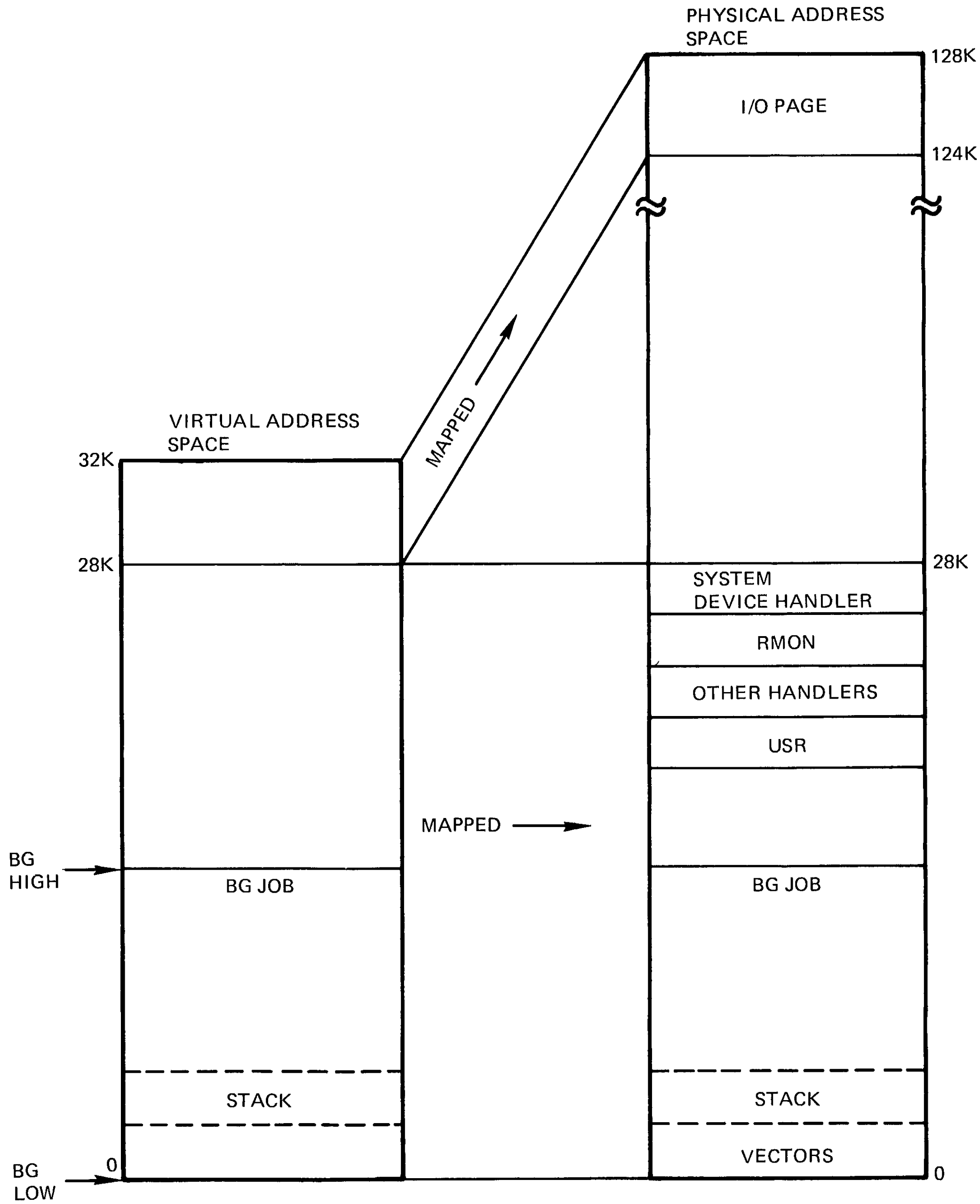
When a privileged job creates a window and executes the mapping programmed requests, the default privileged mapping for that virtual address space is temporarily unmapped. The monitor maps the window using the contents of the internal window control block to the new region of memory. When the privileged job unmaps the window, the monitor remaps that virtual address space according to the contents of the kernel Active Page Register set. This differs from a virtual job that unmaps a window, in which the virtual addresses encompassed by the window are unusable until the window is remapped.

Since interrupt service routines execute in kernel mapping, privileged jobs containing user interrupt service routines should not change the mapping of interrupt service routines, the I/O page, or parts of the monitor during any time period in which an interrupt could possibly occur. The monitor depends on the fact that kernel and user mapping are identical when it services user interrupts.

Privileged Background Job

Use the monitor R or RUN commands to start a privileged background job. Figure 4–28 illustrates the mapping for a privileged background job.

Figure 4–28: Privileged Background Job

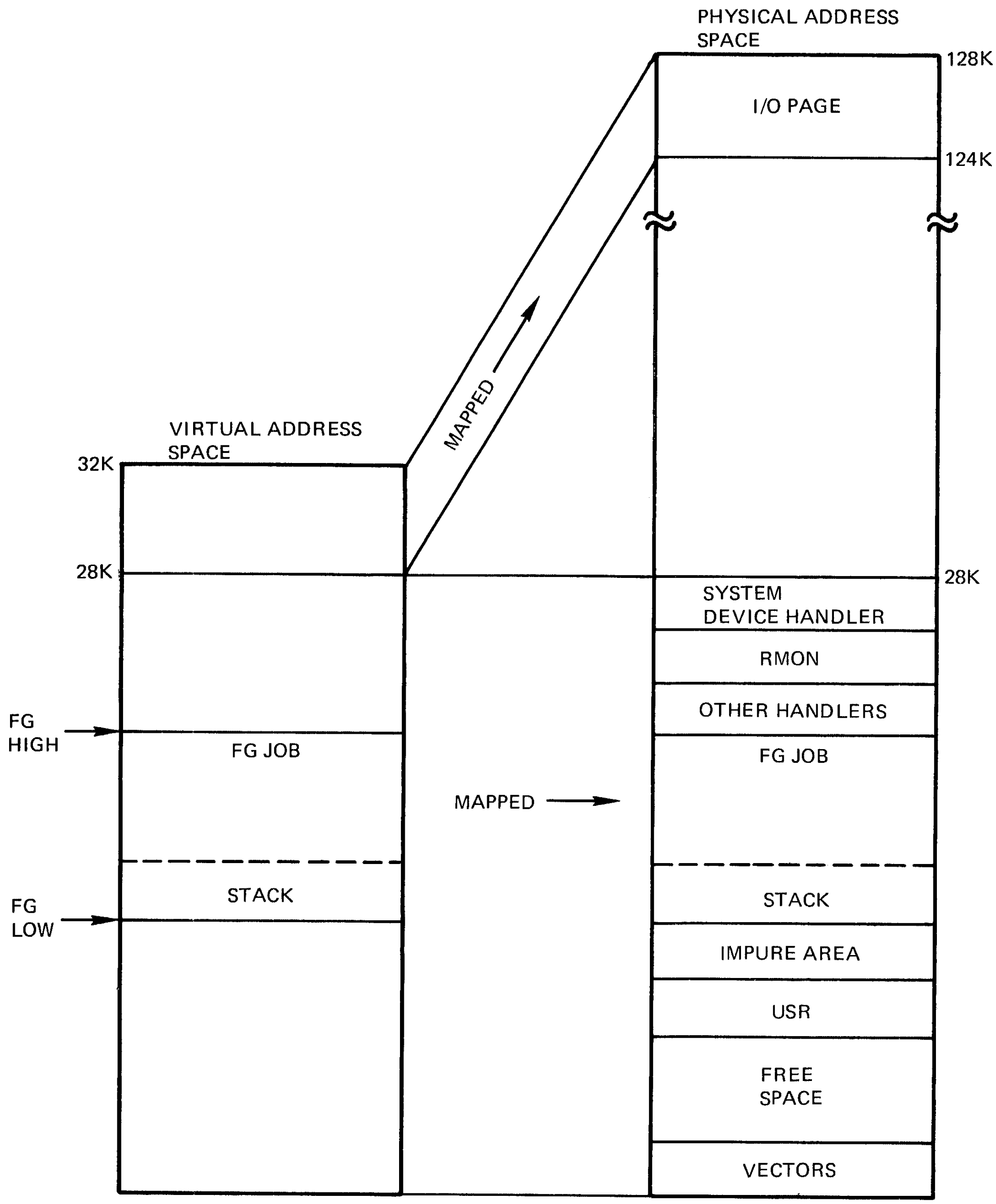


Privileged Foreground or System Job

Use the monitor FRUN command to start a privileged foreground job. Use the SRUN command to start a privileged system job.

Figure 4–29 illustrates the mapping for a privileged foreground or system job.

Figure 4–29: Privileged Foreground or System Job



4.3.3.3 Differences Between Virtual and Privileged Jobs — Table 4–3 summarizes the differences between virtual and privileged jobs.

Table 4–3: Comparison of Virtual and Privileged Jobs

Characteristic	Virtual Job	Privileged Job
Value in bit 10 of JSW	1	0
Original amount of address space available	Accesses only the virtual addresses within its own program bounds.	32K words. Accesses the low 28K words of memory plus the I/O page.
Amount of potential address space	32K words. Creates windows to describe the virtual address space between its own high limit and the 32K word boundary.	32K words. If some portions of virtual address space are already in use (by a background job, for example), this job can unmap them and remap the addresses to memory above 28K words. It must leave certain areas mapped whenever a user interrupt service routine could run.
Benefits	Provides protection for operating system software and other programs; takes minimal physical memory away from other jobs.	Compatible with FB and SJ systems.
Starting procedure	BG: R, RUN, or CCL command (.SAV) FG: FRUN or SRUN (.REL, .SAV; .SAV is recommended)	BG: R, RUN, or CCL command (.SAV) FG: FRUN or SRUN (.REL)
Static window	Extends from program's virtual address 0 to its high limit.	None – all are dynamic.
Static region	BG: Extends from physical location 500 to the lowest address used by the USR. FG: Extends from physical location 0 to the physical high limit of the job.	None – all are dynamic.
Possible number of windows	7 plus the static window. 7 (1 window reserved)	
Possible number of regions	3 plus the static region.	3 (1 region reserved)

4.3.3.4 Context Switching Between Virtual and Privileged Jobs — In an RT–11 system with more than one job, the monitor saves job-dependent information when a new job replaces the one currently running. The monitor restores this information when the original job executes again. This procedure, called *context switching*, is described in detail in Section 3.4.2.

In an XM system, each job in memory could be either a virtual or a privileged job. The monitor, therefore, has more work to do when it switches context in an XM system.

When the monitor switches out the current job, it saves the information listed in Section 3.4.2. However, the monitor never saves the contents of the Active Page Registers that the current job uses. For this reason, your programs should never manipulate the Memory Management registers directly; their contents are lost during a context switch. The monitor also ignores a .CNTXSW programmed request if it occurs in a virtual job. The entire job is saved by the switch, and the virtual job is not permitted to access the vector area in any case.

When the monitor switches in a new job, it assumes at first that the new job is privileged. It copies the contents of the kernel mapping registers into the user registers. The job can then access the low 28K words of memory plus the I/O page. Next, the monitor checks to see if the new job is the Keyboard Monitor. If it is, execution continues with no further modifications.

If the new job is a privileged job, the monitor next checks the window and region control blocks in the job's impure area. If the job defined and mapped one or more windows, the monitor restores the mapping based on the contents of the internal control blocks, thus altering the default privileged mapping for those windows.

If the new job is virtual, the monitor clears the user mapping registers. Then it scans the window and region control blocks in the job's impure area. The monitor maps only the portion of the job's virtual address space that was defined in a window and mapped to a region at the time the job was switched out. Of course, any attempt to access an unmapped address causes a memory management fault. Unused portions of virtual address space remain unmapped unless the virtual job explicitly maps them.

4.4 Typical Extended Memory Applications

The following sections assume you understand the fundamental concepts of extended memory systems; they should help you see how to use extended memory. Some arrangements are suggested that may suit your own particular situation. As you read, keep in mind what benefits you want from an extended memory system. In other words, why do you want to use it?

4.4.1 Extended Memory Overlays

The low 28K words of memory fill up rapidly with the Resident Monitor, device handlers, the USR, a foreground job, one or more system jobs, and a background job. To optimize use of this space and relieve the congestion, make the root segments of the foreground, system, and background jobs (if they are overlaid) as small as possible. Instead of segmenting the programs and using disk overlays though, you can put the overlays into extended memory. Make all the programs virtual jobs, unless they really need to access the monitor or the I/O page.

Instead of accessing the I/O page directly from your program, consider writing a device handler. .SPFUN requests allow a great deal of flexibility in writing special handlers for unusual devices.

The root segment can be minimal in size. All you need put there are queue elements, channels, interrupt service routines (if any – there are none in virtual jobs), and a JMP instruction to the first overlay. The overlay segments can be permanently resident in extended memory to speed up execution.

You can use the linker's /V option to put your overlay segments into extended memory. The Keyboard Monitor creates a region at run time, using information in the overlay handler and tables. The overlay handler creates and maps windows. Figure 4–30 shows a simple virtual background program that uses extended memory overlays in 128K words. You can find detailed information on extended memory overlays in the *RT–11 System User's Guide.*

4.4.2 Large Buffers or Arrays in Extended Memory

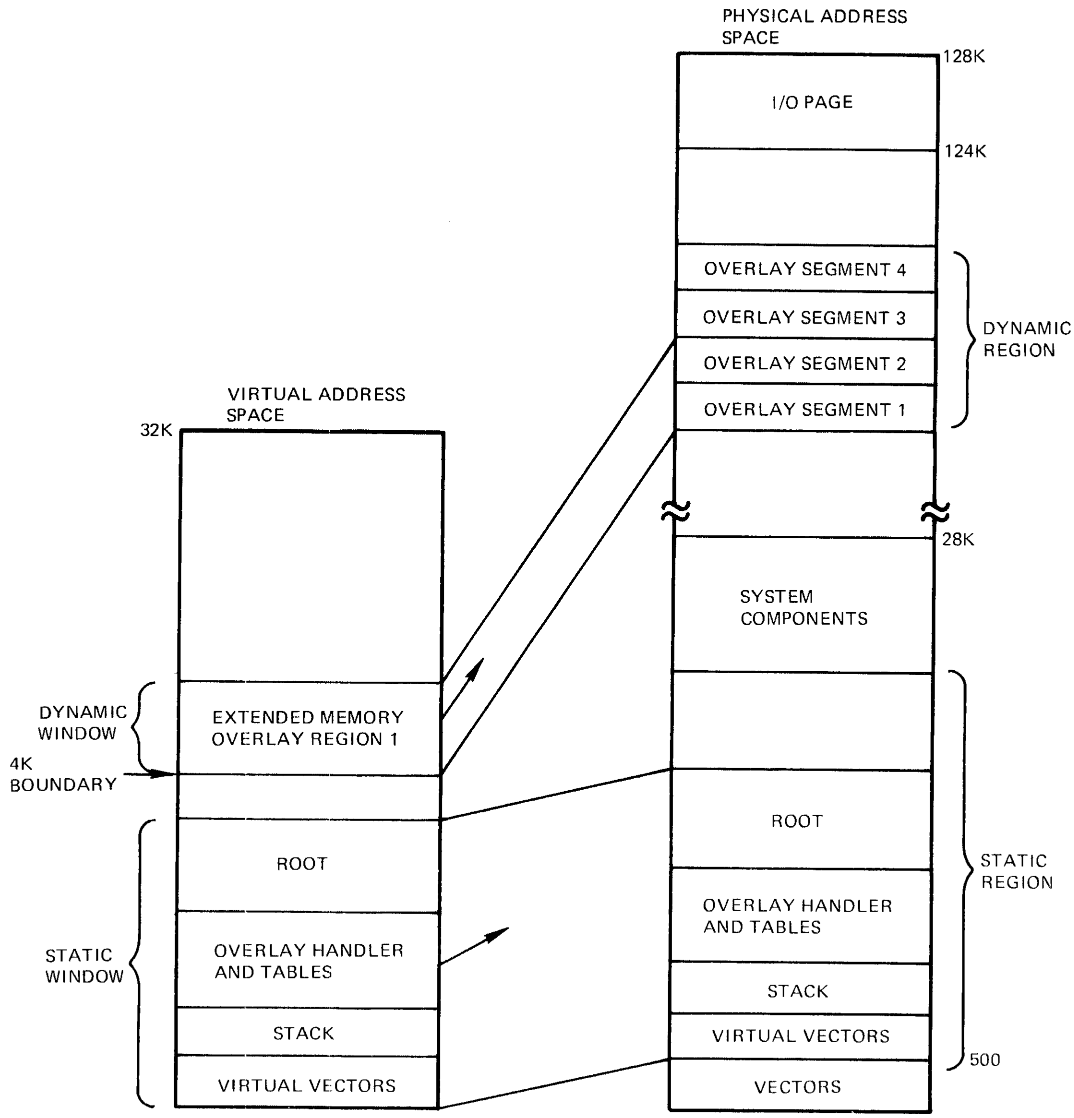
In order to put a large buffer or array into extended memory, you first create a region large enough to accommodate the array. Next, decide how much virtual address space your program can commit to accessing the array and create a virtual address window of that size. Then simply write a subroutine that translates references to the array into instructions to remap the window into the correct part of the region. Figure 4–31 illustrates this situation in 128K words. (The extended memory feature of the .SETTOP programmed request can create an extended memory buffer automatically. See Section 4.4.4 for information.)

4.4.3 Multi-User Program

An extended memory system is ideal for implementing a multi-user application. For example, you could develop a language interpreter that several programmers could use simultaneously. To implement this application, separate your program into two sections: a pure code section that contains the interpreter, and a separate read/write work area for each user. Select part of your virtual address space to be the user scratch area, and create a window of that size. Next, decide how many users you want and create a region equal to the number of users times the size of the window. The interpreter can change user context by remapping the window. Figure 4–32 shows a multi-user program in 128K words.

Your multi-user program can use extended memory overlays. In this case, use one region for the overlays and one for the work areas.

Figure 4–30: Virtual Background Job with Extended Memory Overlays

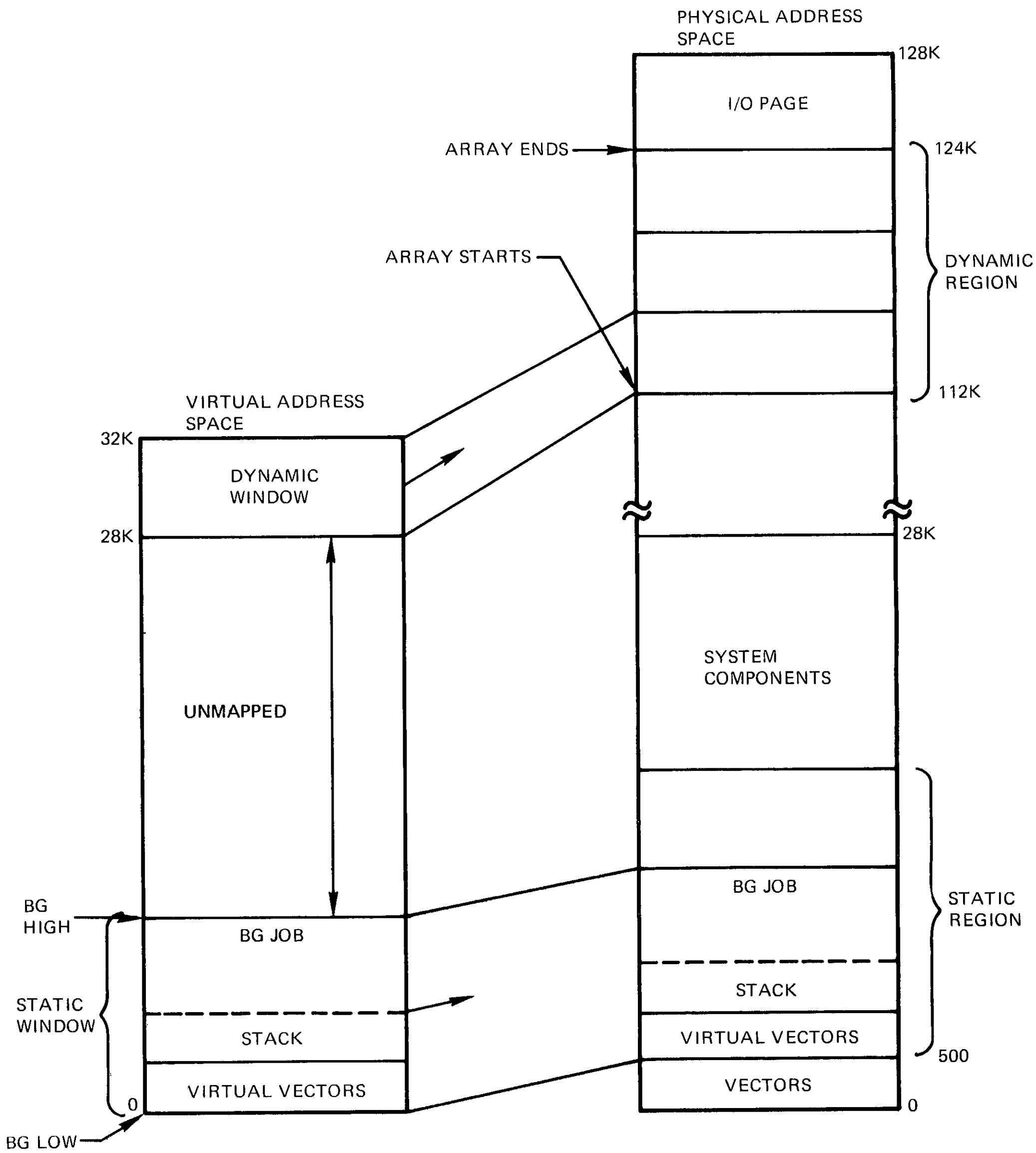


4.4.4 Work Space in Extended Memory

Another application for you to consider is putting a work area into extended memory instead of writing it to disk.

Consider how jobs in an FB system obtain the most space possible for dynamic buffering. A background job gets extra space by issuing a .SETTOP programmed request. It can obtain the space above the job image up to the top of the USR. To obtain extra space for a foreground job, you must allocate it with the FRUN/BUFFER:n command. Once the space is reserved by FRUN, the program can determine its size and claim it with a .SETTOP programmed request. In both cases, the extra space is within the 28K words of low memory.

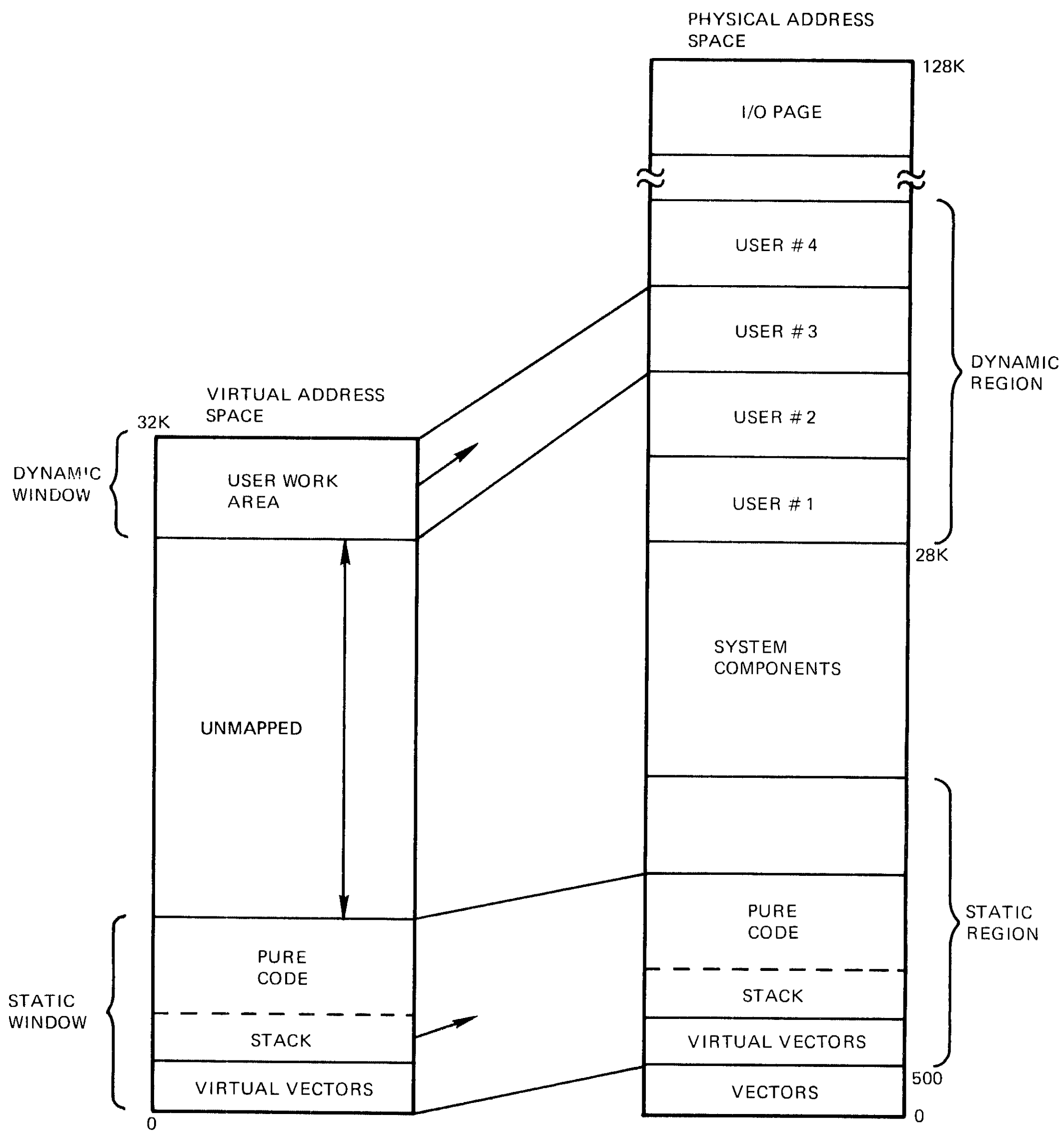
Figure 4–31: Virtual Background Job with an Array in Extended Memory



In an XM system, extra space can be allocated from the physical space either above or below the 28K-word boundary. This feature can make jobs runnable that require too much memory for an unmapped RT–11 system. The ability to allocate extra space is most useful to virtual jobs because they can obtain space up to virtual address 177776 (32K words) by using the XM feature of the .SETTOP programmed request. All the memory obtained by .SETTOP is in extended memory; virtual foreground jobs do not require the FRUN/BUFFER:n command to allocate extra space.

4.4.4.1 Enabling the XM Feature of the .SETTOP Programmed Request – There are two ways to enable the XM feature of the .SETTOP programmed request. If your program has extended memory overlays, using the linker /V

Figure 4–32: Multi-User Virtual Background Program



option to create them enables the XM .SETTOP programmed request automatically. It also enables the XM feature of the .LIMIT directive (see Section 4.4.4.4), links the extended memory overlay handler (VHANDL) into your job image, and establishes an extended memory overlay structure. You use the /V option by issuing the LINK/PROMPT monitor command, and then specifying /V on a subsequent command line.

If your program has no overlays, or if it has only low memory overlays that you create with the linker /O option, you enable the XM feature of the .SETTOP programmed request by using the LINK command with the /XM option. The /XM option enables the XM .SETTOP programmed request and the XM .LIMIT directive. It does not link the extended memory overlay handler into your job image, nor does it establish an extended memory overlay structure for your program.

For all programs, the .LIMIT directive returns as its high value the next available location for the job. The extra space your program obtains with .SETTOP in an extended memory system always begins at the octal address returned as the high value from the .LIMIT directive. This is true for all programs, whether or not they enable the XM feature of the .SETTOP programmed request.

Section 4.4.4.3 describes how .SETTOP works when you execute a program in an extended memory environment without enabling the XM feature of .SETTOP. Section 4.4.4.4 shows how the XM feature of .SETTOP works after you enable it at link time; it also describes the XM feature of the .LIMIT directive.

4.4.4.2 Program and Virtual High Limits and the Next Free Address — To understand XM .SETTOP, it is important that you understand the differences between the **program high limit**, the **virtual high limit**, and the **next free address**. Figure 4–33 shows a program's virtual address space. This program has both low memory overlays created with the /O linker option, and extended memory overlays created with the /V linker option. The **program high limit** is the highest virtual address used by the program's root segment and its low memory (/O) overlay regions, if any exist. The **virtual high limit** is the highest virtual address used by the extended memory (/V) overlay regions, rounded up to a 32-word decimal boundary, minus 2. (In octal, the low-order two digits of the address are always 76.) This is the value that prints on the link map as *nnnnnn*, as the following example shows:

```
Virtual high address = nnnnnn = ddddd. words, next free address = mmmmmm
```

The linker has to calculate the value of the *next free address*. For a job that enables the XM feature of .SETTOP, it rounds up the virtual high limit to the next 4K-word boundary. The next free address, then, is the last word of the virtual address space encompassed by the highest Page Address Register used by the job, plus 2. It is always on a 4K-word boundary. (In octal, the next free address is always a multiple of 20000.)

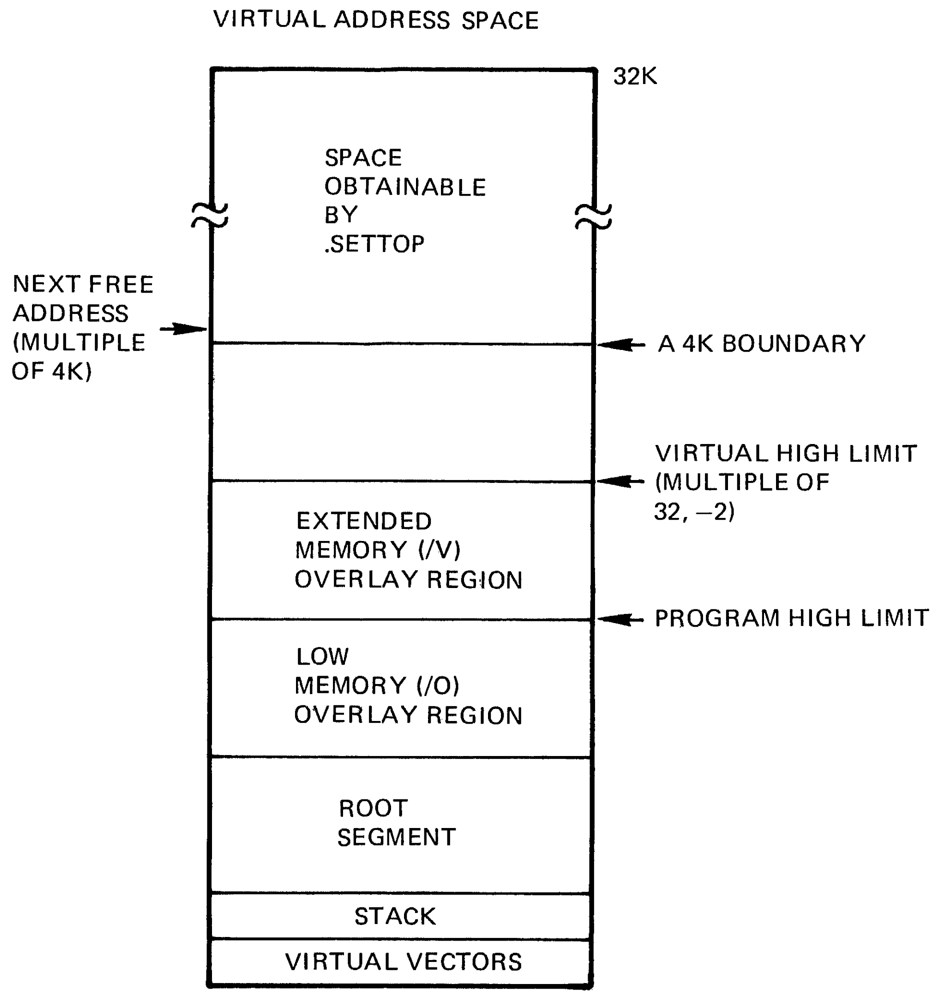
As an example, consider a job with extended memory overlays whose virtual high limit is 55076. Its next free address calculated by the linker is 60000, or the start of the next 4K words of virtual address space. This is the value that prints on the link map as the "next free address". The following example shows the values in our example situation:

```
Virtual high address = 055076 = ddddd. words, next free address = 060000
```

Of course, if a program has no extended memory overlays, it does not have a virtual high limit, and its program high limit is not rounded up. The link map for programs without overlays and for programs whose overlays were created solely by the /O option prints the program high limit as *mmmmmm*, as the following example shows. (The following line prints on all link maps, whether or not extended memory is present.)

```
Transfer address = nnnnnn, High limit = mmmmmm = ddddd. words
```

Figure 4–33: Program and Virtual High Limits, and the Next Free Address



4.4.4.3 Non-XM .SETTOP – If you do not enable the XM .SETTOP feature through the linker, using .SETTOP in an extended memory program has only limited value.

For a privileged job that does not alter the default mapping, .SETTOP works the way it does in an ordinary SJ or FB system. If a privileged job creates a virtual address window and maps it to an extended memory region, the program high limit is not affected by the mapping. The value returned by .SETTOP still represents the highest address available to the program in the low 28K words of memory.

When the monitor performs address checking for programmed requests, it looks first to see if the address (of an argument block, a data buffer, and so on) is entirely within a mapped dynamic window. If it is not, the monitor checks to see if the address is within the job's low memory area. If the address fails both these checks, a monitor error results and the job aborts.

If the job is virtual, the program high limit at load time is set to the highest virtual address used by the root segment and any low memory (/O) overlays. If your job performs its own mapping operations, they do not affect the program high limit as far as .SETTOP is concerned. So, the .SETTOP request is meaningless to these virtual jobs. The non-XM .SETTOP request deals exclusively with the low 28K words of memory. Virtual jobs use the processor user mode and, therefore, are mapped according to the contents of the user Active Page Register set. The virtual job is prevented from accessing memory outside itself (because it is not mapped to any memory but its own dedicated physical space), so issuing a .SETTOP request in a virtual job without the LINK/XM command or the linker /V option does not obtain any extra memory. The value returned can be used by the virtual job to do its own mapping of the area available and then use it.

When the monitor performs address checking for a virtual job, it ignores the program limits and simply checks to see that the virtual address is within a window that is currently mapped. If the address is not within a mapped window, a memory management fault results.

4.4.4.4 XM .SETTOP – When you enable the XM feature of .SETTOP, as Section 4.4.4.1 describes, .SETTOP becomes valuable to privileged and virtual jobs alike, although its value to privileged jobs is limited.

For virtual jobs, not only does .SETTOP obtain virtual address space above the virtual high limit starting at the program's next free address, but it also automatically maps the extra space to physical space. As a result, a job in an extended memory environment can issue a .SETTOP programmed request and obtain more usable virtual address space without concern for the details of managing extended memory.

For privileged jobs, XM .SETTOP functions the way non-XM .SETTOP does, with the following exception: in privileged jobs, the XM .SETTOP request uses the new XM .LIMIT high value as the next free address, thus always returning the start of the buffer on a 4K-word boundary. A .SETTOP to any address below this 4K-word boundary is not permitted.

For both privileged and virtual programs, the linker puts two words of information into locations 0 and 2 of the job image file. Location 0 contains the Radix–50 code for *VIR*. Location 2 contains the value of the *next free address minus 2*, which can be significantly different from the *virtual high limit*.

.LIMIT Directive

For jobs in SJ and FB systems, and in XM systems without the XM feature of .SETTOP, the .LIMIT MACRO directive returns two values to your program. These values are:

- The lowest virtual address used by the program (usually 0)
- The program high limit + 2 (for example, 1644 + 2, or 1646)

In XM programs that enable the XM feature of .SETTOP, .LIMIT returns a significantly different value:

- The lowest virtual address used by the program (usually 0)
- The next free address (always on a 4K-word boundary), which is usually not equal to the program high limit + 2.

Gaps in Virtual Address Space

The linker always starts each extended memory (/V) overlay region at a 4K-word boundary in your program's virtual address space. This restriction results from hardware requirements. Because of this there can be a gap between the program high limit and the start of the virtual overlay region. Your program causes an error if it attempts to reference the virtual addresses within this gap. Similarly, any extra virtual address space that XM .SETTOP obtains for your program also starts on a 4K-word boundary. This means that a gap can exist between your program's virtual high limit and the start of the extra space. Your program cannot reference the addresses within this gap. Figure 4–34 illustrates a typical program with both low memory (/O) and extended memory (/V) overlays.

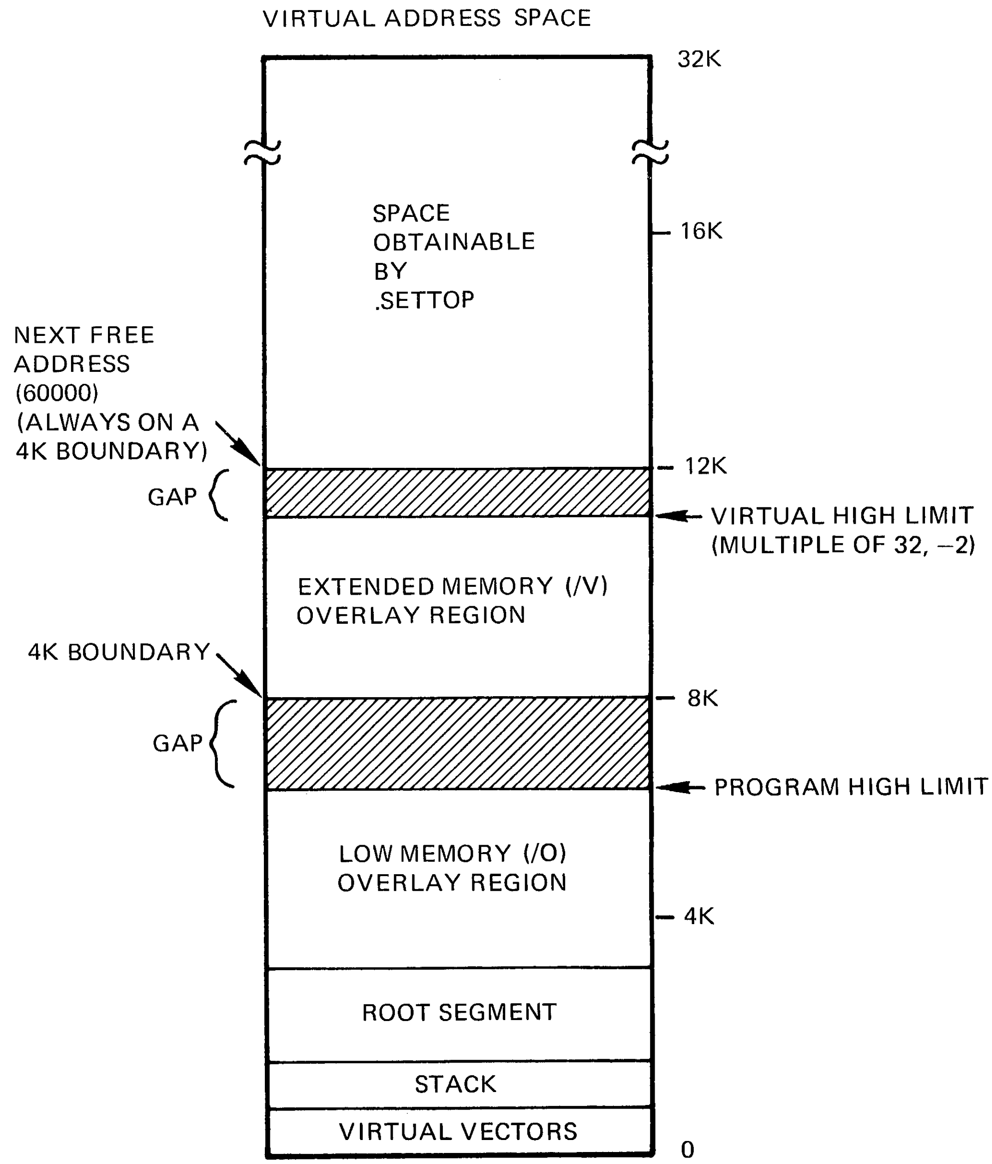
4.4.4.5 XM .SETTOP and Privileged Jobs – When a privileged job issues a .SETTOP request, if the next free address is above the base of the USR, the program is already using the virtual address space above the start of the monitor. Since there is no free memory that can be mapped starting at the program's next free address, the monitor cannot obtain any more space for this program. Thus, a privileged job can never obtain space above SYSLOW, the base of the USR. The .SETTOP request returns the value of the next free address minus 2 to location 50 in your program and to R0. This is the highest usable address.

If there is memory available, the monitor tries to obtain it, basing the size of the area on the argument you specify with .SETTOP. The memory is always within the low 28K words. A privileged job can never obtain an amount of virtual address space less than its own next free address minus 2. In addition, the next free address obtained with XM .SETTOP is always on a 4K-word boundary, and the job cannot issue a .SETTOP for any address below that. Therefore, the job loses the space between its last used address and the next 4K-word boundary.

Privileged Background Jobs

Figure 4–35 shows a privileged background job and all its limits in 128K words. When no foreground job is present in memory, the background job can obtain some space through .SETTOP. Often, there is still space available even when a foreground program is present.

Figure 4–34: Gaps in Virtual Address Space

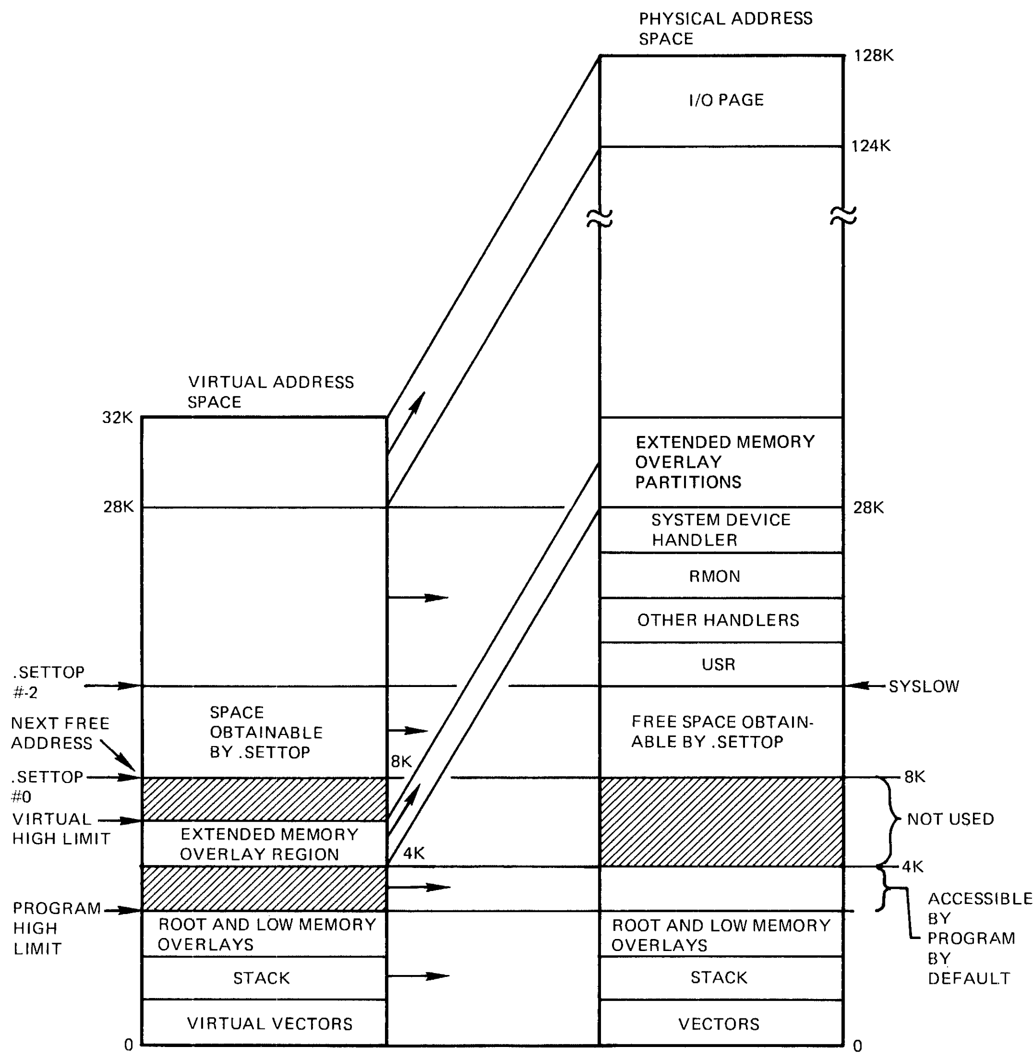


Privileged Foreground Jobs

Since foreground jobs load into memory just below the last device handler and above the USR, there is no extra space available for them through a .SETTOP request.

Because of this situation, privileged foreground jobs are prohibited from using extended memory overlays. This also means they cannot use the linker /V option (either through LINK/FOREGROUND/PROMPT or through LINK/FOREGROUND/XM) to enable the XM feature of .SETTOP and .LIMIT.

Figure 4–35: Privileged Background Job



4.4.4.6 XM .SETTOP and Virtual Jobs — The monitor checks to see if there is some extended memory available. If the next free address is 200000, the program is already using the virtual address space controlled by Page Address Register 7. The request returns the value 177776 in location 50 and in R0.

If .SETTOP can obtain virtual space starting with the next free address (on a 4K-word boundary), the monitor creates a region in extended memory for the necessary amount of space. If not enough space is available, the monitor creates as large a region as possible. (Be sure to check the value .SETTOP returns.) Then the monitor creates a window and maps it to the new region. It returns the new value of the highest available address in location 50 and in R0. If there is no space at all available, or if there are no region or window control blocks available, the request returns the value of the original highest available address in location 50 and in R0.

So, for example, if you issue a .SETTOP request with an address argument, the monitor maps the virtual address space starting at the next 4K-word boundary above the program's virtual high limit, up to and including the address you specify. It maps so that the address specified is mapped, but up to 31 decimal additional words can also be mapped.

If the address you specify in the .SETTOP request is below the highest used address, .SETTOP returns the value of the next free address minus 2 in location 50 and in R0. The static window and virtual overlay regions created with the linker /V option cannot be eliminated by using an argument to .SETTOP.

Assuming your first .SETTOP succeeded and an extended memory region exists for your program, you can issue subsequent .SETTOP requests to control the region. Note, however, that you cannot create yet another region to obtain any more space.

If the argument you specify in your next .SETTOP request is lower than the original next free address minus 2 from the link map, the monitor returns the old next free address minus 2 in location 50 and in R0 and eliminates the region and window, if present (along with any data stored there). You can, of course, issue another .SETTOP later to create a new region again. You can also adjust the size of the buffer by remapping within the same region.

To obtain a larger region, first issue a .SETTOP for a value below the current high limit, which eliminates the region and any data stored there. Then issue another .SETTOP for a larger value, which creates a new region. (Any data stored in the first buffer will be lost.) Note also that to ensure the integrity of your data, only one window exists for the .SETTOP area in an extended memory system.

To get less memory than a previous .SETTOP obtained, issue another .SETTOP with an address argument less than the first one but equal to or greater than the next free address. As a result, the size of the window still equals the size of the region, but a smaller amount of the window is mapped. This does not make any extended memory available for other users or other regions.

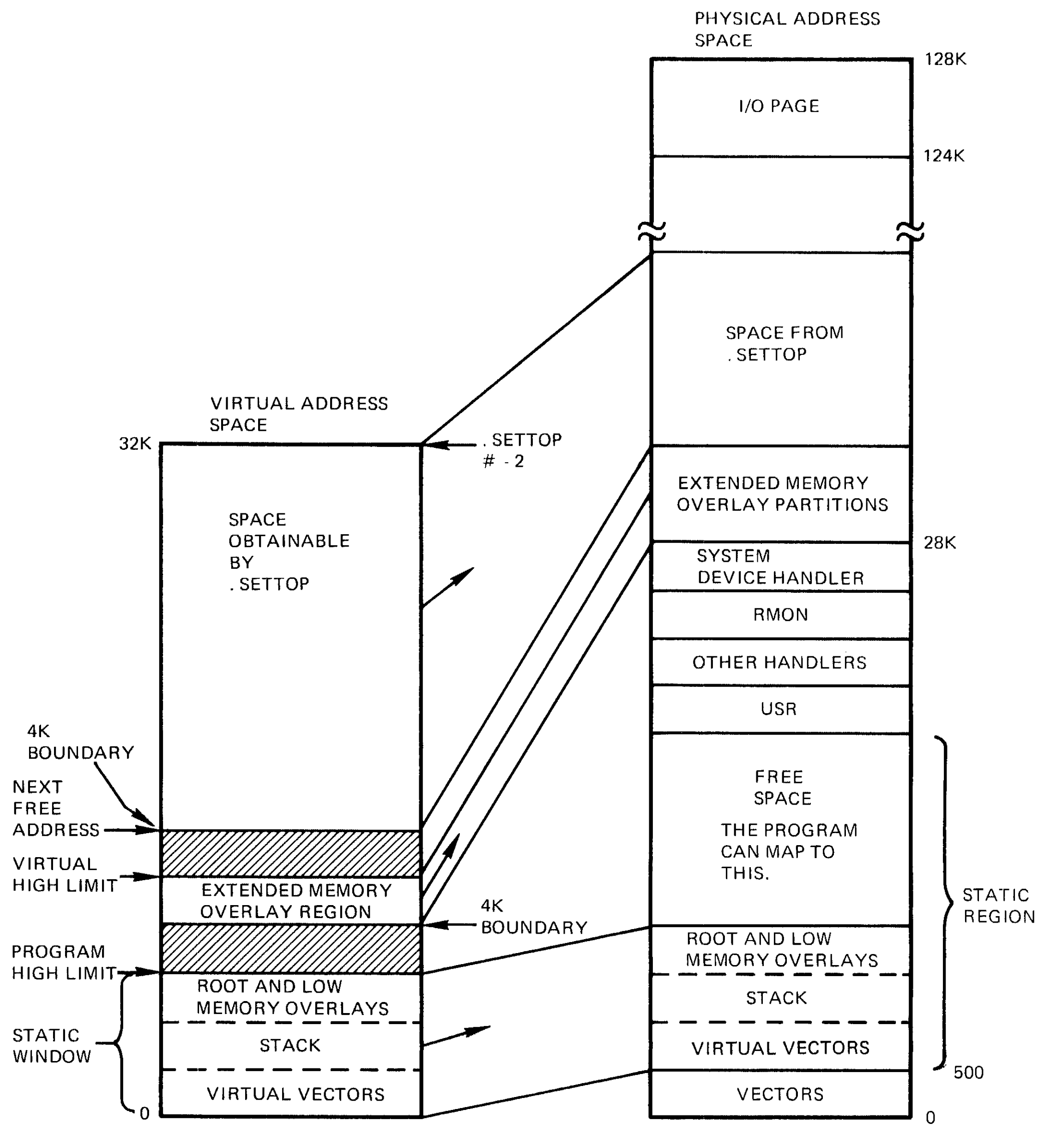
Virtual Background Jobs

Virtual background and foreground jobs are the most likely candidates for using the XM feature of the .SETTOP request. The request permits jobs to create large buffers in extended memory quickly and easily, which can help to reduce congestion in low memory. Figure 4–36 shows a virtual background job in 128K words.

Virtual Foreground Job

The .SETTOP request works in much the same way for foreground jobs as for background jobs. For a virtual foreground job without the XM .SETTOP feature, the only extra space available is the space allocated through the

Figure 4–36: Virtual Background Job



FRUN/BUFFER:n command. For a job with the XM .SETTOP feature, the /BUFFER option is ignored. (The job cannot have buffers in both low and extended memory.) Figure 4–37 shows a virtual foreground or system job with a large buffer in extended memory.

4.4.4.7 Summary of .SETTOP Action – Figures 4–38 and 4–39 and Tables 4–4 and 4–5 work together to summarize the results of all possible .SETTOP requests. In Figure 4–38, Job A is a background job whose next free address is below SYSLOW, the base of the USR. Job B is a background job whose next free address is above SYSLOW. (In the table, *next free address* is abbreviated to *NFA*.) The values in parentheses represent specific ranges for .SETTOP arguments.

Figure 4–37: Virtual Foreground or System Job

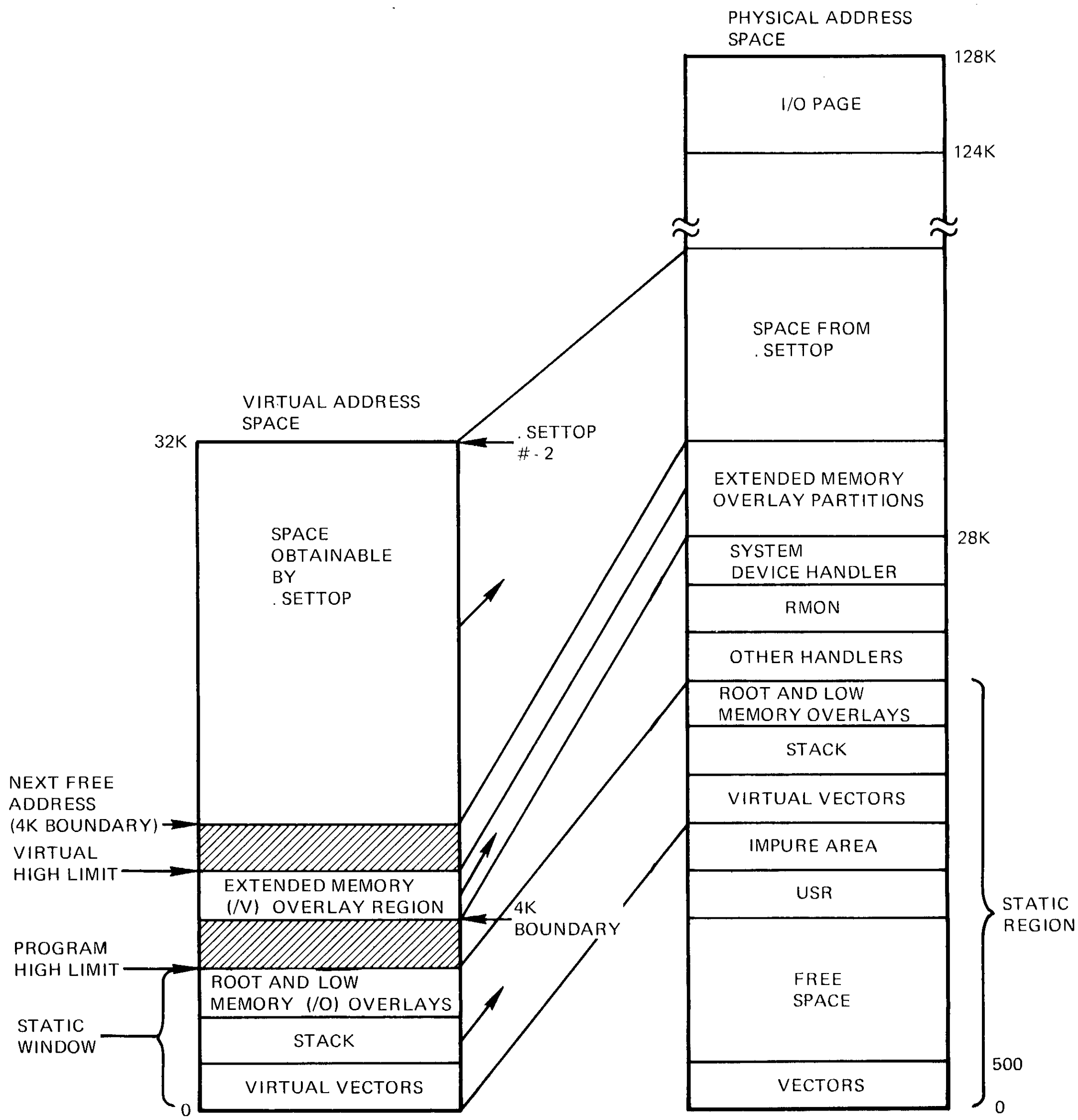


Table 4–4: Background .SETTOP Summary

.SETTOP Argument	Virtual Job Non-XM .SETTOP	Virtual Job XM .SETTOP	Privileged Job Non-XM .SETTOP	Privileged Job XM .SETTOP
High Limit for Job A After .SETTOP				
(1)	(1)	NFA – 2	(1)	NFA – 2
(2)	(2)	NFA – 2	(2)	NFA – 2
(3)	(3)	map to (3)*	(3)	(3)
(4)	SYSLOW – 2	map to (4)*	SYSLOW – 2	SYSLOW – 2
#0	0	NFA – 2	0	NFA – 2
# – 2	SYSLOW – 2	map to 32K*	SYSLOW – 2	SYSLOW – 2

(Continued on next page)

Table 4–4: Background .SETTOP Summary (Cont.)

.SETTOP Argument	Virtual Job		Privileged Job	
	Non-XM .SETTOP	XM .SETTOP	Non-XM .SETTOP	XM .SETTOP
High Limit for Job B After .SETTOP				
(1)	(1)	NFA – 2	(1)	NFA – 2
(2)	(2)	NFA – 2	(2)	NFA – 2
(3)	SYSLOW - 2	NFA – 2	SYSLOW – 2	NFA – 2
(4)	SYSLOW - 2	map to (4)*	SYSLOW – 2	NFA – 2
#0	0	NFA – 2	0	NFA – 2
# – 2	SYSLOW - 2	map to 32K*	SYSLOW - 2	NFA – 2

*If available; otherwise, as much extended memory as possible is obtained for the .SETTOP region.

Figure 4–38: Background .SETTOP Summary

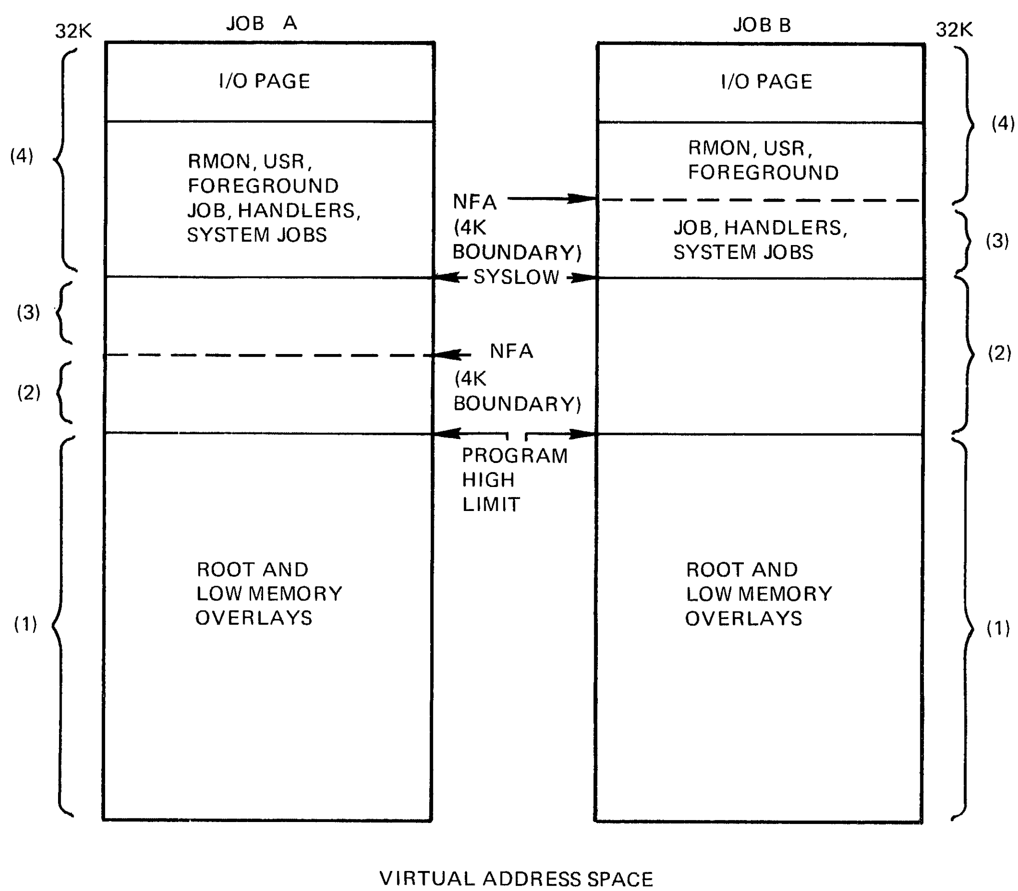


Figure 4–39: Foreground .SETTOP Summary

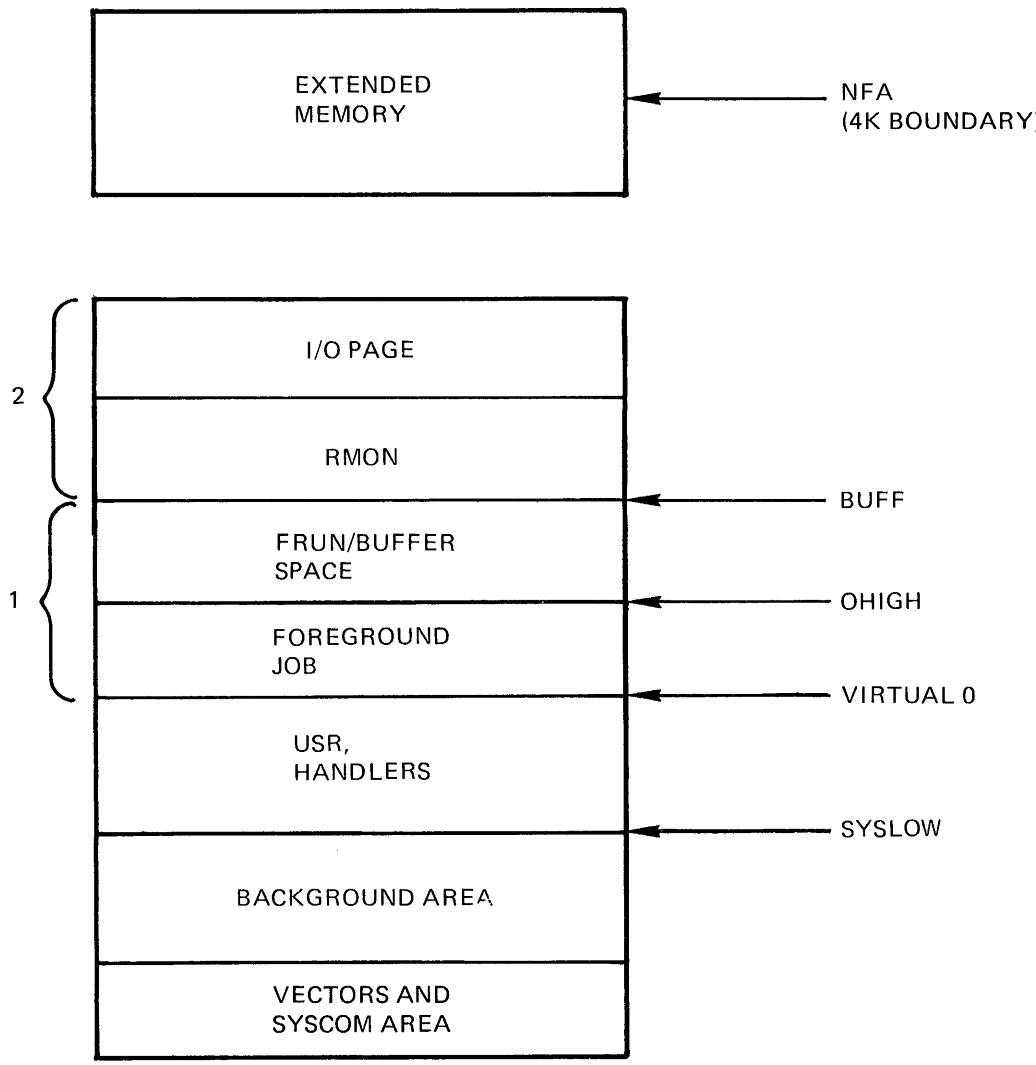


Table 4–5: Summary of Foreground Job High Limit After .SETTOP

.SETTOP Argument	Virtual Job	
	Non-XM .SETTOP	XM .SETTOP
(1)	(1)	NFA – 2
(2)	greater of OHIGH or BUFF	NFA – 2
#0	0	NFA – 2
# – 2	greater of OHIGH or BUFF	Map to 32K

4.4.5 Plan Your Own Application

When you plan your own extended memory application, decide first whether the semi-automatic ways of using extended memory are useful to you. If the XM .SETTOP feature is all you need, your program will be fairly simple to write. Similarly, if you can easily segment your program into overlays, using the extended memory (/V) overlay feature of the linker may be simple for you. If you decide to handle the mapping yourself in a MACRO–11 program, sketch out diagrams ahead of time showing the arrangements of the system components, handlers, and other jobs. Unless your job needs to access the monitor routines or the I/O page, make it a virtual job. Think about the number of windows and regions you need and design the program accordingly. The following sections provide detailed information about the programmed requests and macro calls that a MACRO–11 program in extended memory can use, as well as information about extended memory restrictions.

4.5 Introduction to the Extended Memory Programmed Requests

It is not difficult to access extended memory in a MACRO–11 program through the programmed requests, once you understand the general procedures you must follow and the tools RT–11 provides. Essentially, if your program does its own management of extended memory (rather than relying on any of the semi-automatic means described in the previous section), you must first establish window and region definition blocks. Next, you must specify the amount of physical memory the program requires, and describe the virtual addresses you plan to use. Do this by creating regions and windows. Then, associate virtual addresses with physical locations by mapping the windows to the regions. You can then remap a window to another region or part of a region. You can also eliminate a window or a region. In any case, once the initial data structures are set up, you can manipulate the mapping of windows to regions to suit your needs.

Table 4–6 summarizes the actions a program that uses extended memory may need to take. It also lists the appropriate procedures for the program to follow. Familiarize yourself with the procedures and the corresponding programmed requests and macro calls. The *RT–11 Programmer's Reference Manual* provides detailed information on the format of each programmed request and macro call. Study this information before you attempt to write an extended memory program.

4.6 Extended Memory Data Structures

A program in an extended memory environment communicates with the monitor through special data structures. For each region it defines, a program contains one *region definition block* to describe the size of the extended memory region. The monitor also maintains a set of internal data structures. The *region control block*, located in the job's impure area, describes a region. The monitor can maintain up to four region control blocks per job.

Table 4–6: Summary of Activities for a Program in an Extended Memory System

Activity	Procedure to Follow
Define offsets and symbols for a region definition block.	Use the .RDBDF or .RDBBK macro.
Set up a region definition block and specify the region size.	Use the .RDBBK macro.
Create a region.	Use the .CRRG programmed request.
Confirm the status of the new region.	Examine the contents of the region definition block after you use the .CRRG request to create the region. (Check the status bits in the status word.)
Define offsets and symbols for a window definition block.	Use the .WDBDF or .WDBBK macro.
Set up a window definition block and describe the window.	Use the .WDBBK macro.
Create a window.	Use the .CRAW programmed request.
Confirm the status of the new window.	Examine the contents of the window definition block after you use the .CRAW request to create the window. (Check the status bits in the status word.)
Associate a window with a particular region as preparation for mapping the window.	Move the region identification from R.GID in the region definition block to W.NRID in the window definition block.
Map a window to a region (explicitly).	Use the .MAP programmed request.
Map a window to a region (implicitly).	Set WS.MAP in the window definition block and load W.NRID before you issue the .CRAW request to create the window. This procedure creates the window and then maps it to a region.
Obtain the current mapping status of a particular window.	Use the .GMCX programmed request.
Unmap a window (explicitly).	Use the .UNMAP programmed request.
Unmap a window (implicitly).	Use the .MAP programmed request to map the window elsewhere. You can also unmap a window by eliminating the region to which it is mapped, or by eliminating the window itself.
Eliminate a window.	Use the .ELAW programmed request.
Eliminate a region.	Use the .ELRG programmed request.

For each window it defines, a program also uses one *window definition block* to describe the virtual addresses encompassed by that window. The *window control block*, located in the job's impure area, is the monitor's internal description for a window. The monitor can maintain up to eight window control blocks. The *I/O queue element* contains extra information in an extended memory system. Finally, the monitor allocates regions in extended memory based on its internal *free memory list.*

The following sections describe these data structures and show, where necessary, how to create them.

4.6.1 Region Definition Block

A *region definition block* is a three-word area in your program that contains information about a region you define in extended memory. The monitor uses the region definition block to communicate with your job when you issue a .CRRG or .ELRG programmed request. You must set up the region definition block in your program and define its symbolic offsets before you can create a region in extended memory. You must then place the region's size in the region definition block. After you create the region, the monitor returns its identification and some status information to you through the region definition block. Each time your program needs to refer to this region, it uses the region identification. (Since the monitor creates the static region for you, you do not know its identification. You can always refer to the static region by using 0 as its identification.) Figure 4–40 and Table 4–7 show the structure of a region definition block.

Figure 4–40: Region Definition Block

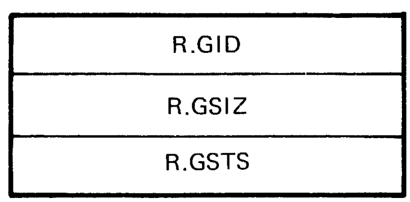


Table 4–7: Region Definition Block

Byte Offset	Symbol	Modifier	Contents
0	R.GID	Monitor's .CRRG routine	A unique region identification. Use it later to reference this region. The region identification is actually a pointer within the job's impure area to the region control block. The identification for the static region in a virtual job is 0.
2	R.GSIZ	.RDBBK macro or user program	The size of the region you need, in 32- word decimal units.
4	R.GSTS	Monitor's .CRRG routine	The region status word.

4.6.1.1 Region Status Word — The region status word contains information on the status of a region. Table 4–8 shows the bits in the region status word and their meaning. Bits 0 through 12 are reserved for future use by DIGITAL.

Table 4–8: Region Status Word

Bit	Name	Bit Pattern	Meaning When Set
15	RS.CRR	100000	The monitor created this region successfully. The .CRRG routine sets this bit; the .ELRG routine clears it.
14	RS.UNM	40000	One or more windows were unmapped as a result of eliminating this region. The .ELRG routine sets this bit when necessary.
13	RS.NAL	20000	Not currently used, but reserved.

4.6.1.2 .RDBDF Macro — Use the .RDBDF macro to define symbols for the region definition block (see the description of .RDBBK in Section 4.6.1.3). It defines the symbolic offset names for the region definition block and the names for the region status word bit patterns. In addition, this macro defines the length of the region definition block by setting up the following symbol:

```
R.GLGH = 6
```

Note that this macro does not reserve space for the region definition block.

The format of the .RDBDF macro is as follows:

.RDBDF

The .RDBDF macro expands as follows:

```
R.GID  = 0
R.GSIZ = 2
R.GSTS = 4
R.GLGH = 6
RS.CRR = 100000
RS.UNM = 40000
RS.NAL = 20000
```

4.6.1.3 .RDBBK Macro — The .RDBBK macro (like the .RDBDF macro) defines symbols for the region definition block. This macro also actually reserves space for it (unlike the .RDBDF macro). You specify as the argument to this macro the size of the region you need. If you use .RDBBK you need not use .RDBDF, since .RDBBK automatically invokes .RDBDF.

The format of the .RDBBK macro is as follows:

.RDBBK rgsiz

rgis is the size of the dynamic region, expressed in 32-word decimal units.

The following example uses the .RDBBK macro to create a region definition block for a region 4K words in size. (4K words is equivalent to 200 32-word units.) Then it creates the region.

```
RGADR:    .RDBBK   #200
          .CRRG    #ARGBLK,#RGADR     ;CREATE REGION
```

See Section 4.10 for an example program that uses .RDBBK.

4.6.2 Region Control Block

A region control block is a three-word area in your job's impure area whose contents are maintained by the monitor. A virtual job dedicates one region control block to the static region. For a privileged job, one region control block is reserved by the monitor and cannot be used by a program. Thus, all jobs can have up to three dynamic regions whose status is maintained by the monitor in the region control blocks.

Figure 4–41 and Table 4–9 show the structure of a region control block. The .ELRG programmed request clears all its fields.

Figure 4–41: Region Control Block

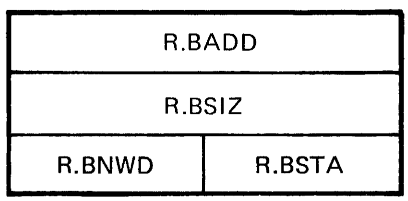


Table 4–9: Region Control Block

Byte Offset	Symbol	Modifier	Contents
0	R.BADD	Monitor's .CRRG routine	The starting address of the region, expressed in 32-word units.
2	R.BSIZ	Monitor's .CRRG routine	The size of the region in 32-word units. If this word is 0, this region control block is free.
4	R.BSTA	The monitor at run time; the monitor's .CRRG routine clears this byte	This byte is always clear unless the region was created by an XM .SETTOP. The monitor then sets bit 1, called R.STOP
5	R.BNWD	Monitor's .CRRG routine clears this byte;. MAP increments it; .UNMAP decrements it	The number of windows currently mapped to this region.

4.6.3 Window Definition Block

A window definition block is a seven-word area in your program that contains information about a virtual address window you define. The monitor uses the window definition block to communicate with your program when you issue a .CRAW, .ELAW, .GMCX, or .MAP programmed request. You must set up the window definition block in your program and define its symbolic offset names before you can create a virtual address window. You must then place a description of the window you need in the window definition block. After you create the window, the monitor returns its identification and some status information to you through the window definition block. Figure 4–42 and Table 4–10 show the structure of a window definition block.

Figure 4–42: Window Definition Block

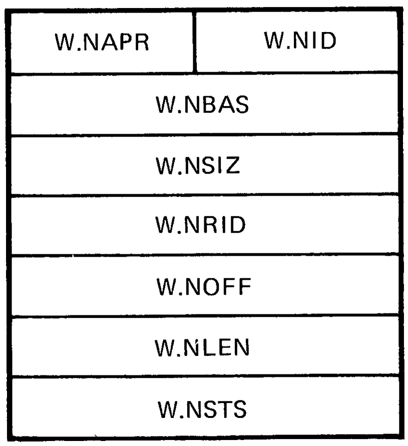


Table 4–10: Window Definition Block

Byte Offset	Symbol	Modifier	Contents
0	W.NID	Monitor's .CRAW routine	A unique window identification. Remember that you can always refer to the static window by using 0 as its identification.
1	W.NAPR	.WDBBK macro; monitor's .GMCX routine	The number of the Active Page Register that includes the window's base address. Remember that a window must start on a 4K- word boundary. See Table 4–11 for the correspondence between Active Page Registers and virtual addresses. For privileged jobs, the valid range of values is from 0 to 7. For virtual jobs, the new window must not overlap the static window. You can find the lowest valid value for W.NAPR by issuing a .GMCX request for the static window, converting the high virtual address to an APR value, and incrementing it.

(Continued on next page)

Table 4–10: Window Definition Block (Cont.)

Byte Offset	Symbol	Modifier	Contents
2	W.NBAS	Monitor's .CRAW and .GMCX routines	The base virtual address of this window. This value should indicate the same address as W.NAPR. It is provided as a validity check. Note that it is expressed as an octal address, not in 32-word decimal units.
4	W.NSIZ	.WDBBK macro; monitor's .GMCX routine	The size of this window, expressed in 32-word units.
6	W.NRID	.WDBBK macro; monitor's .GMCX routine	Identification of the region to which this window maps. The .GMCX request returns a 0 if the window is not mapped. Otherwise, it returns the identification of the region to which it is mapped. Note that the value is also 0 if the window is mapped to the static region.
10	W.NOFF	.WDBBK macro; monitor's .GMCX routine	The offset, expressed in 32-word decimal units, into the region at which to start mapping this window. The .GMCX request clears this word if the window is not mapped; otherwise, it puts the offset value here.
12	W.NLEN	.WDBBK macro; monitor's .MAP and .GMCX routines.	The amount of this window to map, expressed in 32-word units. If you put 0 here (or .CRAW with WS.MAP set), .MAP maps as much of the window as possible. On successful completion of the mapping operation, .MAP puts the actual length it mapped in W.NLEN. If you put a value here (other than 0), .MAP does not change it. The .GMCX request clears this word if the window is not mapped; otherwise, it puts the actual length mapped here.
14	W.NSTS	.WDBBK macro; monitor's .CRAW, .ELAW, and .GMCX routines	The window status word. The .GMCX request clears this word if the window is not mapped; otherwise, it sets WS.MAP to 1.

4.6.3.1 Window Status Word — The window status word serves a dual purpose. First, it allows the .CRAW request to create a window and map it to a region in one step when you put a value of 1 in bit 8. Second, the window status word allows the monitor to communicate status information to your program. Table 4–12 shows the bits in the window status word and their meaning. Bits 0 through 7 and 9 through 12 are reserved for future use by DIGITAL.

Table 4–11: Correspondence Between Active Page Registers and Virtual Addresses

Virtual Address Range	Active Page Register Number
0–17776	0
20000–37776	1
40000–57776	2
60000–77776	3
100000–117776	4
120000–137776	5
140000–157776	6
160000–177776	7

Table 4–12: Window Status Word

Bit	Name	Bit Pattern	Meaning When Set
8	WS.MAP	400	The .CRAW request should also map the new window in addition to creating it. Set this bit in the window definition block by specifying it in the .WDBBK macro. Be sure to load W.NRID before using .CRAW.
13	WS.ELW	20000	The monitor eliminated one or more windows as a result of the current operation. The .CRAW and .ELAW routines can set this bit.
14	WS.UNM	40000	The monitor unmapped one or more windows as a result of the current operation. The .CRAW and .ELAW routines can set this bit. The .MAP and .UNMAP routines set or clear this bit, as required.
15	WS.CRW	100000	The monitor created this window successfully. The .CRAW routine sets this bit; the .ELAW routine clears it.

4.6.3.2 .WDBDF Macro — Use the .WDBDF macro to define symbols for the window definition block (see the description of .WDBBK in Section 4.6.3.3). It defines the symbolic offset names for the window definition block and the names for the window status word bit patterns. In addition, this macro also defines the length of the window definition block by setting up the following symbol:

```
W.NLGH = 16
```

Note that the .WDBDF macro does not reserve any space for the window definition block.

The format of the .WDBDF macro is as follows:

.WDBDF

The .WDBDF macro expands as follows:

```
W.NID  = 0
W.NAPR = 1
W.NBAS = 2
W.NSIZ = 4
W.NRID = 6
W.NOFF = 10
W.NLEN = 12
W.NSTS = 14
W.NLGH = 16
WS.CRW = 100000
WS.UNM = 40000
WS.ELW = 20000
WS.MAP = 400
```

4.6.3.3 .WDBBK Macro – The .WDBBK macro (like the .WDBDF macro) defines symbols for the window definition block. This macro also actually reserves space for it (unlike the .WDBDF macro). The macro permits you to specify enough information about the window to simply create it. Or you can use the optional arguments to provide more information in the window definition block. The extra information allows you to create a window and map it to a region by issuing just the .CRAW programmed request. If you use .WDBBK you need not use .WDBDF, since .WDBBK automatically invokes .WDBDF.

The format of the .WDBBK macro is as follows:

.WDBBK   wnapr,wnsiz[,wnrid,wnoff,wnlen,wnsts]

wnapr is the number of the Active Page Register set that includes the window's base address. Remember that a window must start on a 4K-word boundary. See Table 4–11 for the correspondence between Active Page Registers and virtual addresses. The valid range of values is from 0 through 7.

wnsiz is the size of this window. Express it in 32-word decimal units.

wnrid is the identification for the region to which this window maps. This argument is optional. It is usually filled in at run time, rather than at assembly time.

wnoff is the offset into the region at which to start mapping this window. Express it in 32-word decimal units. This argument is optional; supply it if you need to map this window. The default is 0, which means that the window starts mapping at the region's base address.

wnlen is the amount of this window to map. Express it in 32-word decimal units. This argument is optional; supply it if you need to map this window. The default value is 0, which maps as much of the window as possible.

wnsts is the window status word. This argument is optional; supply it if you need to map this window when you issue the .CRAW request. Set bit 8, called WS.MAP, to cause .CRAW to perform an implied mapping operation.

The example in Figure 4–43 uses the .WDBBK macro to create a window definition block. First it establishes a convention for expressing K-words in units of 32 decimal words each. Then it defines the window definition block, creates the window, and maps the window to a region.

The macro call sets up a window definition block for a window that is 2K words long. The window begins at address 120000, so Active Page Register set 5 controls its mapping. The .CRAW request to create this window will also map it to an area in extended memory. The window will map to the region starting 2K words from the beginning of the region, and the .CRAW request will map as much of the window as possible. Note that the program must move the region identification into this block to select the correct region before it issues the .CRAW request.

Figure 4–43: .WDBBK Macro Example

```
        .MCALL  .WDBBK, .RDBBK, .CRRG, .CRAW, .EXIT

        KMMU=   1024./32.          ;1K in 32-word units

START:  .CRRG    #AREA,#RGADR      ;Create a region

        ;        .
        ;        .
        ;        .

        MOV      RGADR+R.GID,WNADR+W.NRID ;Move region
                                   ;ID to window definition
                                   ;block
       .CRAW     #AREA,#WNADR      ;Create a window and map it

        ;        .
        ;        .
        ;        .

        .EXIT                      ;Exit program

RGADR:  .RDBBK   2*KMMU            ;Region definition block

WNADR:  .WDBBK   5,2*KMMU,2*KMMU,0,WS.MAP   ;Window
                                   ;definition block
AREA:   .BLKW    2                 ;EMT area

        .END     START
```

4.6.4 Window Control Block

The window control block is a seven-word area in your job's impure area whose contents are maintained by the monitor. A virtual job dedicates one window control block to the static window. For a privileged job, one window control block is reserved by the monitor and cannot be used by a program. Thus, all jobs can have up to seven dynamic windows whose status is maintained by the monitor in the window control blocks. Figure 4–44 and Table 4–13 show the structure of a window control block.

Figure 4–44: Window Control Block

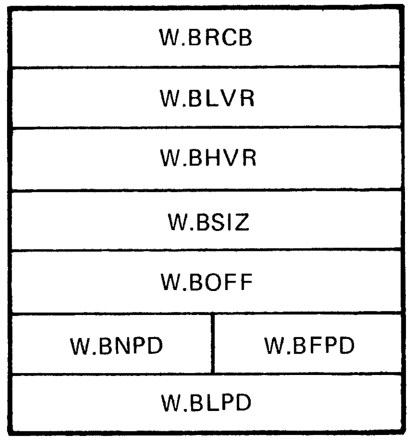


Table 4–13: Window Control Block

Byte Offset	Symbol	Modifier	Contents
0	W.BRCB	Monitor's .MAP routine; the .UNMAP request clears it	A pointer to the region control block of the region to which this window is mapped. If the value is 0, the window is not mapped.
2	W.BLVR	Monitor's .CRAW routine	The window's low virtual address limit.
4	W.BHVR	Monitor's .MAP routine	The window's high virtual address limit.
6	W.BSIZ	Monitor's .CRAW routine; the .ELAW request clears it	The window's size, in 32-word decimal units. If the value is 0, this window control block is free.
10	W.BOFF	Monitor's .MAP routine	The offset into the region at which this window begins to map, in 32-word decimal units.
12	W.BFPD	Monitor's .CRAW routine	The low byte of the address of the first Page Descriptor Register that affects this window.
13	W.BNPD	Monitor's .MAP routine	The number of Page Descriptor Registers that affect this window.
14	W.BLPD	Monitor's .MAP routine	The contents of the last Page Descriptor Register that affects this window.

4.6.5 I/O Queue Element

The I/O queue element in an extended memory system is ten words long, rather than seven words long as it is in FB and SJ systems. Section 7.9.3 describes the XM I/O queue element in detail.

4.6.6 Free Memory List

The monitor maintains a data structure called the free memory list, which it uses to allocate areas of extended memory. The list consists of a table of 10 decimal doublewords. The address of the top of the table is $XMSIZ, and the table is located in p-sect XMSUBS. The high-order word of each word pair indicates the size of an available area in extended memory, expressed as a number of 32-word decimal units. The low-order word of the pair contains the address of the area, divided by 100 octal. A value of –1 ends the table.

At bootstrap time, the table contains only one entry. The high-order word of the pair contains the total amount of extended memory. The low-order word contains the value 1600. When a job requests an extended memory region, the monitor searches through the table for an area large enough to meet the request. It returns the area in extended memory that meets the size requirement and has the lowest starting address. The monitor reduces the amount of memory in the first doubleword of the free memory list, and adjusts its starting address.

The other nine words of the free memory list are used when jobs return areas of extended memory to the available pool. In a very active system, the extended memory area can become quite fragmented.

4.7 Flow of Control Within Each Programmed Request

This section summarizes the activities that take place internally for each programmed request your program can issue. Consult the *RT–11 Programmer's Reference Manual* for the detailed syntax of each request.

4.7.1 Creating a Region: .CRRG

Issue the .CRRG programmed request to create a region in physical address space.

The monitor's .CRRG routine first checks R.GSIZ in the region definition block to make sure that you have requested a region with a valid size. (The size must be between 1 and 96K.) If the size is invalid, the request returns with error code 10 in byte 52.

Next, the routine looks for a free region control block. The request returns with error code 6 in byte 52 if no region control blocks are free.

The routine attempts to allocate the appropriate amount of memory for the region, based on the amount you specified in the programmed request. To get the most memory possible, ask for 96K words. The routine scans the free memory list for a region with the correct size. The request returns with error code 7 in byte 52 if it cannot allocate a region with the size you requested. In addition, R0 contains the largest amount of memory available. Issue the .CRRG request again for this amount of memory. If this second request fails, it means that some other job in the system just acquired some of the memory. Continue to reissue the .CRRG request with the new value from R0 until you finally obtain a region.

The request succeeds when the monitor allocates the region. The routine puts the region identification into R.GID in the region definition block. It sets RS.CRR in the region status word; it clears R.BSTA and R.BNWD in the region control block, and it puts values into R.BADD and R.BSIZ, which are also located in the region control block. The memory obtained is then removed from the monitor's free memory list and reserved for your job.

4.7.2 Creating a Window: .CRAW

Issue the .CRAW programmed request to create a virtual address window.

First, the monitor's .CRAW routine checks W.NAPR in the window definition block for a valid value. The request returns with error code 0 in byte 52 if the number of the Active Page Register set is invalid for any reason.

Next, the routine shifts W.NAPR to set up the window's base address in W.NBAS, which is also located in the window definition block.

The routine then checks W.NSIZ in the window definition block to make sure that you requested a valid size for the window (the window cannot exceed the 32K-word boundary). If there is any problem with the size, the request returns with error code 0 in byte 52.

The routine clears bits WS.ELW, WS.UNM, and WS.CRW in the window status word.

The next check is to see if the new window will overlap with an existing window. If the job is a virtual job and the new window overlaps with the static window, the request returns with error code 0. In all other situations where the new window overlaps an existing window, the routine eliminates the existing window. If the existing window is mapped, the routine unmaps it. The .CRAW routine sets WS.ELW in the window status word if it eliminates a window to create the new one. It sets WS.UNM if it also unmaps a window as it eliminates it.

Next, the routine looks for an available window control block. The request returns with error code 1 if there are no free window control blocks.

The request succeeds when the monitor modifies the appropriate data structures. It puts values in W.BSIZ, W.BLVR, and W.BFPD in the window control block; it puts the window identification in W.NID in the window definition block, and it sets WS.CRW in the window status word.

If WS.MAP in the window status word was set when you issued the .CRAW request, the routine now maps the window to the region whose identification is stored in the window definition block. To do this, the routine follows the steps outlined in the .MAP programmed request.

4.7.3 Mapping a Window to a Region: .MAP

Issue the .MAP programmed request to map a virtual address window to a physical address region. The window definition block must contain the identification of the region to which the window will map.

First, the monitor's .MAP routine finds the window control block that corresponds to the window you specify in the request. It checks W.NID to do this, and returns with error code 3 if the value is 0 or not valid.

Next, the routine finds the region control block for the region to which this window will map. The request returns with error code 2 if the region identification is invalid for any reason.

The routine looks at the offset into the region at which the window is to begin mapping. This value is contained in W.NOFF in the window definition block. If the offset is beyond the end of the region, the request returns with error code 4.

The routine checks the length of the window it is to map. This value is contained in W.NLEN in the window definition block. If the value is 0, the routine picks up the size of the region from the offset value to the end of the region. If this amount of memory is bigger than the window, the routine reduces the amount until it equals the window size, which it stores in W.NLEN. Note that if you put 0 into W.NLEN, the value that is there after the .MAP request executes is not 0, but is instead the actual length of the window that was mapped.

If the value of W.NLEN is not 0 at the start of the .MAP routine, it indicates the explicit length of the window to map. If the value is larger than the window size, or if the window would extend beyond the bounds of the region, the request returns with error code 4.

The routine increments R.BNWD in the region control block, which maintains a count of the number of windows mapped to this region.

If this window is already mapped elsewhere, this routine unmaps it and sets WS.UNM in the window status word; otherwise, this routine clears WS.UNM.

The routine next loads the user mode Active Page Register set with the correct values to map this window to this region.

Finally, the routine updates the window control block values W.BRCB, W.BHVR, W.BOFF, W.BNPD, and W.BLPD.

4.7.4 Getting the Mapping Status: .GMCX

Issue the .GMCX programmed request to obtain the current mapping status of a particular virtual address window.

First, the .GMCX monitor routine looks at the corresponding window control block for this window. If you specify a window whose identification is 0, you obtain the status of the static window for a virtual job. There is no window with the identification of 0 in a privileged job. If there is any problem with the window, the request returns with error code 3.

The routine sets W.NAPR in the window definition block to be equal to the top three bits of W.BLVR in the window control block. This sets up the starting Active Page Register set number.

Next, the routine puts values into W.NBAS, W.NSIZ, and W.NRID in the window definition block.

If the window is not currently mapped, the routine clears W.NOFF, W.NLEN, and W.NSTS in the window definition block. If the window is mapped, the routine puts the offset into the region in W.NOFF, puts the length of the window in W.NLEN, and sets the bit WS.MAP in the window status word.

4.7.5 Unmapping a Window: .UNMAP

Issue the .UNMAP programmed request to explicitly unmap a window from a region.

First, the monitor's .UNMAP routine finds the appropriate window control block. It checks W.NID in the window definition block. If the value is 0, or if it is invalid for any reason, the request returns with error code 3. If the window is not currrently mapped, the request returns with error code 5.

To unmap the window, the routine modifies the appropriate data structures. It clears W.BRCB in the window control block, and decrements R.BNWD in the region control block.

If the job is virtual, the routine clears the Page Descriptor Registers that correspond to this window so that your program can no longer reference the virtual addresses in this window.

If the job is privileged, the monitor copies the kernel Page Descriptor Register values into the user Page Descriptor Registers so that the mapping defaults to that of kernel mode.

Finally, the routine sets WS.UNM in the window status word.

4.7.6 Eliminating a Region: .ELRG

Issue the .ELRG programmed request to eliminate a physical address region.

First, the monitor's .ELRG routine checks to see if the region identification you specified is 0. In a virtual job, a region identification of 0 indicates the static region, which you cannot eliminate. In a privileged job, there is no region whose identification is 0. In either case, the request returns with error code 2.

Next, the routine looks for the corresponding region control block for this region. If the region identification is invalid for any reason, the request returns with error code 2.

Then, the routine clears RS.CRR and RS.UNM in the region status word. If there are any windows mapped to this region, the routine unmaps them and sets RS.UNM.

The routine deallocates the region by returning its physical address space to the monitor's list of free memory.

Finally, the routine clears the region control block.

4.7.7 Eliminating a Window: .ELAW

Issue this programmed request to eliminate a virtual address window.

As with the .ELRG request, the .ELAW routine first finds the corresponding window control block for this window. It checks W.NID in the window definition block. If the window identification is 0, or is not valid for any reason, the request returns with error code 3.

The routine next clears WS.CRW and WS.UNM in the window status word. If the window was mapped, the routine unmaps it and sets WS.UNM. The routine eliminates the window by clearing W.BSIZ in the window control block. Finally, the routine sets WS.ELW in the window status word.

4.7.8 Summary of Extended Memory Programmed Request Error Codes

Table 4–14 summarizes the error codes that the extended memory programmed requests can put into byte 52. Table 4–15 shows which error codes each programmed request can use.

Table 4–14: Extended Memory Programmed Request Error Codes and Meanings

Byte 52 Code	Meaning
0	There is a problem with the window ID. The window is too large, the value of W.NAPR is greater than 7, or you specified it incorrectly.
1	You tried to create more than seven windows in your program. Remember that the static window is always defined for a virtual job, and one window is always reserved by the monitor in a privileged job. You can either unmap another window and then try to create a window, or you can redefine your virtual address space into fewer windows.
2	The region identification was invalid for some reason.
3	The window identification was invalid for some reason.
4	The combination of the offset into the region and the size of the window to map to the region is invalid.
5	The window you specified was not currently mapped.

(Continued on next page)

Table 4–14: Extended Memory Programmed Request Error Codes and Meanings (Cont.)

Byte 52 Code	Meaning
6	You tried to create more than three regions in your program. Remember that the static region is always defined for a virtual job, and one region is always reserved by the monitor in a privileged job. You can eliminate another region and then try to create a new one, or you can redefine your physical address space into fewer regions. Note that extended memory overlays and XM .SETTOP account for one region each.
7	There is not enough memory available to create a region as large as the one you requested. The routine returns the size of the largest available region in R0, but does not create it.
10	You specified an invalid size for a region. A value of 0 or a value greater than 96K words is invalid.

Table 4–15: Summary of Error Codes

Programmed Request	Error Code 0	1	2	3	4	5	6	7	10
.CRRG							X	X	X
.CRAW	X	X							
.MAP			X	X	X				
.GMCX				X					
.UNMAP				X		X			
.ELRG			X						
.ELAW				X					

4.8 Restrictions and Design Implications

The manner in which RT–11's support for extended memory is implemented imposes some restrictions on the ways you can use the system. The following sections outline the implications of the design of the extended memory system.

4.8.1 PAR1 Restriction

The RT–11 monitor sometimes "borrows" kernel Page Address Register 1 for its own use. For example, it uses PAR1 to map to the EMT area blocks when it processes a programmed request.

Because the monitor alters kernel PAR1, references to virtual addresses in the range 20000 through 37777 do not always access the corresponding physical addresses. To avoid problems due to the occasional remapping of the virtual addresses controlled by kernel PAR1, observe the following programming restrictions.

1. Any channel areas you allocate with the .CDFN programmed request must be entirely within the low 28K words of memory. In addition, they must not be located within the addresses 20000 through 37777.
2. Any queue elements you allocate with the .QSET programmed request must be entirely within the low 28K words of memory. In addition, they must not be located within the addresses 20000 through 37777. Remember to allow 10 decimal words per queue element.
3. Interrupt service routines must be located entirely within the low 28K words of memory. In addition, if your XM monitor has been generated without .FETCH support, they must neither reside in nor reference addresses in the range 20000 through 37777. Section 6.7 describes the factors you must take into consideration if your program includes an in-line interrupt service routine. Be sure to execute your program as a privileged job if it contains an interrupt service routine, so that it can access the monitor and the device I/O page. Section 7.9 lists the implications of the XM design restrictions on device handlers and I/O.

This aspect of RT–11's design is important for you to understand if you have a program with its own in-line interrupt service routine, if you put a data buffer for I/O in extended memory, or if you write a device handler for an XM system.

4.8.2 Programmed Requests

Some of the RT–11 programmed requests have special restrictions when you use them in an extended memory system. These requests and their restrictions are as follows:

Programmed Request	Restriction
.CDFN	The channel area you specify in this request must be entirely within the low 28K words of memory.
.QSET	The queue element space you specify must be entirely within the low 28K words of memory. In addition, you must allow 10 decimal words for each queue element.
.CNTXSW	Virtual jobs cannot use this request, since they have no need for it in an extended memory system.

4.8.3 PAR2 Restriction

The MQ message handler resides within the physical memory mapped by Page Address Register 2. If you use the MQ handler to send and receive messages, be sure to read Section 3.5.7. When you use the MQ handler, all the PAR1 restrictions apply as well to the virtual addresses controlled by PAR2: the addresses in the range 40000 through 57777.

4.8.4 Synchronous System Traps

A *synchronous system trap* is a software interrupt that takes place synchronously with your program's execution. For example, a TRAP instruction that a program issues is a synchronous system trap. A program that issues an illegal instruction causes a trap to 10 to occur, which is also a synchronous system trap. When a trap occurs, the PDP–11 computer pushes the current PS and PC onto the stack and loads the new PS and PC from the contents of the trap vector. Table 4–16 lists the synchronous system traps and their corresponding vectors.

Table 4–16: Synchronous System Traps and Their Vectors

Vector	Synchronous System Trap
4	Trap to 4, caused by a reference to an odd address, or by a bus time-out.
10	Trap to 10, caused by an attempt to execute a reserved instruction.
14	Breakpoint trap, usually issued by a debugging utility program such as ODT.
20	I/O trap.
34	TRAP instruction, issued by a program to change the flow of execution.
114	Memory parity trap, caused by a memory parity error.
244	FPU trap, caused by a floating point unit exception or error.
250	Memory management trap, caused by a program's attempt to reference a virtual address that is not mapped to a physical address.

In an XM system, synchronous system traps, like device interrupts, take the new PS and PC from the appropriate vector in kernel space. For example, when a program issues a BPT instruction, the new PS and PC are taken from physical locations 14 and 16. As you remember, a privileged job is initially mapped to the kernel vector area, so virtual address 14 in the program maps to physical location 14. A virtual job, on the other hand, is prevented from accessing the kernel vector area. Initially, the virtual job's virtual vector area maps to physical addresses starting at location 500, not 0. For a virtual job then, the virtual vector 14 is not in physical location 14.

For each synchronous system trap, RT–11 provides a mechanism to field the trap and provide values for the new PS and PC from the virtual vector. The following sections describe the effect of the XM environment on specific synchronous system traps.

4.8.4.1 TRAP, BPT, and IOT Instructions – When a program in an XM system issues a TRAP, BPT, or IOT instruction, execution switches to the processor's kernel mode. The hardware picks up the contents of the appropriate vector (see Table 4–16) from kernel space. However, rather than dispatching immediately to the trap handling routine specified in the kernel vector, the monitor replaces the new PS and PC with values that cause execution to continue within a monitor routine. The purpose of the monitor routine is to pick up the contents of the corresponding virtual vector in user space, and then transfer control to the routine specified by the virtual PC. The kernel and user vectors for a privileged job are identical. A virtual job cannot access the kernel vectors; you can, however, put values into the virtual vectors so that the monitor will pick them up when a trap occurs. In summary, the net effect of the monitor's trap handling routine is that control is transferred to a job's specific trap routine through the contents of the job's virtual vector.

If the virtual vector contains an even, nonzero value, the monitor does not clear the vector after the first trap. This permits recursion with no effort on the part of the program.

4.8.4.2 Traps to 4 and 10, and FPU Traps — For traps to 4 and 10, and floating point unit exception traps, the monitor provides a mechanism that protects the vectors while still permitting you to use your own trap handling routines. The .TRPSET and .SFPA programmed requests permit your program to set up the addresses of trap handling routines without modifying either the kernel or the user virtual vector area. These two programmed requests function in XM systems the way they do in FB systems. Thus, you specify the address of your trap handling routine when you issue the programmed request and the monitor puts this information in the job's impure area. The monitor clears out the routine address in the impure area, so your trap handling routine should reset this area by issuing either .TRPSET or .SFPA as its last instruction before returning to the main program.

4.8.4.3 Memory Management Faults — A memory management fault occurs when a program references a virtual address that is not mapped to a physical address. If a memory management fault occurs while execution is in system state, the entire system halts. If a memory management fault occurs while execution is in user state, the monitor fields the trap through the kernel vector and provides a new PS and PC from the user virtual vector area. Once the monitor picks up the contents of a job's virtual vector, it clears the vector. If a second fault occurs and the virtual vector is 0, the monitor prints its *?MON-F-MMU fault* message and aborts the job.

To permit recursion, your program's trap handling routine must reset the contents of the memory management fault vector (at locations 250 and 252) in the job's virtual vector area. If RT–11 permitted automatic recursion, your program could loop indefinitely on a memory management fault until you halted the processor.

4.8.4.4 Memory Parity Errors — A hardware device that is an optional part of your PDP–11 computer system performs memory parity checking. You enable RT–11 support of this hardware option by selecting the memory parity special feature at system generation time. If you have memory parity hardware but do not generate a system with the memory parity checking special feature, a memory parity error causes a system halt.

For systems that support memory parity checking, the synchronous system trap procedure is similar to the procedure for memory management faults. Thus, the monitor fields the trap through the kernel vector at locations 114 and 116. It then picks up the contents of your program's virtual addresses 114 and 116, clears them, and passes control to your trap handling routine based on the new PS and PC.

If a second memory parity error occurs and the virtual vector is 0, the message *?MON-F-Mem err* prints and the job aborts. To enable recursion, your program's trap handling routine must reset the contents of the memory parity fault vector at virtual addresses 114 and 116.

4.9 Debugging an XM Application

Use VDT, the Virtual Debugging Technique, to debug virtual and privileged jobs in an XM system. VDT also handles correctly jobs in FB and SJ systems, as well as jobs in multi-terminal systems.

Use VDT.OBJ the same way you use ODT.OBJ; link it with the program you need to debug. The transfer address for VDT is O.ODT. The syntax for VDT commands is the same as the syntax for ODT. See the *RT–11 System User's Guide* for instructions on using ODT.

VDT.OBJ is created from a conditional assembly of ODT.MAC, with the conditional $VIRT equal to 1. VDT.OBJ is provided on the distribution kit; you need not assemble it yourself. VDT does not contain the interrupt service or priority routines that ODT does. Unlike ODT, which runs at priority 7 and performs its own terminal I/O, VDT runs at the same priority as your program, and uses .TTYIN and .TTYOUT programmed requests to perform terminal I/O.

Because VDT uses .TTYIN and .TTYOUT requests, you can run it from a job's console terminal; it is not limited to the hardware console interface. Since VDT alters the contents of the Job Status Word, it must save the original contents elsewhere. You can use the $J/ command to obtain the original contents of the JSW; you can also modify it there.

VDT runs in user, not in kernel mode. When you debug a virtual job with VDT, you are limited to accessing the job's area only. You cannot access the protected system areas such as the monitor, the vectors, and the I/O page. When you debug a privileged job with VDT, you have access to the same memory the job does.

4.10 Extended Memory Example Program

Figure 4–45 provides an example program that uses extended memory programmed requests. This example assumes that any necessary handlers are already loaded.

Figure 4–45: Extended Memory Example Program

```
        .TITLE  XMCOPY
;+
; THIS IS AN EXAMPLE IN THE USE OF THE RT-11 EXTENDED
; MEMORY REQUESTS. THE PROGRAM COPIES FILES AND THEN
; VERIFIES THE RESULTS. IT USES EXTENDED MEMORY TO
; IMPLEMENT 4K TRANSFER BUFFERS. THIS PROGRAM USES MOST
; OF THE EXTENDED MEMORY PROGRAMMED REQUESTS, AND
; DEMONSTRATES OTHER PROGRAMMING TECHNIQUES.
;-
        .NLIST  BEX
        .MCALL  .UNMAP,.ELRG,.ELAW,.CRRG,.CRAW,.MAP
        .MCALL  .PRINT,.EXIT,.CLOSE,.CSIGEN,.READW,.WRITW
        .MCALL  .RDBBK,.WDBBK,.TTYOUT,.WDBDF,.RDBDF
        JSW     = 44                ;JSW LOCATION
        J.VIRT  = 2000              ;VIRTUAL JOB BIT IN JSW
        ERRBYT  = 52                ;ERROR BYTE LOCATION
        APR     = 2                 ;PAR/PDR FOR 1ST WINDOW
        APR1    = 4                 ;   "      "  2ND   "
        BUF     = WDB+W.NBAS        ;VIRTUAL ADDR OF 1ST
                                    ; BUFFER
        BUF1    = WDB1+W.NBAS       ;VIRTUAL ADDR OF 2ND
                                    ; BUFFER
        CORSIZ  = 4096.             ;SIZE OF BUFFER IN WORDS
        PAGSIZ  = CORSIZ/256.       ;PAGE SIZE IN BLOCKS
        WRNID   = WDB+W.NRID        ;REGION ID ADDR OF 1ST
                                    ; REGION
        WRNID1  = WDB1+W.NRID       ;REGION ID ADDR OF 2ND
                                    ; REGION
        .ASECT                      ;ASSEMBLE IN THE VIRTUAL
                                    ; JOB BIT
        .= JSW
        .WORD   J.VIRT              ;MAKE THIS A VIRTUAL JOB
        .PSECT                      ;START CODE NOW
        .WDBDF                      ;CREATE WINDOW DEFINITION
                                    ; BLOCK SYMBOLS
        .RDBDF                      ;CREATE REGION DEFINITION
                                    ; BLOCK SYMBOLS
START:: .CSIGEN #ENDCRE,#DEFLT,#0 ;GET FILESPECS,
                                    ; HANDLERS, OPEN FILES
        BCS     START               ;BRANCH IF ERROR
        INCB    ERRNO               ;ERR = 1X
        .CRRG   #CAREA,#RDB         ;CREATE A REGION
        BCC     10$                 ;BRANCH IF SUCCESSFUL
        JMP     ERROR               ;REPORT ERROR
                                    ; (JMP DUE TO RANGE!)
10$:    MOV     RDB,WRNID           ;MOVE REGION ID TO WINDOW
                                    ; DEFINITION BLOCK
        INCB    ERRNO               ;ERR = 2X
        .CRAW   #CAREA,#WDB         ;CREATE WINDOW...
        BCC     20$                 ;BRANCH IF NO ERROR
        JMP     ERROR               ;REPORT ERROR...
        INCB    ERRNO               ;ERR = 3X
        .MAP    #CAREA,#WDB         ;EXPLICITLY MAP WINDOW...
        BCC     30$                 ;BRANCH IF NO ERROR
        JMP     ERROR               ;REPORT ERROR
        CLR     R1                  ;R1 = RT11 BLOCK #
                                    ; FOR I/O
        MOV     #CORSIZ,R2          ;R2 = # OF WORDS TO READ
        INCB    ERRNO               ;ERR = 4X
```

(Continued on next page)

Figure 4–45: Extended Memory Example Program (Cont.)

```
READ:   .READW   #RAREA,#3,BUF,R2,R1 ;TRY TO READ 4K-WORTH
                                ; OF BLOCKS
        BCC      WRITE          ;BRANCH IF NO ERROR
        TSTB     @#ERRBYT       ;EOF?
        BEQ      PASS2          ;BRANCH IF YES
        JMP      ERROR          ;MUST BE HARD ERROR,
                                ; REPORT IT
WRITE:  MOV      R0,R2          ;R2 = SIZE OF BUFFER
                                ; JUST READ
        .WRITW   #RAREA,#0,BUF,R2,R1 ;WRITE OUT THE BUFFER
        BCC      ADDIT          ;BRANCH IF NO ERROR
        INCB     ERRNO          ;ERR = 5X
        JMP      ERROR          ;REPORT ERROR
ADDIT:  ADD      #PAGSIZ,R1     ;ADJUST BLOCK #
        BR       READ           ;THEN GO GET ANOTHER
                                ; BUFFER
PASS2:  INCB     ERRNO          ;ERR = 6X
        .CRRG    #CAREA,#RDB1   ;CREATE A REGION
        BCC      35$            ;BRANCH IF NO ERROR
        JMP      ERROR          ;REPORT ERROR
35$:    MOV      RDB1,WRNID1    ;GET REGION ID TO WINDOW
                                ; DEFINITION BLOCK
;* EXAMPLE USING THE .CRAW REQUEST DOING *
;* IMPLIED .MAP REQUEST.                 *
        INCB     ERRNO          ;ERR = 7X
        .CRAW    #CAREA,#WDB1   ;CREATE WINDOW USING
                                ; IMPLIED .MAP
        BCC      VERIFY         ;BRANCH IF NO ERROR
        JMP      ERROR          ;REPORT ERROR
VERIFY:: INCB    ERRNO          ;ERR = 8X
        CLR      R1             ;R1 = RT11 BLOCK # AGAIN
GETBLK: MOV      #CORSIZ,R2     ;R2 = 4K BUFFER SIZE
        .READW   #RAREA,#3,BUF1,R2,R1 ;TRY TO GET 4K-WORTH
                                ; OF INPUT FILE
        BCC      40$            ;BRANCH IF NO ERROR
        TSTB     @#ERRBYT       ;EOF?
        BEQ      ENDIT          ;BRANCH IF YES
        JMP      ERROR          ;REPORT HARD ERROR
40$:    MOV      R0,R2          ;R2 = SIZE OF BUFFER READ
        .READW   #RAREA,#0,BUF,R2,R1 ;TRY TO GET SAME SIZE
                                ; FROM OUTPUT FILE
        BCC      50$            ;BRANCH IF NO ERROR
        INCB     ERRNO          ;ERR = 9X
        JMP      ERROR          ;REPORT ERROR
50$:    MOV      BUF,R4         ;GET OUTPUT BUFFER ADDRESS
        MOV      BUF1,R3        ;GET INPUT BUFFER ADDRESS
70$:    CMP      (R4)+,(R3)+    ;VERIFY THAT DATA IS THE
                                ; SAME
        BNE      ERRDAT         ;IT'S NOT, REPORT ERROR
        DEC      R2             ;ARE WE FINISHED?
        BNE      70$            ;BRANCH IF WE AREN'T
        ADD      #PAGSIZ,R1     ;ADJUST BLOCK # FOR PAGE
                                ; SIZE
        BR       GETBLK         ;GO GET ANOTHER BUFFER
                                ; PAIR
ENDIT:  .PRINT   #ENDPRG        ;ANNOUNCE WE'RE FINISHED
XCLOS:  .CLOSE   #0             ;CLOSE OUTPUT FILE
        .UNMAP   #CAREA,#WDB    ;EXPLICITLY UNMAP 1ST
                                ; WINDOW
        .ELAW  #CAREA,#WDB      ;EXPLICITLY ELIMINATE 1ST
                                ; WINDOW
        .ELRG  #CAREA,#RDB      ;ELIMIMATE 1ST REGION
        .ELRG  #CAREA,#RDB1     ;UNMAP, ELIMINATE 2ND
                                ; WINDOW & REGION
        .EXIT                   ;EXIT PROGRAM
```

(Continued on next page)

Figure 4–45: Extended Memory Example Program (Cont.)

```
ERROR:  MOVB    @#ERRBYT,R0         ;MAKE ERROR BYTE CODE
                                    ; 2ND DIGIT
        ADD     #'0,R0              ;OF ERROR CODE...
        MOVB    R0,ERRNO+1          ;PUT IT IN ERROR MESSAGE
        .PRINT  #ERR                ;PRINT IT...
        BR      XCLOS               ;GO CLOSE OUTPUT FILE
ERRDAT: .PRINT  #ERRBUF             ;REPORT VERIFY FAILED...
        BR      XCLOS               ;GO CLOSE OUTPUT FILE
RDB:   .RDBBK   CORSIZ/32.          ;.RDDBK DEFINES REGION
                                    ; DEFINITION BLOCK
WDB:   .WDBBK   APR,CORSIZ/32.      ;.WDDBK DEFINES WINDOW
                                    ; DEFINITION BLOCK
RDB1:  .RDBBK   CORSIZ/32.          ;DEFINE 2ND REGION SAME
                                    ; WAY
WDB1:  .WDBBK   APR1,CORSIZ/32.,0,0,CORSIZ/32.,WS.MAP
                                    ; AND 2ND WINDOW
                                    ; (BUT WITH MAPPING
                                    ; STATUS SET!)
CAREA:  .BLKW   2                   ;EMT ARGUMENT BLOCKS
RAREA:  .BLKW   6
DEFLT:  .WORD   0,0,0,0             ;NO DEFAULT FILE TYPES
ENDPRG: .ASCIZ  / * END OF XM EXAMPLE PROGRAM */
ERR:    .ASCII  /?XM REQUEST OR I-O ERROR # /
ERRNO:  .ASCIZ  /00/
ERRBUF: .ASCIZ  /?DATA VERIFICATION ERROR?/
ENDCRE  = .                         ;FOR CSIGEN - XM
                                    ; HANDLERS LOADED !
        .END    START
```

Chapter 5
Multi-Terminal Feature

In describing the multi-terminal feature of RT–11 this chapter provides background information on the hardware and describes the data structures of a multi-terminal system. It also describes the interrupt service and polling routines, the programmed requests available to application programs, and typical situations in which you can use two terminals without making use of the multi-terminal special feature. Finally, restrictions are listed and a sample program is provided.

5.1 Components of a Multi-Terminal System

RT–11 implements support for multiple terminals as a special feature that you select at system generation time and that is available to SJ, FB, and XM monitors. Essentially, the multi-terminal feature permits an application program to control one or more terminals. It does not change RT–11's basic characteristic of being a single-user operating system. Specifically, multi-terminal support does not permit more than one terminal at a time to be the command terminal, the terminal at which you communicate with RT–11 through the keyboard monitor commands.

Support for multiple terminals is implemented through the following components:

- MTTEMT.MAC, which processes the multi-terminal programmed requests.
- MTTINT.MAC, which contains the multi-terminal interrupt service and polling routines.
- TRMTBL.MAC, which defines the multi-terminal terminal control blocks.

MTTEMT, MTTINT, and TRMTBL assemble and link together as part of the Resident Monitor for a multi-terminal system.

There are also some important data structures in multi-terminal systems:

- *Terminal control blocks*, called TCBs (one per terminal), which contain information about the terminal and the job. The TCBs also contain the input and output ring buffers for the terminal.
- *Logical unit numbers*, called LUNs, through which RT–11 refers to the terminals that are part of your system.

- *Asynchronous terminal status words*, called AST words (one per LUN), in which RT–11 maintains event flags to reflect the current status of each terminal. This word is a special feature you can select at system generation time.

5.2 Hardware Background Information

This section provides some background information that is useful if you are unfamiliar with the communication hardware RT–11 supports.

RT–11 can support both the DL series (including DL11 and DLV11, or compatible equivalent, such as the PDT–11 terminal and modem ports) and the DZ series (including DZ11 and DZV11) of serial interfaces. An interface is similar to a device controller; it stands between the computer and a serial line. The other end of the line can be connected to a terminal, a communication device, a peripheral device, or another computer.

The DL interface connects the computer system to a single serial line. Each DL interface has its own Control and Status Register (CSR) address and vector address. You can have as many as eight DL interfaces on your computer system, including the hardware console interface. Since each DL interface is a separate controller, there is no real physical unit number; 0 is assigned for consistency. Note that even though the DLV11-J module contains four serial lines, they appear to the software as four separate and distinct DL interfaces.

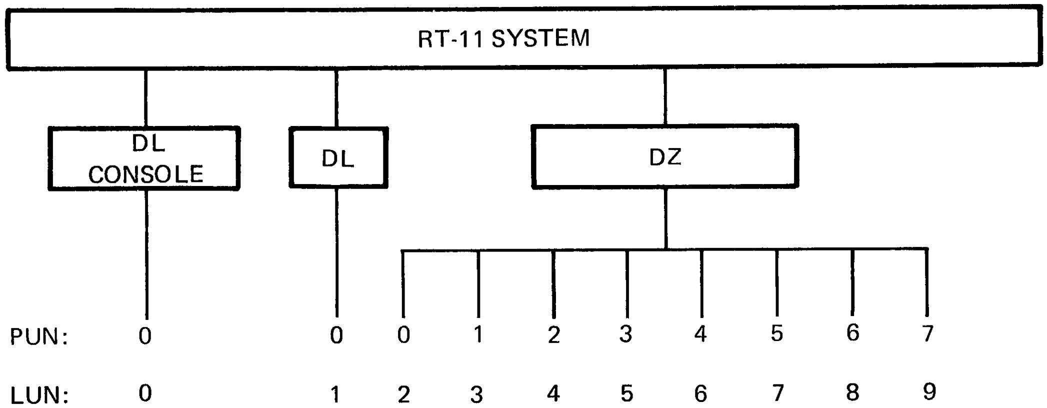
Each RT–11 system must have a hardware console interface so that the hardware can use it at bootstrap time to locate the console terminal. The hardware bootstrap on many systems requires that a terminal be connected at the standard console addresses for diagnostic purposes and for operator communication at bootstrap time. Your hardware console interface must be a local DL. Its interrupt vectors are located at 60 and 64 in low memory, and its LUN is always 0.

A DZ interface is called a *multiplexer*; it connects several serial lines through a single pair of CSR and vector addresses. The DZ11 interface connects the computer system to eight lines that have physical unit numbers from 0 through 7. The DZV11 is similar to the DZ11, but it connects the system to only four lines that have physical unit numbers from 0 through 3. You can have two DZ11 or four DZV11 interfaces, for a total of 16 additional lines.

Figure 5–1 illustrates DL and DZ interfaces and their physical and logical unit numbers.

At system generation time, you specify through the SYSGEN dialogue how many DL and DZ interfaces your target system has. You also indicate how many of their physical units are actually connected to terminals on the system. Of those terminals, you must indicate which are local and which are remote lines. Unlike physical unit numbers, which are numbered starting at 0 for each interface, the logical unit numbers that RT–11 uses are unique.

Figure 5–1: Interfaces and Physical and Logical Unit Numbers



They begin at 0 and continue until all terminals have been accounted for. SYSGEN assigns the physical unit numbers of the interfaces to its software logical unit numbers in the following order:

1. Local DL lines (the hardware console interface is always LUN 0)
2. Remote DL lines
3. Local DZ lines
4. Remote DZ lines

The order in which SYSGEN assigns physical lines to logical unit numbers is also the order in which it generates the terminal control blocks. It generates one TCB for each line you specify in the SYSGEN dialogue. The TCBs are arranged in RMON in the order in which you specify the lines to SYSGEN. There are no TCBs for any unused interface physical lines.

PDT–11 systems with cluster controllers, and PDP–11/03 and 11/23 systems with a DLV11-J interface have three additional DL interfaces at the standard addresses. The PDT–11 ports are labeled with terminal numbers that are the same as the corresponding RT–11 logical unit numbers. Some systems have SLU (serial line unit) port numbers; the RT–11 logical unit number corresponding to a port is the SLU number plus 1.

When you bootstrap a multi-terminal system, RT–11 checks for the presence of each interface for which a TCB exists by attempting to access its CSR, as specified in the SYSGEN dialogue. If the interface does not exist, the logical unit number associated with that interface is marked as nonexistent, and any attempt to attach such a LUN results in an error. The space occupied by the TCB of a nonexistent LUN is not recoverable. You can use the SHOW TERMINALS monitor command to verify that the information you supplied during system generation was correct.

Note that RT–11 does not attempt to determine whether or not a terminal or modem is actually connected to an interface line; it assumes the connection is present. For an unconnected line, no input characters can be generated; output directed to the line is sent out and lost.

5.3 What Is the Console Terminal?

A potentially confusing aspect of RT–11's multi-terminal support is its ability to change the console terminal. This section defines precisely what is meant by the terms *hardware console interface, boot-time console, background console*, and *private console*. You will avoid confusion if you familiarize yourself with these terms and use them consistently.

The *hardware console interface*, as Section 5.2 describes, is the terminal interface located at vectors 60 and 64, whose control and status registers begin at 177560 in the I/O page. This is the serial line interface the hardware bootstrap uses at bootstrap time. (Generally, you must have a terminal connected to the hardware console interface in order to bootstrap the system.) This is almost always the terminal on which RT–11 prints its startup message. Remember that the hardware console interface is always LUN 0.

The *boot-time console* is the terminal on which RT–11 prints its startup message. This is almost always the same as the terminal connected to the hardware console interface. In a system without the multi-terminal feature, the CSR for this terminal, 177560, is contained in TTKS. (TTKS is located at fixed offset 304 from the start of the Resident Monitor.) In a multi-terminal system, the CSR is located at offset T.CSR in the first TCB in the Resident Monitor.

The *background console*, also called the *command console*, is originally the same as the boot-time console. (It remains the same until you use the SET TT: CONSOL command, described below, to move the background console.) It is the terminal on which you type commands to the Keyboard Monitor, and through which you communicate with the background job. If you run a foreground job or system jobs, they can share the background console. In this case, you must use CTRL/B to communicate with the background job, CTRL/F for the foreground job, and CTRL/X for the system jobs. For example, to abort a job from a shared console, you must either type the appropriate CTRL sequence, followed by two CTRL/C characters, or use the DCL ABORT command. (See Chapter 3 for more information on control sequences.)

The programmed requests .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT interact with the background console for the background job, and also for any foreground or system jobs that happen to be sharing this terminal.

NOTE

> RT–11 ignores any unit number you specify with device TT. Therefore, references to TT:, TT0:, TT1:, and so on, are all equivalent, and default to the background console.

In a multi-terminal system you can move the background console to another terminal by issuing the SET TT: CONSOL monitor command. By specifying another logical unit number in the SET command, you can move the background console to any other local terminal in the system, except to a private console.

A *private console* is a local terminal used by a single foreground or system job. You give a job its own private console when you start the job by using the FRUN/TERMINAL:n or SRUN/TERMINAL:n commands. No other job can share a private console with the original job. A job's private console is the terminal with which its .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT programmed requests interact. In addition, any .READ or .WRITE requests to TT that this job makes access the private console. When a job has its own private console, you can no longer communicate with the job through the background console. Thus, you can no longer use CTRL/F at the background console, for example, to interact with a foreground job that has its own private console; instead, you must type on the private console. To abort this foreground job, you must type two CTRL/Cs on its private console. You cannot issue keyboard monitor commands from a private console.

You cannot change a private console to a different terminal by using the SET TT: CONSOL command; that command is valid only for the background console. This is because the Keyboard Monitor runs as a background job, and it can run only on the background console. The background console is private if there are no jobs sharing it.

A *shared console* refers to the background console unless the following conditions apply:

1. In an FB or XM system without the system job feature, the foreground job is running with a private console;
2. In an FB or XM system with the system job feature, all six system jobs and the foreground job are running, and each has a private console.

Remember that a private console can never be shared.

A *console* simply refers to a terminal being used as the background shared console, or as a foreground or system job private console.

5.4 Using Two or More Terminals

There are several situations in which you may need to use more than one terminal, but you do not need any of the special features available through the multi-terminal programmed requests. The following sections describe some of those situations and show how to arrange the terminals, often without generating support for the multi-terminal feature.

5.4.1 A Video Console Terminal and a Hard Copy Printing Terminal

A typical situation that arises in RT–11 applications is the case in which it is desirable to use a video terminal as the background console terminal and a hard copy terminal as a line printer. The next two sections describe the procedures to use, depending on whether the video terminal or the hard copy terminal is the boot-time console.

5.4.1.1 The Video Terminal Is the Boot-Time Console — If your video terminal is the boot-time console, it is simple to use a hard copy printing terminal as a line printer. (Note that the hard copy terminal must be on a DL interface to use this procedure.) You set up the vectors and CSR addresses for the hard copy terminal in the LS device handler file (by using the SET LS: commands described in the *RT–11 System User's Guide*) and install LS. You can then simply assign LP to LS and proceed to use the hard copy terminal as a line printer.

This is the simplest of multiple-terminal applications, since it does not involve system generation. This procedure is not effective, however, if the hard copy terminal is not on a local DL interface.

Under many circumstances, it may be desirable to have the hard copy terminal become the console terminal. Use the procedure described in Section 5.4.2 to do this.

5.4.1.2 The Hard Copy Terminal Is the Boot-Time Console — How you make the hard copy terminal the line printer when the hard copy terminal is the boot-time console depends on whether the video terminal is on a DL or DZ interface. If the video terminal is on a DL interface, there are four possible approaches that permit you to use the hard copy terminal as a line printer.

In *Procedure 1* you can perform a system generation (without including the multi-terminal feature) to make the video terminal appear to be the boot-time console. Note that the hard copy terminal remains the hardware console interface. That is, you must still type the name of the system device on the hard copy terminal in response to the *$* prompt. However, RT–11 does print its boot message on the video terminal. Once the system is bootstrapped, you can use the LS handler to access the hardcopy terminal as a line printer.

In *Procedure 2* you can authorize a DIGITAL Field Service representative to change your system configuration so that the video terminal is the boot-time console, and the hard copy terminal is on a local DL interface. Then you can use the procedure outlined in Section 5.4.1.1.

In *Procedure 3* you can use a special program to switch the background console to the video terminal. Except that the default boot-time console defaults to the hard copy terminal after each reboot, this is similar to procedure 1, above. You can use the LS handler to access the hard copy terminal as a line printer. Section 5.4.2 shows the program you run to use Procedure 3.

Procedure 4 is similar to Procedure 3, except that you alter the monitor image on a mass storage device instead of in memory. This procedure is useful only in systems without the multi-terminal feature. Figure 5–2 shows the patch for Procedure 4. You must supply the correct value for the vector, CSR, protection offset, and protection code (see Section 3.6.1.2) for your application.

Figure 5–2: Patch for Procedure 4

```
! Permanent modification of monitor using CSR and Vector addresses
! CSR = 175620-175626 / Vec = 310-316

.R SIPP (RET)
*monitr.SYS (RET)                                 ! monitr represents the file name
Base?       ;S (RET)                              ! of the monitor file you are
Search for?       60 (RET)                        ! changing
Start?            5100 (RET)
End?              5200 (RET)
Found at nnnnnn
Base?        nnnnnn (RET)
Offset?      (RET)
  Base                  Offset       Old     New?
nnnnnn                  000000      000060   310 (RET)     ! New vector
nnnnnn                  000002      xxxxxx   ^Z  (RET)
Offset?     6 (RET)
  Base          Offset       Old     New?
nnnnnn          000006     000064    314 (RET)     ! New vector plus 4
nnnnnn          000010     xxxxxx    ^Z  (RET)
Offset?     ^Z (RET)
Base?       $RMON(RET)                             ! Find the value of $RMON on your
Offset?     304 (RET)                              ! link map.

  Base         Offset        Old      New?
$RMON          000304       177560    175620 (RET) ! New CSR
$RMON          000306       177562    175622 (RET) ! New CSR
$RMON          000310       177564    175624 (RET) ! New CSR
$RMON          000312       177566    175626 (RET) ! New CSR
$RMON          000314       177777    ^Z (RET)
Offset?     342 (RET)                                      ! Offest for protection byte
  Base         Offset        Old      New?
$RMON          000342       000000    17 (RET)     ! Enable protection
$RMON          000344       000000    ^Y
*^C
.
```

If the video terminal is on a DZ interface, you must perform a system generation for a multi-terminal system. Specify information about your system configuration to SYSGEN exactly as it exists. Once you bootstrap the new system, set the LS vector and CSR to those of the hard copy terminal (by using the SET LS: commands described in the *RT–11 System User's Guide*). Note that this action changes the handler file on a mass storage device, and that you cannot use the hard copy terminal in any multi-terminal application. You need to modify the vector and CSR only once.

Before you use the LS handler, issue the SET TT: CONSOL command to set the background console to the video terminal. Since this setting reverts to its original state after each bootstrap, put this SET command in your startup indirect command file.

NOTE

> You must never issue the SET TT: CONSOL=0 command or access LUN 0 in any way; this is guaranteed to crash the system.

5.4.2 Switching the Console Terminal

Figure 5–3 lists a program called CONSOL that you can use to switch the console terminal to another terminal in a system without the multi-terminal special feature. Edit the source file to supply values for the CSR and vector for the new console; use the symbols CSRAD and VEC. To switch the console back and forth between two terminals, maintain two copies of the program, one for each terminal. The terminal interfaces must be DL11s; the program will not work on DZ11 interfaces.

Figure 5–3: Program to Switch the Console Terminal

```
.MAIN.  MACRO V04.00  3-JAN-80 18:44:25 PAGE 1


                          ;+
                          ;       PROGRAM TO CHANGE CONSOLE TO ONE
                          ;       OTHER THAN BOOT CONSOLE
                          ;-

                                  .MCALL  .MTPS,.PRINT,.EXIT

                175620            CSRAD   = 175620         ;*** NEW CONSOLE
                                                           ;INPUT CSR ***
                000310            VEC     = 310            ;*** NEW CONSOLE
                                                           ;VECTOR ***
                000372            SYSGEN  = 372            ;OFFSET TO SYSGEN WORD
                020000            MTTY$   = 20000          ;MULTI-TERMINAL BIT IN
                                                           ;SYSGEN WORD

                000017            BMASK   = 360/<<15.*<VEC-<20*<VEC/20>>>/8.>+1>
                000342            BITMAP  = 326+<VEC/20>

     000000     013700   PROC3:   MOV     @#54,R0          ;R0 => RMON
                000054
     000004     032760            BIT     #MTTY$,SYSGEN(R0) ;MULTI-TERMINAL SYSTEM?
                020000
                000372
     000012     001044            BNE     2$               ;YES - CAN'T USE THIS
                                                           ;TECHNIQUE!
     000014                       .MTPS   7                ;GO TO PRIORITY 7 !!!
     000014     005046   .IIF NB <7>      CLR     -(6.)
     000016     116716   .IIF NB <7>      MOVB    7,(6.)
                000007
     000022     013746            MOV     @#^054,-(6.)
                000054
     000026     062716            ADD     #^0360,(6.)
                000360
     000032     004736            JSR     7,,@(6.)+
     000034     152760            BISB    #BMASK,BITMAP(R0) ;PROTECT NEW CONSOLE
                000017
                000342
                                                           ;VECTORS
     000042     062700            ADD     #304,R0          ;R0 => CONSOLE REGISTER
                000304
                                                           ;LIST IN RMON
     000046     012701            MOV     #CSR,R1          ;R1 => NEW CSR/DATA
                000206
                                                           ;REG LIST
     000052     005070            CLR     @(R0)            ;DISABLE OLD INPUT CSR
                000000
                                                           ;INTERRUPTS
     000056     012120   1$:      MOV     (R1)+,(R0)+      ;MOVE IN NEW CSR/DATA
                                                           ;REGISTER ADDR
     000060     005711            TST     @R1              ;DONE?
     000062     100775            BMI     1$               ;IF MINUS, NO...
                                                           ;DO ANOTHER
     000064     012700            MOV     #60,R0           ;R0 = PRESENT CONSOLE
                000060
```

Figure 5–3: Program to Switch the Console Terminal (Cont.)

```
                                                              ;VECTOR
   000070  011101              MOV     @R1,R1                 ;R1 = NEW VECTOR
           000004              .REPT   4
                               MOV     (R0)+,(R1)+            ;LOAD NEW CONSOLE VECTORS
                               .ENDR
   000072  012021              MOV     (R0)+,(R1)+            ;LOAD NEW CONSOLE VECTORS
   000074  012021              MOV     (R0)+,(R1)+            ;LOAD NEW CONSOLE VECTORS
   000076  012021              MOV     (R0)+,(R1)+            ;LOAD NEW CONSOLE VECTORS
   000100  012021              MOV     (R0)+,(R1)+            ;LOAD NEW CONSOLE VECTORS
   000102                      .MTPS   0                      ;BACK TO PRIORITY 0
   000102  005046    .IIF NB <0>       CLR     -(6.)
   000104  116716    .IIF NB <0>       MOVB    0,(6.)
           000000
   000110  013746              MOV     @#^054,-(6.)
           000054
   000114  062716              ADD     #^0360,(6.)
           000360
   000120  004736              JSR     7,,@(6.)+
   000122                      .EXIT                          ;TERMINATE PROGRAM
   000122  104350              EMT     ^0350

   000124            2$:       .PRINT  #NOMT                  ;PRINT ERROR MESSAGE
   000124  012700              MOV     #NOMT,%0
           000134'
   000130  104351              EMT     ^0351
   000132                      .EXIT                          ; AND LEAVE
   000132  104350              EMT     ^0350

                               .NLIST  BEX
   000134     077    NOMT:     .ASCIZ  /?MULTI-TERMINAL SYSTEM, USE SET TT CONSOL/
                               .EVEN

   000206  175620    CSR:      .WORD   CSRAD                  ;CSR/DATA BUFFER/VECTOR LIST
   000210  175622              .WORD   CSRAD+2
   000212  175624              .WORD   CSRAD+4
   000214  175626              .WORD   CSRAD+6
   000216  000310              .WORD   VEC
           000000'             .END    PROC3


SYMBOL TABLE

BITMAP= 000342         CSRAD  = 175620        PROC3    000000R
BMASK = 000017         MTTY$  = 020000        SYSGEN= 000372
CSR     000206R        NOMT     000134R       VEC    = 000310


. ABS.  000000     000
        000220     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  8448 WORDS  ( 33 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  56 PAGES
,V4:CONSOL/L:MEB/L:TTM=V4:CONSOL
```

5.4.3 A Separate Terminal for Each Job

Once you perform a system generation for the multi-terminal feature, you can easily establish private consoles for up to eight jobs. Of course, you must be running an FB or XM monitor with the system job feature in order to support more than two jobs.

As Section 5.3 describes, simply use the FRUN/TERMINAL:n or SRUN/TERMINAL:n commands to start foreground and system jobs, and assign them to private consoles. You need not use any multi-terminal programmed requests to do this. Remember that each console is truly private – no two jobs can share terminals through the FRUN or SRUN /TERMINAL:n mechanism.

Each job can attach its own console terminal and issue subsequent multi-terminal programmed requests.

5.4.4 Multi-Terminal Applications

Some applications need to take advantage of RT–11's multi-terminal feature by using the programmed requests to manage more than one terminal per job. Typical DIGITAL applications include MU BASIC–11, CTS–300, and FMS–11. These represent applications in which one program controls several terminals. Jobs that must control more than one terminal use the multi-terminal data structures and programmed requests.

5.5 Introduction to Multi-Terminal Programmed Requests

It is not difficult for a program to use more than one terminal in a multi-terminal system. Table 5–1 summarizes the actions a program may need to take in order to use a terminal in addition to its own console terminal. It also lists the appropriate procedures for the program to follow. Familiarize yourself with the procedures and the corresponding programmed requests. The *RT–11 Programmer's Reference Manual* provides detailed information on the format of each programmed request. Study this information before you attempt to write a multi-terminal application program.

Table 5–1: Summary of Activities for a Program in a Multi-Terminal System

Activity	Procedure to Follow
Obtain the status of a multi-terminal system.	Use the .MTSTAT programmed request.
Acquire a terminal.	Use the .MTATCH programmed request to attach the terminal and dedicate it to this program. As part of its startup procedure a program usually attaches all the terminals it needs. Note that only one job can attach a shared console, and only the terminal's owner can issue multi-terminal programmed requests for it. However, all the jobs sharing the background console can issue .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT requests for it, as well as .READ and .WRITE requests for TT. To detect status changes without issuing a programmed request, examine the AST word for each terminal.
Examine the characteristics of each attached terminal.	Use the .MTGET programmed request.
Change terminal characteristics if necessary.	Use the .MTSET programmed request.

(Continued on next page)

Table 5–1: Summary of Activities for a Program in a Multi-Terminal System (Cont.)

Activity	Procedure to Follow
Get a character from a terminal and wait for it.	Use the .MTIN programmed request.
Get a character from a terminal; do not wait for it.	Use .MTSET to set the status word, then use the .MTIN programmed request. (You need issue the .MTSET only once.)
Send a character to a terminal and wait for it.	Use the .MTOUT programmed request.
Send a character to a terminal; do not wait for it.	Use .MTSET to set the status word, then use the .MTOUT programmed request. (You need issue the .MTSET request only once.)
Send a line to a terminal; wait until it prints.	Use the .MTPRNT programmed request.
Reset CTRL/O for a terminal, enabling output.	Use the .MTRCTO programmed request.
Relinquish ownership of a terminal so that another job can use it.	Use the .MTDTCH programmed request.

5.6 Multi-Terminal Data Structures

The following sections describe the two important data structures for multi-terminal systems: terminal control blocks, and asynchronous terminal status words.

5.6.1 Terminal Control Block (TCB)

RT–11 creates one terminal control block, called a TCB, for each terminal you describe at system generation time. Each TCB located in the Resident Monitor contains terminal characteristics, terminal status, and the input and output ring buffers and pointers for the terminal. The length of a TCB varies depending on the special features you select through system generation. Note, though, that the first 20 decimal words in each TCB are fixed.

5.6.1.1 Format — Figure 5–4 illustrates the format of the TCB; Table 5–2 describes its contents. An asterisk (*) marks the data structures whose size, offset, or existence depends on the special features you select through system generation.

Figure 5–4: Format of the Terminal Control Block (TCB)

	T.CNFG	
	T.CNF2	
	T.FCNT	T.TFIL
	T.WID	
	T.LPOS	T.OCHR
	T.OWNR	
	T.STAT	
	T.CSR	
	T.VEC	
	T.PRI	
	T.PUN	T.JOB
	T.PTTI	T.NFIL
	T.TNFL	T.TCTF
	T.TID	
	T.TTLC	
	T.IRNG	
	T.IPUT	
	T.ICTR	
	T.IGET	
	T.ITOP	
*	INPUT RING (DEFAULT SIZE = 134 BYTES)	
*	T.OPUT	
*		T.OCTR
*	T.OGET	
*	T.OTOP	
*	OUTPUT RING (DEFAULT SIZE = 40 BYTES)	
*	T.RTRY	
*	T.TBLK (7 WORDS)	
*	T.AST (2 WORDS IN XM)	
*	T.XCNT	T.XFLG
*	T.XPRE	
*	T.XBUF (3 WORDS)	
*	T.CNT	

Table 5–2: Contents of the Terminal Control Block (TCB)

Offset	Name	Description
0	T.CNFG	The terminal configuration word. A program and the monitor communicate with each other about terminal characteristics through the .MTGET and .MTSET programmed requests. These requests use a four-word status block within the program to store terminal information. The first word, M.TSTS, has the same structure as T.CNFG. Table 5–3 describes the meaning of each bit in T.CNFG.
2	T.CNF2	The second terminal configuration word. The structure of this word is the same as that of M.TST2, the second word of the four-word status block for .MTGET and .MTSET programmed requests. Table 5–4 describes the meaning of each bit in T.CNF2.
4	T.TFIL	Contains the character after which this terminal requires one or more fill characters. The counterpart of this byte in the four-word status block for .MTGET and .MTSET programmed requests is called M.TFIL.
5	T.FCNT	Contains the number of fill characters that this terminal requires. The counterpart of this byte in the four-word status block for .MTGET and .MTSET programmed requests is called M.FCNT.
6	T.WID	Contains the carriage width of this terminal. The counterpart of this word in the four-word status block for .MTGET and .MTSET programmed requests is called M.TWID. The maximum value is 255 decimal.
10	T.OCHR	Contains the character to output.
11	T.LPOS	Contains the current carriage position for this terminal.
12	T.OWNR	A pointer to the impure area of the job that currently owns this terminal. This word has a value when this terminal is a private console for a job, or, when it is a shared console and one job has attached it. This word is 0 when this terminal is a shared console and no job has attached it, or when it is not a console and no job has attached it. This word is simply nonzero in an SJ system if the job issues an .MTATCH request.
14	T.STAT	Contains the terminal status. Table 5–5 describes the meaning of each bit in T.STAT
16	T.CSR	Contains the CSR for the keyboard of this terminal. It is 0 if the bootstrap could not find the CSR; this makes the LUN unusable.
20	T.VEC	Contains the first interrupt vector for this terminal.
22	T.PRI	Contains the device interrupt priority.
24	T.JOB	Contains the job number of the job that currently owns this terminal.

(Continued on next page)

Table 5–2: Contents of the Terminal Control Block (TCB) (Cont.)

Offset	Name	Description
25	T.PUN	Contains the physical unit number of this terminal. This value is always 0 for terminals on DL interfaces. For terminals on DZ interfaces, the value ranges from 0 through 7 (0 through 3 for DZV11s).
26	T.NFIL	Active fill character counter. This byte contains the number of nulls left to print.
27	T.PTTI	Contains the last character typed on the terminal keyboard.
30	T.TCTF	Contains the special fill character. (For example, a space is the special fill character for a tab, and a line feed is the special fill character for a form feed.)
31	T.TNFL	Contains the count for the special fill character. The value is stored as a negative number.
32	T.TID	A pointer to the terminal identification prompt string, which contains the job name, and which is used only when the monitor is actually printing an identification. It is 0 at all other times.
34	—	Reserved.
36	T.TTLC	Contains the terminal line count (the number of lines in the input buffer).
40	T.IRNG	A pointer to the first byte of the input ring buffer. (For more information on ring buffers, see Chapter 3.)
42	T.IPUT	Input PUT pointer.
44	T.ICTR	Input character count.
46	T.IGET	Input GET pointer.
50	T.ITOP	Indicates the top of the input ring buffer. This word points to the byte just beyond the high limit of the buffer.
52	—	Input ring buffer. Its length is determined at system generation time. It is TTYIN bytes long.
	T.OPUT	Output PUT pointer.
	T.OCTR	Output character count.
	—	CTRL/O flag. A value of 0 means CTRL/O is not in effect; a value of 1 means that CTRL/O is in effect.
	T.OGET	Output GET pointer.
	T.OTOP	Indicates the top of the output ring buffer. This word actually points to the byte just beyond the high limit of the buffer.
	—	Output ring buffer. Its length is determined at system generation time. It is TTYOUT bytes long.

(Continued on next page)

Table 5–2: Contents of the Terminal Control Block (TCB) (Cont.)

Offset	Name	Description
	T.RTRY	Present if device time-out support or support for modems was selected at system generation time. This word contains the retry count for output.
	T.TBLK	Present if device time-out support or support for modems was selected at system generation time. This seven-word area is the time-out block for this terminal.
	T.AST	Present if the asynchronous terminal status word was selected at system generation time. This word is a pointer to the AST word. In XM systems, the AST pointer is followed by a second word that contains a PAR1 value for mapping to the AST word.
	T.XFLG	Present if the system job feature was selected at system generation time. If this flag byte is nonzero, it indicates that a CTRL/X sequence is in progress.
	T.XCNT	Present if the system job feature was selected at system generation time. This byte contains the number of characters typed in a CTRL/X sequence.
	T.XPRE	Present if the system job feature was selected at system generation time. This word contains the previous character typed on the terminal keyboard.
	T.XBUF	Present if the system job feature was selected at system generation time. This three-word area contains the characters typed as part of a CTRL/X sequence.
	T.CNT	Present if the system job feature was selected at system generation time. This word contains the number of jobs that are sharing the background console.

Table 5–3: Terminal Configuration Word, T.CNFG

Bit	Meaning
0	Hardware tab bit. When set, it indicates that this terminal has hardware tab support. The monitor does not convert a tab character to spaces before sending it to the output ring buffer. Your program can set this bit for a particular terminal through the .MTSET programmed request (described in Section 5.7.3). The SET TT: TAB command sets this bit for the background console.
1	When this bit is set, the monitor sends a carriage return/line feed combination to the terminal when its carriage width is exceeded. Your program can set this bit for a particular terminal through the .MTSET request. The SET TT: CRLF command sets this bit for the background console.

(Continued on next page)

Table 5–3: Terminal Configuration Word, T.CNFG (Cont.)

Bit	Meaning
2	Hardware form feed bit. When set, it indicates that this terminal has hardware form feed support. The monitor does not convert a form feed character to line feeds before sending it to the output ring buffer. Your program can set this bit for a particular terminal through the .MTSET programmed request. The SET TT: FORM command sets this bit for the background console.
3	When this bit is clear, the monitor treats CTRL/F, CTRL/B, and CTRL/X as ordinary characters and ignores their special meanings. The SET TT: NOFB command clears this bit for the background console. Your program cannot set this bit for other terminals; only the shared console can use it.
4–5	Reserved.
6	The inhibit TT wait bit. It is similar to bit 6 in the Job Status Word, which a program can set. When this bit is set, the program does not wait for I/O to complete on the terminal before execution continues. Note that bit 6 in the JSW affects only the job's current console; it does not affect any other terminals attached to this job. If the program uses other terminals for I/O, it can set this bit in each TCB by using the .MTSET programmed request. If this terminal is a private console for this job, the job can set bit 6 in the JSW. In a multi-terminal application, the job can set bit 6 in either the JSW or in the TCB for the console terminal. In any case, setting bit 6 in one place (the TCB or the JSW) results in both bits being set.
7	The XON/XOFF bit. When set, it enables recognition of the XON (CTRL/Q) and XOFF (CTRL/S) characters. The SET TT: PAGE command sets this bit for the background console. (See Chapter 3 for more information on XON/XOFF processing.)
8–11	The baud rate mask for terminals on DZ lines. (The baud rate for terminals on DL lines is not programmable through the .MTSET request.) The values are as follows: Mask — Rate 0000 — 50 0400 — 75 1000 — 110 1400 — 134.5 2000 — 150 2400 — 300 3000 — 600 3400 — 1200 4000 — 1800 4400 — 2000 5000 — 2400 5400 — 3600 6000 — 4800 6400 — 7200 7000 — 9600 7400 — not used

(Continued on next page)

Table 5–3: Terminal Configuration Word, T.CNFG (Cont.)

Bit	Meaning
12	The special mode bit. It is similar to bit 12 in the Job Status Word, which affects the job's console. If this terminal is a private console for this job, the job can set bit 12 in the JSW to enable special mode. In a multi-terminal application, the job can set bit 12 in either the JSW or in the TCB for the console terminal. In any case, setting bit 12 in one place (the TCB or the JSW) results in both bits being set. (See the description of .TTYIN in the *RT–11 Programmer's Reference Manual* for more information on special mode.) If the program uses other terminals for I/O, it can set this bit in each TCB by using the .MTSET programmed request.
13	The remote terminal bit. It is read-only, and your program cannot alter it. When set, this bit indicates that this terminal is remote.
14	When this bit is set, lower- and upper-case typing is enabled. When this bit is clear, the monitor converts all typed characters to upper-case. If this terminal is a private console for this job, the job can set bit 14 in the JSW. In a multi-terminal application, the job can set bit 14 in either the JSW or in the TCB for the console terminal. In any case, setting bit 14 in one place (the TCB or the JSW) results in both bits being set.
15	When this bit is set, the monitor takes the appropriate action for a video terminal when the DELETE key is pressed. Your program can set this bit for a particular terminal through the .MTSET programmed request. The SET TT: SCOPE command sets this bit for the background console.

Table 5–4: Second Terminal Configuration Word, T.CNF2

Bit	Meaning
0–1	These two bits indicate the length of a character. The DZ11 can transmit characters that are five, six, seven, or eight bits long. The values are as follows: Value — Character Length 00 — 5 bits 01 — 6 bits 10 — 7 bits 11 — 8 bits These bits are unused for DL interfaces.
2	Unit stop bit. Depending on the speed, it indicates the number of stop bits to send. The values are as follows: 0 = send one stop bit 1 = end two stop bits (one and one-half stop bits if five-bit characters are used) This bit is unused for DL interfaces.
3	The parity enable bit. When set, it enables parity checking.

(Continued on next page)

Table 5–4: Second Terminal Configuration Word, T.CNF2 (Cont.)

Bit	Meaning
4	Indicates whether parity checking will be odd or even. The values are as follows: Value — Parity Checking 0 — even parity 1 — odd parity This bit is unused for DL interfaces.
5–6	Reserved.
7	When set, this bit indicates "read pass-all" mode. In this mode, RT–11 transmits all eight bits of each character without interpreting or echoing the characters. This feature is often referred to as "transparency." For example, it passes ^C as 203 in "read pass-all" mode if the terminal sets the high bit upon transmission. If set, the terminal is implicity in single-character mode.
8–14	Reserved.
15	When set, this bit indicates "write pass-all" mode. In this mode, RT–11 transmits all eight bits of each character without interpreting the characters.

Table 5–5: Terminal Status Word, T

Bit	Meaning When Set
0	Indicates that a fill sequence is in progress.
1–3	Reserved.
4	Indicates that a detach operation is in progress. Input from the terminal is ignored.
5	This is the TT handler synchronization bit.
6	Indicates that an output interrupt is expected.
7	Indicates that the terminal has sent XOFF to request suspension of output.
8–9	Reserved.
10	Indicates that this terminal is the shared console.
11	Indicates that the remote terminal has hung up.
12	Indicates that the terminal interface is a DZ.
13	Reserved.
14	Indicates that two CTRL/Cs were typed at this terminal. This bit is reset by .MTGET.
15	Indicates that this terminal is a console for some job. It can be shared or private.

5.6.1.2 Patching a TCB – You can use SIPP to make binary patches to the terminal control blocks in your monitor file, *xxxx*.SYS. The TCBs are located in p-sect MTTY$, which you can find on your monitor link map. They appear in the same order in which SYSGEN assigned physical units to logical unit numbers at system generation time (see Section 5.2). The first TCB is for LUN 0; it starts at the label DLTCB::. The TCBs are all the same size; TCBSZ contains their length.

5.6.2 Asynchronous Terminal Status (AST) Word

The asynchronous terminal status (AST) word is a special feature that you can select at system generation time. If you select this feature, you can set aside space for one AST word per LUN in your own program. Then, when you issue the .MTATCH programmed request to attach a terminal to your job, you specify as an argument the address of the AST word for that terminal. The purpose of the AST word is to monitor each terminal's line so that the program can obtain certain information without issuing a programmed request. RT–11 sets or clears bits in the AST word as significant events occur. The AST word contains information on whether:

- Input is available from the terminal
- The terminal's output ring buffer is empty
- Double CTRL/C was typed on the terminal
- A remote line just dialed in or just hung up

Table 5–6 shows the event flags in the AST word and their meaning. Unused bits are reserved for future use by DIGITAL.

Table 5–6: Asynchronous Terminal Status (AST) Word

Bit	Name	Bit Pattern	Meaning When Set
15	AS.CTC	100000	Double or multiple CTRL/C was typed on this terminal. You must reset this bit; the monitor never turns it off.
14	AS.INP	40000	Input is available from this terminal.
13	AS.OUT	20000	The output ring buffer is empty.
7	AS.CAR	200	Carrier is present (for remote lines only).
6	AS.HNG	100	This remote line just hung up and RT–11 dropped it.

The monitor sets bit 15, AS.CTC, whenever two or more consecutive CTRL/Cs are typed on any terminal. Typing two CTRL/Cs on a job's console terminal always aborts the job, unless the job already issued the .SCCA programmed request to intercept the characters. The job must reset this bit before it continues processing.

The monitor sets bit 14, AS.INP, when input is available from the terminal. It can be a line of characters in normal mode, or a single character in special mode. The monitor clears this bit when the program reads the characters.

The monitor sets bit 13, AS.OUT, when the terminal's output ring buffer is empty. This occurs after the last character in the ring buffer is printed on the terminal. The monitor clears this bit when there are characters in the ring buffer.

The monitor sets bit 7, AS.CAR, when it answers a remote line. It clears this bit when the remote line hangs up or drops carrier. *Carrier* is a tone transmitted over the remote line. It carries information through its modulation.

The monitor sets bit 6, AS.HNG, when it drops a remote line that just hung up.

5.7 Using the Multi-Terminal Programmed Requests

The routines in MTTEMT, which are part of the Resident Monitor, dispatch the multi-terminal programmed requests and process them.

The dispatch routine accepts programmed requests that translate into EMT 375 instructions with a subcode of 37 and a function code in the range 0 through 10 octal. The dispatch routine first checks to see if the programmed request is a valid one. Then it verifies the logical unit number and makes sure that the terminal is installed. If the programmed request is for an attach operation, the dispatch routine verifies that the terminal is not already attached. For all other requests, the dispatch routine verifies that the terminal is attached to the calling program.

If the request passes all the checks in the dispatch routine, control passes to the EMT processing code for the individual request.

5.7.1 Attaching a Terminal: .MTATCH

Issue the .MTATCH programmed request to attach a terminal to your job. This permits your program to print characters on the terminal, get characters from it, and alter its characteristics.

When a job attaches a terminal, the terminal remains attached until the job issues a .MTDTCH request, or until the job exits or aborts. If the terminal is detached through the .MTDTCH request, the job is blocked until output in process for the terminal finishes and the monitor detaches the terminal. If the terminal is detached when the job aborts, the output terminates and the monitor detaches the terminal immediately.

The attach routine first checks to see if the terminal is the shared console, but not this job's console. If so, the routine issues error code 4. If the terminal is already attached to another job, the routine also issues error code 4. No other errors can occur in the attach operation.

The routine attaches the terminal by setting up two words in the TCB for this terminal. In FB and XM systems, it stores the job number in T.JOB. In SJ systems, T.OWNR is made nonzero when the terminal is attached. In FB and XM systems, T.OWNR contains a pointer to the owning job's impure area.

If AST support is part of the system, the routine puts a pointer to the AST word in T.AST. In XM systems, it also stores a value in T.AST + 2 to be used as a PAR1 value in mapping to the AST word.

The routine next moves some bits from the JSW into T.CNFG, if this terminal is the job's console. It copies bits 14 (for lower case), 12 (special mode), and 6 (wait inhibit). If the terminal is the background console the attach routine loads T.TFIL from location 56.

5.7.2 Getting Terminal Status: .MTGET

Issue the .MTGET programmed request to obtain the status of a terminal. (The terminal need not be attached to your program in order to obtain the status.)

The .MTGET routine moves information from the TCB to the status block in your program. The following transfers occur:

T.CNFG to M.TSTS
T.CNF2 to M.TST2
T.TFIL to M.TFIL
T.FCNT to M.FCNT
T.WID to M.TWID
high byte of TSTAT to M.TSTW

Then, if the terminal is not attached to any job, the routine returns error code 1. If the terminal is attached, but not to this job, the routine returns error code 4 and R0 contains the job number of the terminal's owner. If the terminal is the shared console, but the job has its own private console, R0 contains the job's own job number. Note that despite the fact that an error is returned from this operation, the status information is always placed in the status block in your program.

Finally, if no error was returned, the routine clears bit 14 (CTRL/C) in T.STAT.

5.7.3 Setting Terminal Characteristics: .MTSET

Issue the .MTSET programmed request to set the characteristics of a terminal. If the terminal is not attached to your program, the routine gives error code 1.

The routine moves the contents of M.TSTS to T.CNFG, except for bit 13 (the remote terminal bit), which is read only in T.CNFG. If the terminal is the job's console, the routine moves some bits from T.CNFG into the JSW. It copies bits 14 (for lower case), 12 (special mode), and 6 (wait inhibit).

Whether or not the terminal is the job's console, the routine moves the following information:

M.TST2 to T.CNF2
M.TFIL to T.TFIL
M.FCNT to T.FCNT
M.TWID to T.WID

If DZ support is part of the system, and if this terminal is on a DZ interface, the routine waits for any characters to finish printing on this terminal, then sets up the DZ line parameters.

NOTE

Always issue an .MTGET request before an .MTSET request. Change only the fields you are interested in. For a one-bit field, use a BIS or BIC instruction to set or clear it. For a multiple-bit field, clear it first with a BIC and then use BIS to load the field. Use MOVB or MOV instructions only for byte or word fields. Changing other bits can cause unusual terminal service errors. Finally, issue the .MTSET specifying the same status block that you used for the .MTGET request.

5.7.4 Getting a Character: .MTIN

Issue the .MTIN programmed request to get a character from the terminal.

The routine moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies bits 14 (for lower case), 12 (special mode), and 6 (wait inhibit). If the terminal is the background console, the attach routine loads T.TFIL from location 56.

The routine gets a character from the input ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB.

If the input character is CTRL/C on a console terminal, and .SCCA is not in effect, the job aborts.

5.7.5 Printing a Character: .MTOUT

Issue the .MTOUT programmed request to print a character on the terminal.

The routine moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies bits 14 (for lower case), 12 (special mode), and 6 (wait inhibit). If the terminal is the background console, the attach routine loads T.TFIL from location 56.

The routine moves a character from the user buffer into the output ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB.

5.7.6 Printing a Line: .MTPRNT

Issue the .MTPRNT programmed request to print a string of characters on the terminal. The string can end with a null byte (to print a carriage return and a line feed at its end) or a 200 octal byte, just as in the .PRINT programmed request.

The routine moves a line from the user buffer into the output ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB. If there is no room in the output ring, the job is blocked until room is available, regardless of the value of bit 6 in T.CNFG.

5.7.7 Resetting CTRL/O: .MTRCTO

Issue the .MTRCTO programmed request to enable output on a terminal even though CTRL/O may have been typed.

This routine clears the CTRL/O flag in the TCB for the terminal and moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies bits 14 (for lower case), 12 (special mode), and 6 (wait inhibit). If the terminal is the background console, the attach routine loads T.TFIL from location 56.

If you ever alter the contents of the JSW, DIGITAL recommends that your program issue the .MTRCTO request immediately afterward so that the TCB and the JSW always have the same information. In particular, if you require lower-case input for a .GTLIN request, set bit 14 in the JSW and issue .MTRCTO or .RCTRLO before using .GTLIN.

5.7.8 Getting System Status: .MTSTAT

Issue the .MTSTAT programmed request to obtain status information about the multi-terminal system. This request returns the following four words of information to your program:

- The offset from the start of the Resident Monitor to the first TCB
- The offset from the start of the Resident Monitor to the TCB of the current console terminal for this job
- The vaule of the highest TCB (equivalent to the highest LUN)
- The size of each TCB in bytes. (Note that all TCBs are the same size.)

Remember that the TCBs are located in the Resident Monitor in the order in which you specified your DL and DZ lines to the SYSGEN dialogue. That is, the TCBs for local DLs appear first, followed by remote DLs, local DZs, and remote DZs.

With the information returned to you by .MTSTAT you can find the TCB for any terminal in the system and examine its contents with the .GVAL request. Figure 5–4 and Table 5–2 describe the contents of each TCB.

5.7.9 Detaching a Terminal: .MTDTCH

Issue the .MTDTCH programmed request to detach a terminal from your job and make it available for use by another job.

The routine first sets the DTACH$ bit, bit 4, in T.STAT to indicate that a detach operation is in progress. This avoids any race conditions in the module MTTINT. (A race condition is a situation in which two or more 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.) It then forces XON if XOFF had been previously set. If the terminal is not a shared console, the output buffer is then flushed. In SJ, the routine loops until T.OUTR is clear. In FB and XM, the job is blocked until T.OCTR is clear.

The words T.OWNR and T.AST are set to zero to detach the terminal. DTACH$ is finally cleared to finish the operation.

Whenever a job aborts, terminals attached to it are detached without having their buffers flushed.

5.8 Summary of Multi-Terminal Programmed Request Error Codes

Table 5–7 summarizes the error codes that the multi-terminal programmed requests can put into byte 52. Table 5–8 shows which error codes each programmed request can generate.

5.9 The Console as a Special Case

The console terminal is always a special case for I/O in multi-terminal systems. Recall that each job has input and output ring buffers and pointers, both in its console's TCB and in its impure area. Whenever a job gets characters from its console terminal, or writes characters to it, the monitor uses the set of ring buffers located in the job's impure area. In this case, the console can be the background console, if this job is sharing it, or it can be a private console, if this job has one.

For all I/O requests involving the job's console, the monitor performs the request based on the characteristics indicated in the Job Status Word rather than in the terminal configuration word. However, if you set or clear a

Table 5–7: Multi-Terminal Programmed Request Error Codes and Meanings

Byte 52 Code	Meaning
0	There is no character in the input ring buffer for this terminal; or, there is no room in the output ring buffer for this terminal.
1	The logical unit number is invalid.
2	The logical unit number does not exist.
3	The programmed request you issued is invalid. The function code for EMT 375, subcode 37, must be in the range 0 through 10 octal.
4	This terminal is already attached to another job. The program cannot attach it, detach it, or set its status.
5	The user buffer address, the status block, or the AST word address is outside the valid addressing space for this program. This error occurs in XM systems only.

Table 5–8: Summary of Error Codes

Programmed Request	Error Code 0	1	2	3	4	5
.MTATCH			X	X	X	X
.MTGET		X	X	X	X	X
.MTSET		X	X	X		X
.MTIN	X	X	X	X		X
.MTOUT	X	X	X	X		X
.MTPRNT		X	X			X
.MTRCTO		X	X	X		
.MTSTAT						X
.MTDTCH		X	X	X		

terminal-related bit in the JSW, the monitor automatically sets or clears the corresponding bit in the terminal configuration word for the job's console the next time the job does any kind of input or output request or reset CTRL/O request for that terminal (see Table 5–3). DIGITAL recommends that you issue the .MTRCTO request immediately after altering the JSW to make sure that the contents of the JSW are duplicated in the TCB for the terminal. Similarly, if you modify the terminal configuration-word with .MTSET for a job's console, the monitor also modifies the JSW.

On entry to the EMT processor, R3 contains a pointer to the job's TCB, and R5 contains a pointer to the impure area.

Note that a program must issue the .SCCA programmed request to inhibit CTRL/C on its console terminal.

5.10 Interrupt Service

Terminal service in multi-terminal systems is centralized in the routines contained in MTTINT. This source file is assembled and linked together with other files to become part of the Resident Monitor.

In general, RT–11 services terminals in one of two ways, depending on whether the terminal is connected through a local or a remote line.

5.10.1 Local Terminals

Local terminals are connected to an interface by a minimum of four wires:

- Receive data
- Transmit data
- Receive ground
- Transmit ground

Some interface circuitry, such as the EIA RS232-C, combines the receive ground and transmit ground into one signal ground; for these, a minimum of three wires in required. In addition, PDT–11 terminal ports require that the data terminal ready signal be connected and asserted for proper operation.

RT–11's interrupt service routine for multi-terminal systems contains the following data structures:

- Receive CSR I/O page address
- Receive data buffer I/O page address
- Transmit CSR I/O page address
- Transmit data buffer I/O page address

RT–11's interrupt service is essentially simple. The bootstrap sets the input (or receiver) interrupt enable bit; the monitor leaves it set at all times. If a character is typed on a local terminal, an interrupt occurs and the monitor picks up the character. If the terminal is not attached to any job, the character is ignored. In multi-terminal systems with time-out support, the monitor turns on the interrupt enable bit for each DL once every 30 clock ticks.

The monitor only sets the output interrupt enable bit when it is ready to print a character. It clears the bit after the output ring buffer is empty.

5.10.2 Remote Terminals

Remote terminals are connected to RT–11 through modems (also known as data sets) and telephone lines so that someone can call up the computer and ring its data phone. When this occurs, it causes an interrupt, which the monitor recognizes. If the unit is attached, the multi-terminal service routine answers the phone call and sends out carrier in response. (Carrier is a tone transmitted over the remote line that carries information through its modulation.)

The remote terminal can communicate with RT–11 through an approved protocol. Essentially, the terminal must send its own carrier to the computer. If the terminal immediately sends carrier, RT–11 recognizes the signal, and I/O can begin. If, however, the terminal does not send its own carrier immediately, RT–11 sets a 30-second timer. This time interval gives someone an opportunity to place a telephone receiver into an acoustic coupler. If the terminal does not send carrier within 30 seconds, RT–11 disconnects the line.

Once communication has begun, RT–11 never takes the initiative to terminate the connection. It always continues to send carrier. However, there are two situations in which RT–11 does hang up on the remote line. If the terminal stops sending carrier for any reason, RT–11 waits two seconds for it to resume. When the interval expires, RT–11 hangs up on the remote line. In the other situation, the remote terminal hangs up. RT–11 detects loss of carrier and waits two seconds before disconnecting the remote line. Special requirements for customers in the United Kingdom are met through assemblies based on the U.K. conditional being set to 1.

Remote terminals require a DL11-E, DLV11-E (or equivalent, such as the PDT–11 modem port), or DZ interface. In addition to the data lines required for remote terminals, the following control lines must be connected:

- data terminal ready
- ring indicator
- carrier detect

A local terminal can be connected to a remote terminal interface if it is identified during system generation as a local terminal. The control lines listed above are then ignored and you can leave them unconnected.

5.11 Polling Routines

RT–11's multi-terminal support includes two polling routines, which the following sections describe.

5.11.1 Time-Out Routine for DL Terminals

You can select the time-out polling routine as a special feature at system generation time. It is an example of the device time-out feature that is available to application programs through the .TIMIO programmed request. RT–11 executes this routine once every half second. Its purpose is to periodically reenable the I/O interrupt enable bits so that noise on a line or local static electricity cannot seriously affect transmissions.

Every half second, the polling routine examines each DL line on the system. It turns on the line's input interrupt enable bit and, if the line is remote, its modem interrupt enable bit. Then, if output is pending with no output interrupt, it turns the output interrupt enable bit off and then on, to force an output interrupt on the line. (Depending on the hardware failure that caused the loss of the output interrupt, this may occasionally cause a character to be repeated.)

The last thing the time-out routine does is schedule itself to run again.

5.11.2 DZ Remote Line Polling Routine

The DZ polling routine polls the terminals connected to the system through DZ interfaces. It is necessary because these terminals do not interrupt when their status changes.

The remote line polling routine schedules a mark time request. It waits 30 seconds after the data set rings to detect carrier. If there is no carrier after the required amount of time, the routine disconnects the remote line. The routine takes similar action on line errors and lost carrier. This routine is automatically included in the multi-terminal service for remote DZ lines.

5.12 Restrictions

The following restrictions apply to systems with the multi-terminal special feature:

1. Support of the DL11-W interface requires the presence of a REV E or later module. In the absence of a REV E module, ECO (Engineering Change Order) number DEC-O-LOG M7856-S0002 must be applied to the M7856 module.

   Support of the DLV11–J interface requires the presence of a REV E or later module. In the absence of such a module, ECO M8043-MR002 must be applied to the M8043 module.

2. The multi-terminal handler can support remote terminals. Modem control is available for both DL11-E and DZ11 interfaces. The DL11 control answers ring interrupts, permitting terminals to dial in to the system. Dial-in is possible with the DZ11 interface, despite lack of a ring interrupt in the DZ11, if the modem is operated in auto-answer mode. This is achieved through a polling routine that periodically checks the status of each line on the multiplexer (see Section 5.11.2). Dial-up support for DZ interfaces requires BELL 103A-type modems with "common clear to send and carrier" jumpers installed. With this option installed, the modem operates in auto-answer mode.

3. The hardware console interface must be a DL interface, and it must be a local terminal. You can use the SET TT: CONSOL command to move the background console to any other local terminal in the system.

4. The number of DL interfaces RT–11 supports, both local and remote, is limited to eight. This number includes the hardware console interface.

5. The number of DZ11 controllers is limited to two, for a total of 16 lines. The total of DZV11 controllers is limited to four, for the same total of 16.

6. The VT11 scroller option is disabled when the multi-terminal special feature is present in a system. The commands GT ON and GT OFF are not valid in multi-terminal systems. For this reason, EDIT cannot use the display support. The use of graphics is still supported, though, and the display support in TECO works as well.

7. The maximum input data rate for a single terminal is 300 baud. The aggregate total input data rate for a system is 4800 baud.

   You can set the output baud rate to any speed; RT–11 sends output as fast as possible, depending on the capacity of the CPU and the nature of its load.

8. When you type double CTRL/C in an SJ system, the monitor does a hardware RESET instruction. This causes the DZ multiplexer to reset its status and to drop Data Terminal Ready on all lines, thus hanging them up. This action is part of the general cleanup the system performs after a program aborts.

9. If you plan to devote a terminal line to the LS handler, do not specify that terminal's DL interface in the SYSGEN dialogue for a multi-terminal system. Do not attempt to attach the terminal from a multi-terminal application program, either.

10. Setting the baud rate, character length, number of stop bits, and parity via the .MTSET programmed request is supported only for DZ interfaces.

5.13 Debugging a Multi-Terminal Application

Use VDT, the Virtual Debugging Technique, to debug a multi-terminal application. See Section 4.9 for more information on VDT.

5.14 Multi-Terminal Example Program

Figure 5–5 shows a program that uses the multi-terminal programmed requests.

Figure 5–5: Multi-Terminal Example Program

```
.TITLE
  MTYSET.MAC - Auto-baud and Initialize DEC Terminals
.IDENT    /X05.00/

;                    COPYRIGHT (c) 1982, 1983 BY
;            DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
;                 ALL RIGHTS RESERVED.
;
```

Figure 5–5: Multi-Terminal Example Program (Cont.)

```
; 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
  .NLIST  BEX
  .ENABL  GBL

;+
; Auto-baud and Initialize DEC Terminals
;
; AUTHOR: L.C.P.  -   10/79
;
; This program will attach all "Known" terminals and
; if they are VT5x, VT1xx or LA1xx series it will determine
; at what baud rate they are set and put that information in
; their TCBs. ("Foreign" terminals will be assumed to
; be set at the correct baud rate). As each terminal is
; "initialized", its screen will be cleared, a "sign-on"
; message will be displayed, and the terminal type and
; baud rate will be logged on the background console.
;-

.SBTTL    Macros and definitions

.MCALL    .MTATCH,.MTDTCH,.MTGET,.MTOUT,.MTIN
.MCALL    .MTPRNT,.MTSET,.MTSTAT,.EXIT
.MCALL    .MTRCTO,.PRINT,.TTYOUT,.MRKT,.CMKT

M.TSTW    = 7             ;Offset to state word in TCB
S.FTCB    = 0             ;Stat offset to 1st TCB offset
S.CTCB    = 2             ;Stat offset to console TCB
S.NTCB    = 4             ;Stat offset to # TCB (LUN)
S.STCB    = 6             ;Stat offset to TCB size
MSPEED    = 7400          ;Baud rate mask = bits 8-11
TCBIT$    = 100           ;Inhibit TT wait
TTSPC$    = 10000         ;TT special bit
HNGUP$    = 4000          ;Terminal had hung up (offline)
DZ11$     = 10000         ;DZ11
REMOT$    = 20000         ;DZ11 line is remote
BKSP      = 100000        ;Backspace for rubout(delete)
TAB       = 1             ;Hardware tab
NOCRLF    = 2             ;*CLEAR* CRLF bit
LF        = 12            ;Line feed
CR        = 15            ;Carriage return
ESC       = 33            ;Escape

.SBTTL    Start of program

.ENABL    LSB,LC                   ;MUST enable Lower case!

MTYSET:   MOV     #STAT,R3
;R3 => 8 word status
  .MTSTAT #AREA,R3        ;Get MTTY status
  MOV     S.NTCB(R3),R2   ;R2 = # of LUNs
  BEQ     MTEXIT          ;Just exit if none!
  MOV     S.CTCB(R3),R4   ;R4 = Offset to console
                          ;TCB
```

Figure 5–5: Multi-Terminal Example Program (Cont.)

```
  SUB      @R3,R4            ;R4 = Diff from 1st TCB
  BEQ      1$                ;No difference, so
                             ;LUN:0 = console...
  MOV      S.STCB(R3),R5     ;R5 = Size of TCB
  CLR      R1                ;R1 = Quotient
DIV$:      INC      R1               ;Divide diff by size
                             ;of a TCB
  SUB      R5,R4             ;to get LUN of console
  BHI      DIV$              ;Repeat until done...
  MOV      R1,(PC)+          ;Save console LUN...
CLUN:      .WORD    0                ;for later reference
1$:        CMP      R2,CLUN          ;Is this the Console?
  BEQ      4$                ;Yes...already set up
  .MTATCH  #AREA,#0,R2       ;Try to attach terminal
  BCS      MTERR1            ;If carry set, can't !
  .MTGET   #AREA,R3,R2       ;Get terminal's status
  BCS      MTERR2            ;Can't! (Very Bad!!!)
  BITB     #DZ11$/400,M.TSTW(R3) ;Is line a DZ11?
  BEQ      6$                ;No...assume a DL11
  BITB     #REMOT$/400,M.TSTW(R3) ;Remote line?
  BEQ      2$                ;Nope...
  BITB     #HNGUP$/400,M.TSTW(R3) ;Is it online?
  BNE      5$                ;Branch if not
2$:        CALL     TSETUP           ;Figure out baud rate
                             ;and terminal type
3$:        .MTRCTO  #AREA,R2
;Reset CTRL/O
  .MTPRNT  #AREA,#HELLO,R2   ;Clear screen (if CRT)
                             ;and say hello...
  CALL     LOGLUN            ;Log term ID on console
4$:        DEC      R2               ;Are we finished?
  BPL      1$                ;No...go do another LUN
MTEXIT:    .EXIT                     ;We're done...exit

.SBTTL     Terminal ID Log routines, error routines

5$:        .PRINT   #OFFLIN          ;Log terminal offline
  CALL     PRNLUN            ;Include LUN...
  .PRINT   #CRLF             ;...and CRLF
  BR       4$                ;Merge...
6$:        BIS      #<TTSPC$!TCBIT$>,@R3 ;DL11 - Set the
                             ;special bits in TCB
  MOV      #ENDTBL,R4        ;Don't know speed...
  CALL     TERMID            ;Try to figure out
                             ;the terminal ID
  CALL     RSET              ;Set new status...
  BR       3$                ;Merge...

LOGLUN:    .PRINT   #ATMSG           ;Print 1st part of log
  CALL     PRNLUN            ;Print LUN...
  .PRINT   R1                ;...then terminal ID...
  .PRINT   #TINIT            ;...and finally...
  .PRINT   R4                ;....the baud rate
  RETURN

PRNLUN: MOV         R2,R0            ;Copy LUN into R0
  SWAB     R0                ;Put it in high byte
7$:        ADD      #<-10.*400>+1,R0 ;Divide by 10 with
                             ;repeated subtracts
  BPL      7$                ;Q=Q-10, R=R+1 till
                             ;overflow (V set)
  ADD      #'0*400+'0+<10.*400-1>,R0 ;Correct
                             ;Q & R then ASCIIfy...
  .TTYOUT                    ;Print Q...
  SWAB     R0                ;R to low byte...
  .TTYOUT                    ;Print it...
  RETURN

MTERR1:    .PRINT   #MSG1            ;Log attatch error
  BR       8$                ;Merge
```

Figure 5–5: Multi-Terminal Example Program (Cont.)

```
MTERR2:    .PRINT   #MSG2            ;Log get status error
8$:        CALL     PRNLUN           ;Include LUN
  BR       4$               ;Try next LUN

.SBTTL     Main terminal setup subroutine

TSETUP:    MOV      #SPTABL-2,R4     ;R4 => Baud rate table
  MOV      @R3,MSTAT        ;Save old status...
  BIS      #<TTSPC$!TCBIT$>,@R3 ;Set special bits
10$:       TST      (R4)+            ;R4 => Next table entry
  BIC      #MSPEED,@R3      ;Clear baud rate mask
  MOV      (R4)+,R5         ;R5 = Baud from table
  BIS      R5,@R3           ;Set it in CONFG1
  CMP      #ENDTBL,R4       ;Are we thru table?
  BEQ      14$              ;Yes...use as is
  MOV      #32,LOTIM        ;Magic # in .MRKT arg
  SWAB     R5               ;Put mask in low byte
  SUB      R5,LOTIM         ;Subtract from magic #
                            ;to get # ticks to wait
  CALL     TERMID           ;Try to get terminal ID
  BCS      10$              ;No dice...
RSET:      BIC      #<TTSPC$!TCBIT$>,@R3 ;Clear special bits
  BIC      (R1)+,@R3        ;Turn off unwanted options
  BIS      (R1)+,@R3        ;Turn on desired options
                            ;R1 => Terminal ID string
12$:       MOV      @R4,R4           ;R4 => ASCII baud rate
13$:       .MTSET   #AREA,R3,R2      ;Store status
  RETURN                    ;Return to caller

14$:       CALL     GETSP            ;Get ASCII of baud rate
  BR       13$              ;Merge...

TERMID:    .MTSET   #AREA,R3,R2      ;Set new status
  MOV      #TTLIST,R5       ;R5 => List of Terminals
15$:       MOV      (R5)+,R1
;R1 => Terminal specific
                            ;character sequence
  BEQ      18$              ;End of table - leave !
  CALL     TOUT             ;Try to communicate...
  BCS      15$              ;Carry set = no dice
  ADD      OUTCT,R1         ;R1 => Expected response
  BIT      #1,R1            ;Odd address?
  BEQ      16$              ;No...
  INC      R1               ;YES! Make it even
16$:       CMP      MSGIN,(R1)+      ;Match?
  BNE      15$              ;Nope...
  CMP      MSGIN+2,(R1)+    ;Still match?
  BNE      15$              ;Nope...
  RETURN                    ;Return with R1 => options
18$:       MOV      #UNKTT,R1        ;R1 => "Unknown terminal"
  SEC                       ;Set carry...
  RETURN

.SBTTL     Terminal I/O & Get baud rate routines

TOUT:      MOVB     (R1)+,INCNT      ;Get # char in response
  MOVB     (R1)+,OUTCT      ;Get # char in "What-
                            ;are-you?" sequence
  .MTOUT   #AREA,R1,R2,OUTCT ;Send What-are-you?
  BCS      20$              ;Output error
  CLRB     TFLG             ;Clear flag
  CLR      MSGIN+2          ;init input buffer
  .MRKT    #AREA,#WAITM,#CRTNE,#1 ;Set time-out
19$:       TSTB     TFLG
  BEQ      19$
  .MTIN    #AREA,#MSGIN,R2,INCNT ;Get response,
20$:       RETURN                    ;(with carry status)
```

Figure 5–5: Multi-Terminal Example Program (Cont.)

```
GETSP:     MOV     #SPTABL,R4        ;R4 => baud rate table
  MOV      @R3,R5           ;R5 = TCB config word 1
  BIC      #^C<MSPEED>,R5   ;Clear all but baud rate
21$:       CMP     (R4)+,R5
;compare it with table
  BEQ      22$              ;Branch if equal
  CMP      #UNKSP,(R4)+     ;End of table?
  BNE      21$              ;Try another if not
22$:       MOV     @R4,R4            ;R4 => ASCII baud rate
  RETURN                    ;Return to caller

.SBTTL     Timeout Completion Routine

CRTNE:     INCB    TFLG              ;Set time-out flag
  RTS      PC               ;Return to mainline

; Argument blocks & working storage

INCNT:     .WORD   0                 ;Input byte count
OUTCT:     .WORD   0                 ;Output byte count
AREA:      .BLKW   5                 ;EMT Argument block
WAITM:     .WORD   0                 ;Time-out argument
LOTIM:     .WORD   0                 ; Lo order ticks
STAT:      .BLKW   8.                ;Status block (8 words)

.SBTTL     Baud rate mask & ASCII baud rate tables

; Baud rate table - in "best guess" order

SPTABL:    .WORD   7000,B9600        ;9600 baud ;Scopes
  .WORD    3400,B1200       ;1200 baud ;LA120
  .WORD    2400,B300        ;300  baud ;LA36
  .WORD    6000,B4800       ;4800 baud ;Scopes
  .WORD    5000,B2400       ;2400 baud ;Scopes
  .WORD    2000,B150        ;150  baud ;LA36
  .WORD    1400,B134        ;134.5 baud ;IBM
MSTAT:     .WORD   0                 ;Orig status
ENDTBL:    .WORD   UNKSP             ;End-of-table
                            ;=> "Unknown baud"
MSGIN:     .BLKB   8.                ;Response buffer
TFLG:      .BYTE   0                 ;Time-out flag

.NLIST     BEX

B134:      .ASCIZ  /134.5 Baud/
B150:      .ASCIZ  /150  Baud/
B300:      .ASCIZ  /300  Baud/
B1200:     .ASCIZ  /1200 Baud/
B2400:     .ASCIZ  /2400 Baud/
B4800:     .ASCIZ  /4800 Baud/
B9600:     .ASCIZ  /9600 Baud/
  .EVEN

.SBTTL     Terminal ID tables

TTLIST:                              ;Terminal List...
  .WORD    VT100
  .WORD    VT52
  .WORD    LA120
  .WORD    LA34
  .WORD    VT55
  .WORD    0                ;Table Stopper


; DEC terminal command sequences

VT100:     .BYTE   4,3,ESC,'[,'c     ;INCNT,OUTCNT,"W-A-Y" seq
  .EVEN
  .BYTE    ESC,'[,'?,'1     ;Response
  .WORD    NOCRLF,<TAB!BKSP> ;Undesired,Desired options
  .ASCII   / VT100/<200>    ;ASCII terminal ID
```

Figure 5–5: Multi-Terminal Example Program (Cont.)

```
VT52:      .BYTE   2,2,ESC,'Z
  .EVEN
  .BYTE    ESC,'/,0,0       ;VT52 response varies w/ model!
  .WORD    NOCRLF,<TAB!BKSP>
  .ASCII   / VT52 /<200>

LA120:     .BYTE   4,3,ESC,'[,'c
  .EVEN
  .BYTE    ESC,'[,'?,'2
  .WORD    0,0
  .ASCII   / LA120/<200>

LA34:      .BYTE   4,3,ESC,'[,'c
  .EVEN
  .BYTE    ESC,'[,'?,'3
  .WORD    0,0
  .ASCII   / LA34 /<200>

VT55:      .BYTE   2,2,ESC,'Z
  .EVEN
  .BYTE    ESC,'E,0,0
  .WORD    NOCRLF,<TAB!BKSP>
  .ASCII   / VT55 /
  .EVEN

.SBTTL     Message text & Initialization string

; Message text...

MSG1:      .ASCII   <CR><LF>/?Cannot attach terminal LUN:/
  .ASCII   <200>
MSG2:      .ASCII   <CR><LF>/?Status error - LUN:/<200>
ATMSG:     .ASCII   <CR><LF>/Attaching LUN:/<200>
TINIT:     .ASCII   / initialized at /<200>
UNKSP:     .ASCIZ   /unknown baud rate/
UNKTT:     .ASCII   / unidentifiable terminal/<200>
OFFLIN:    .ASCII   /Terminal offline - LUN:/<200>
CRLF:      .ASCIZ   //

; Clear screen & say hello character string...

HELLO:     .ASCII   <ESC>"[2J"        ;VT100 Erase screen
  .ASCII   <ESC>"\";VT52 "Exit hold
                    ;      screen mode"
  .ASCII   <ESC>"H"<ESC>"J" ;VT52 Home + "Erase-
                    ;      to-End-of-Screen"
  .ASCII   <CR><LF>        ;CRLF (for hardcopy)
  .ASCIZ   /TERMINAL INITIALIZED/

.END       MTYSET                     ;End of program
```

Chapter 6
Interrupt Service Routines

This chapter describes the ways a program can transfer data between memory and a peripheral device. First it covers non-interrupt programmed I/O; next it introduces the concept of using interrupts to handle device I/O by comparing the advantages and disadvantages of in-line interrupt service routines and device handlers. After these general points have been discussed, the chapter continues with a description of the structure of an interrupt service routine, and shows in detail how to organize and write one. A skeleton example of a foreground program that contains an interrupt service routine ends this discussion of applications. The discussion is followed by a final section dealing with the considerations involved in using interrupt service routines in an extended memory environment.

6.1 Non-Interrupt Programmed I/O

One way to move data between memory and a peripheral device is to use non-interrupt programmed I/O. According to this method, your program operates with the device interrupts disabled and uses flags to coordinate the data transfer. Your program checks the ready bit in the status register for a particular device, moves the data when appropriate, and then either waits in a tight loop for another ready signal or does other processing and polls the device occasionally. Programmed I/O is device-specific and does not make use of operating system features designed for I/O processes. In addition, it ties up system resources until the I/O transfer is complete.

However, programmed I/O is sometimes the best method to use. For example, the Resident Monitor uses programmed I/O to print its *?MON-F-System halt* error message. It first performs a RESET to stop all active I/O. Then it waits in a tight loop for the console terminal to print the error message, one character at a time. Clearly in such a situation, where the monitor itself may be corrupted, no other job or data transfer could be running, and the console terminal is the only desirable output device. Also, the monitor .PRINT routine may have been corrupted and should not be used. Given these requirements, programmed I/O is the best method to use for printing this error message.

In an application program you could use non-interrupt programmed I/O for a time-critical device when the program must respond as soon as a character becomes available in a register.

The following lines of code from RMON demonstrate non-interrupt programmed I/O:

```
;
; Note that R1 points to the message text.
; TTPS is a word in memory containing the address of
; the terminal printer status register;
; its ready flag is the high-order bit of the low byte.
; TTPB is a word in memory containing the address of
; the terminal printer buffer.
; Moving a character to the printer buffer resets
; the busy flag in the status register.
;
5$:     TSTB    @TTPS           ;TEST FOR TT BUSY
        BPL     5$              ;IF YES, TEST AGAIN
        MOVB    (R1)+,@TTPB     ;IF NO, PRINT A CHARACTER
        BNE     5$              ;BRANCH BACK IF MORE TO PRINT
```

The device handler for the single-density diskette, DX, provides another example of programmed I/O. Reading data from the diskette one sector at a time, the handler first requests a read of one sector. The diskette completes the read operation, places the data in an internal silo, and issues an interrupt. The handler then disables diskette interrupts and uses programmed I/O to move data from the silo into memory. When it is ready to read another sector, the handler enables interrupts again.

The following lines of code are from a DX handler:

```
;
; Note that R4 points to the diskette status register;
; R5 points to the silo;
; R2 points to the data buffer in memory.
;

TRBYT:  TSTB    @R4             ;WAIT FOR TRANSFER READY
        BPL     TRBYT           ;BRANCH IF TR NOT UP
EFBUF:  MOVB    @R5,(R2)+       ;TRANSFER A CHARACTER
        DEC     @SP             ;CHECK FOR COUNT DONE
        BGT     TRBYT           ;TRANSFER MORE
```

Refer to the *PDP–11 Processor Handbook* for your computer for more information on non-interrupt programmed I/O.

6.2 Interrupt-Driven I/O

Although programmed I/O is useful in a few situations, generally the best way to handle device I/O is through interrupt processing. According to this method, a program starts an I/O transfer but continues processing. When the transfer completes, the device issues an interrupt. An interrupt service routine then determines whether the transfer is incomplete, complete, or has encountered an error. It takes the appropriate action (restarting the transfer, returning to the program, or possibly retrying the transfer in case of error). The advantages of using interrupt-driven I/O are that it enables two or more processes to run concurrently and it does not monopolize system resources.

6.2.1 How an Interrupt Works

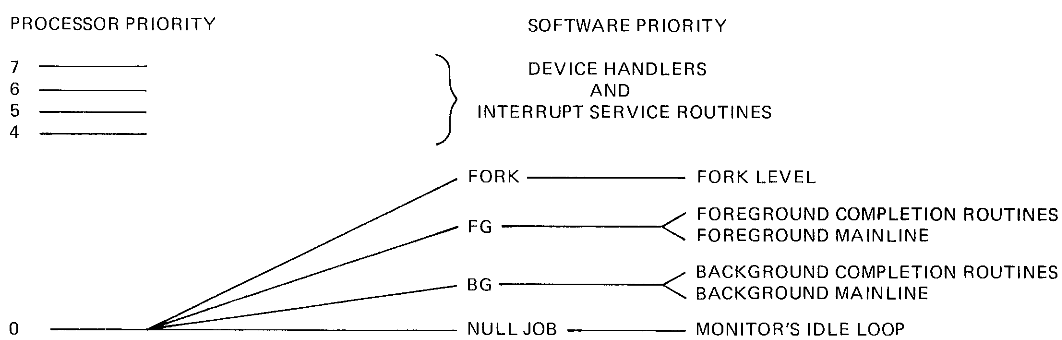
An interrupt is a forced transfer of program execution that occurs because of some external event, such as the completion of an I/O transfer. The state of the processor prior to the interrupt is saved on the stack so that processing can continue smoothly after the return from the interrupt. The processor saves the Processor Status word, or PS, which reflects the current machine state, and the Program Counter, or PC, which indicates the return address.

Next, the processor loads new contents for the PC and PS from two preassigned locations in low memory, called an *interrupt vector*. These words contain the address of the interrupt service routine and the new PS, which indicates the new processor priority. When the interrupt service routine completes, it executes an RTI instruction, which restores the old PS and PC from the stack, and execution resumes at the interrupted point in the original program.

6.2.2 Device and Processor Priorities

Interrupt processing is closely related to device and processor priorities. Figure 6–1 illustrates the RT–11 priority structure. Each device on the system has a priority assigned to it and devices that must be serviced as soon as possible after they interrupt have the highest priority. DECtape, for example, has priority 6; disks typically have priority 5; terminals and other character-oriented devices usually have priority 4. This priority system has been carefully designed and in general is adjustable through a pluggable priority selector on each I/O device interface. You can control the ordering of devices with the same priority. For these devices, the one closest to the CPU on the bus is serviced before other devices when interrupts occur simultaneously.

Figure 6–1: RT–11 Priority Structure



The central processor operates at any one of eight levels of priority, from 0 to 7. (The LSI processor is an exception; it operates at either 0 or 7.) When the CPU is operating at priority 7, no device can interrupt it with a request for service. When the CPU is operating at a lower priority, only a device with a higher priority can cause an interrupt. You can adjust the processor's priority from within an interrupt service routine by modifying the Processor

Status word. In an RT–11 system, software tools are provided to do this for you, so you never directly modify the PS yourself. The tools include the .MTPS and .MFPS programmed requests, and the .INTEN and .FORK macros.

The interrupt system allows the processor to continually compare its own priority with that of any interrupting devices and to acknowledge the device with the highest level above the processor's. This system can be nested — that is, the servicing of one interrupt can be left in order to service an interrupt with a higher priority. Service continues for the lower priority device when the higher priority device is finished.

See the *PDP–11 Processor Handbook* for your computer for more information on priorities and interrupts. See also the *Peripherals Handbook*, the *Microcomputer Handbook*, the *Terminals and Communications Handbook* and the *Memories and Peripherals Handbook*.

6.2.3 Processor Status (PS) Word

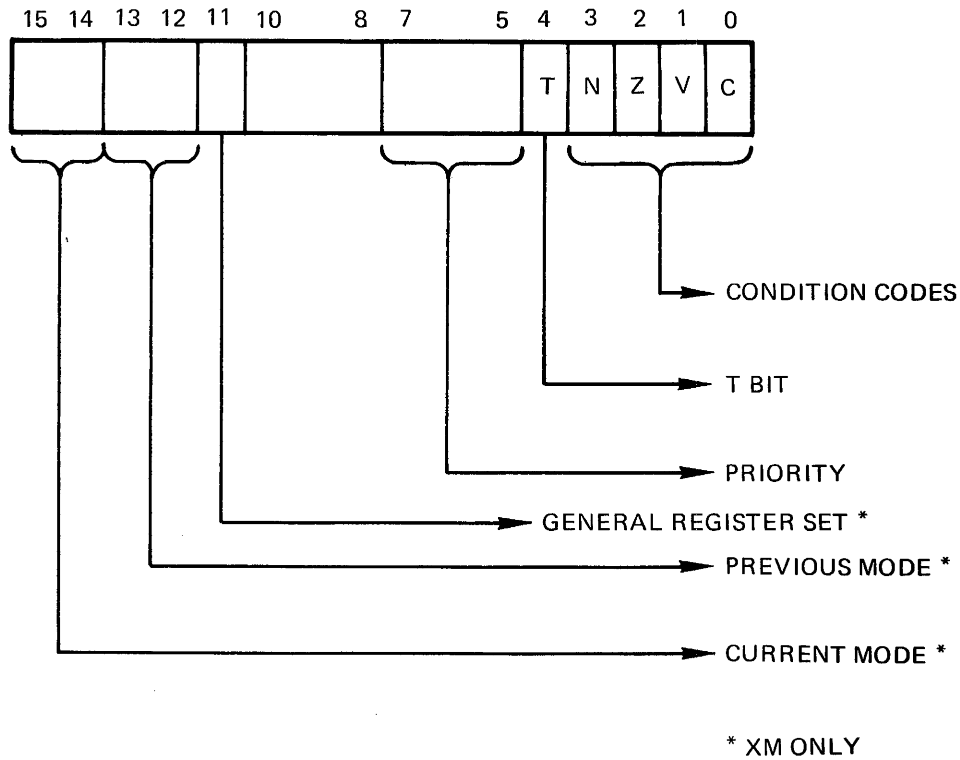
The Processor Status (PS) word occupies the highest address on the I/O page. (Again, the LSI processor is an exception; its PS is not addressable on the I/O page. The monitor refers to the PS by using the MTPS and MFPS instructions.) It contains information on the current status of the machine. This information includes the current processor priority, current and previous operational modes, the condition codes describing the results of the last instruction, and an indicator to cause the execution of an instruction to be trapped (used for program debugging).

Figure 6–2 illustrates the bits in the PS. Bits 5 through 7 determine the current processor priority. (In an LSI system, only bit 7 determines the priority; priority is either 0 or 7.) By changing bits, you alter the CPU's priority. You can change the priority to 7, for example, to prevent any more interrupts from occurring. When you are servicing a particular interrupt, you can change the processor priority to the priority of that device so that only devices with a higher priority will interrupt that service routine. (Specifically, the device you are servicing cannot interrupt.) In general, you need not access the PS yourself; use the macros provided in RT–11, such as .INTEN and .FORK, to change the processor priority.

6.3 In-Line Interrupt Service Routines Versus Device Handlers

Because both non-interrupt programmed I/O and interrupt-driven I/O are valid processes in an RT–11 system, when you need to interface a new device to your system — one that is not already supported by RT–11 — your first decision must be whether to use in-line interrupt service or to write a device handler for it. Whatever your decision, both interrupt service routines and device handlers can include non-interrupt programmed I/O sections as well as interrupt-driven code. The normal RT–11 interface between the monitor and a peripheral device is a device handler, which exists as a memory image

Figure 6–2: Processor Status (PS) Word



file on a mass storage device, and resides in memory when it is needed to perform device I/O (see Chapter 2). A device handler usually includes an interrupt service routine within it.

If you choose to use an interrupt service routine, you must place the routine within your program so that your program directly changes the status and buffer registers for a specific device, and it can service the interrupts within its own code. This means, of course, that the interrupt service code must always be resident in memory.

On the other hand, if you choose to use a device handler, the interrupt service code is contained within the handler, not in your program. You issue .READ and .WRITE programmed requests from your main program, and the monitor and the handler together initiate the data transfer, service the interrupts, and notify your program when the transaction is done. In an SJ system, or for a background job in FB, the handler must be resident only when your program actually needs it to perform I/O. (That is, the handler must be resident whenever a file or channel is open.) For foreground jobs and system jobs in an FB or XM system, you must load the handler (by using the monitor LOAD command) before you execute your program, so that the handler is always resident.

How you decide which method is more suitable for your new device depends largely on how you want the device to appear to system and application programs. In general, you should use in-line interrupt service for sensor or control devices, such as analog-to-digital converters. You should service devices that appear to be block-replaceable, file-structured mass storage devices, such as disks and diskettes, through device handlers. You can service most communications hardware by either method; the decision rests on other criteria.

The two major advantages of in-line interrupt service routines are their speed and the amount of control information they provide. Because there is no monitor overhead involved in a data transfer, an in-line routine can often handle interrupts faster than a device handler can. If the speed of servicing interrupts is crucial to your application, you may choose to write an in-line interrupt service routine even if the device is a disk.

An in-line routine has access to all the device control and status registers for a device, as well as its data buffer registers. (Of course, a device handler has access to all the same registers, but the program using the handler does not.) It can pass a lot of information to the program. This provides a great deal of flexibility in the way the program calls the interrupt service routine, and in the amount of information the routine returns to it.

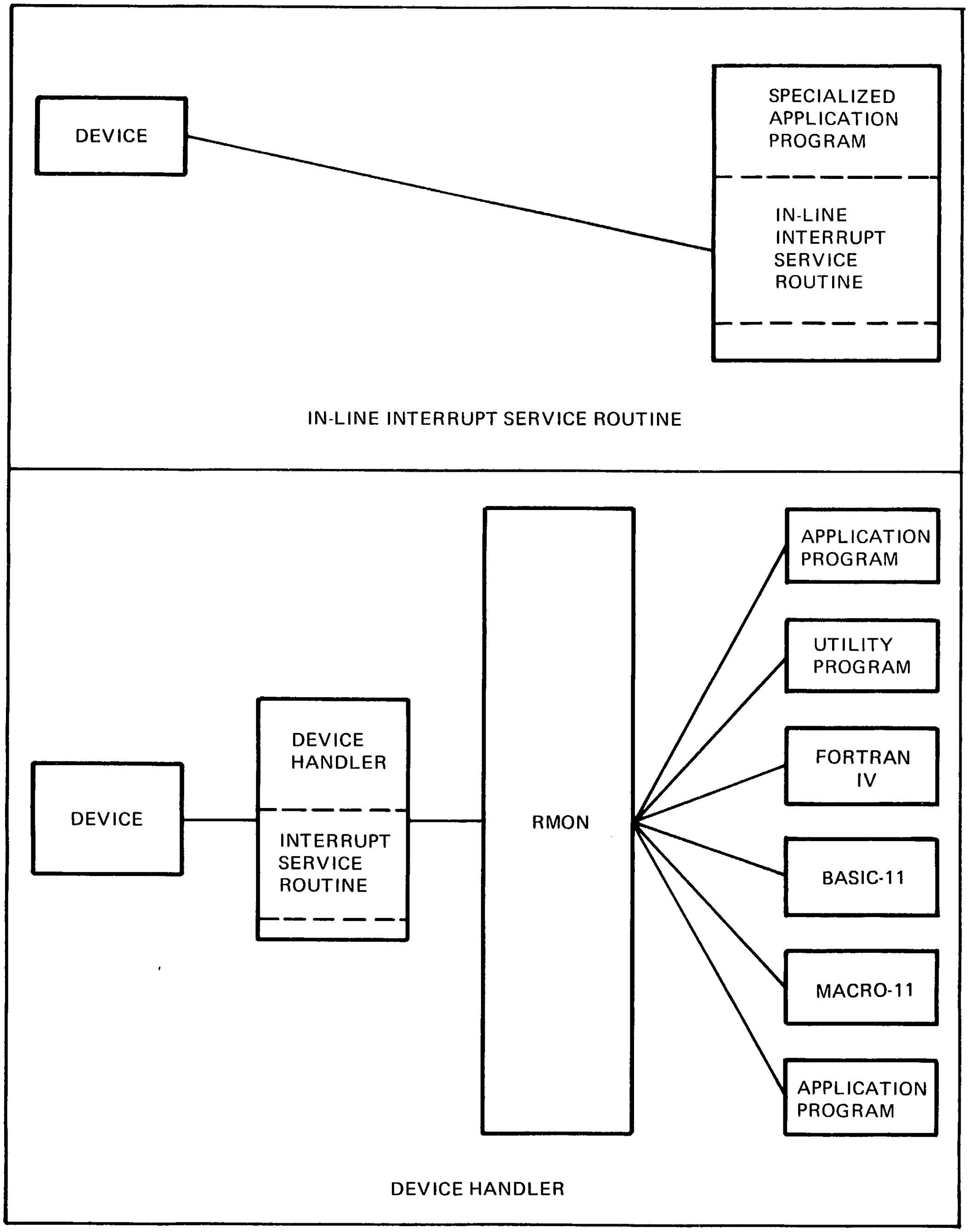
The three major advantages of using device handlers are that they provide device independence for your programs, they can share processor time with other processes, and they are simple to use. Device handlers have a standard protocol for interfacing to the RT–11 monitor. There is also a standard protocol for the interface between the monitor and a program, so that any program that conforms to the monitor standards can use the handler. This includes application programs, system utility programs, and RT–11 language processors such as MACRO–11, FORTRAN IV, and BASIC–11. Thus, the device handler makes a new device available to a large number of programs without any special modification. (In addition, a device handler for a random-access device makes the RT–11 file system available on the device at no extra cost.) In contrast, an in-line interrupt service routine makes the new device available to just one application program.

Device handlers are easy to use. Because they are the standard RT–11 means of handling device I/O, the procedure for writing them and using them is clear and straightforward. This procedure is simplified further by the fact that RT–11 provides macros to write a handler; there are also keyboard monitor commands that install handlers into the monitor device tables and load them into memory. In addition, a device handler permits you to take advantage of the monitor programmed requests for performing data transfers. Finally, a device handler is the only way you can interface a device to a virtual job in an XM system.

Figure 6–3 highlights some differences between in-line interrupt service routines and device handlers.

If you decide that your new device requires an in-line interrupt service routine, read the rest of this chapter to learn how to plan and write one. If you decide that a device handler is more suitable, read the rest of this chapter and then go on to Chapter 7 to learn how to plan, write, and debug a handler.

Figure 6–3: In-Line Interrupt Service Routines and Device Handlers



6.4 How to Plan an Interrupt Service Routine

The most important part of writing an in-line interrupt service routine is taking the time to plan carefully. Follow these guidelines:

- Get to know your device
- Study the structure of an interrupt service routine
- Study the skeleton interrupt service routine
- Think about the requirements of your program
- Prepare a flowchart of your program
- Write the code
- Test and debug the program

6.4.1 Get to Know Your Device

Getting to know your new device is crucial to writing an interrupt service routine that works correctly. If your device is a DIGITAL peripheral, consult the hardware reference manual for that device. You can also learn a lot from the *PDP–11 Peripherals Handbook*. If your device is not from DIGITAL, study the documentation for it carefully. Regardless of the format of the documentation (whether it is a manual, a brochure, or a set of engineering prints), it should contain the vital information you need to support it on a PDP–11 system. Be sure you obtain this information.

In any case, you must understand how the device operates: what it needs from you, and how it handles data transfers. Use the following checklist to make sure you have enough device-specific information to write the service routine. Do not attempt to write any code until you have considered each question.

Some of the following questions do not apply to all types of devices. Some are for mass storage devices, some are more appropriate for sensor devices or communications devices. Consider each question carefully, though, to see if it applies to your device.

- What is the interrupt vector (or vectors) for the device?

  Decide what the interrupt vector should be. Consider both conflicts with existing RT–11-supported devices and also conflicts with devices supported by other PDP–11 operating systems, if you use those systems. Once you decide on the vector, make sure the device is installed properly and that the hardware is jumpered to that address. RT–11 requires all vectors to be below location 500 and some low-memory locations are not available for use as vectors. Chapter 2 lists the current PDP–11 vector assignments.

- What are the control and status registers?

  Learn where these registers are located and what the bits in each mean.

- What is the priority for the device?
- Is the device DMA (Direct Memory Access) or programmed transfer (word- or character-oriented)?
- What are the data buffer registers?

  Learn where these registers are located and what the bits in each mean.
- What are the op codes for typical operations?

  Learn how to initiate the various operations by manipulating the bits in the device registers. Device handlers tend to perform read, write, seek, and reset operations.
- When does the device interrupt?

  Some devices interrupt for each character; others are word-oriented, block-oriented, or packet-oriented. Some devices interrupt twice for certain operations, such as seek or drive reset. Find out if your device does this, and plan now to take this information into account later.
- What is the basic unit for data transfers?

  This relates to the previous question, of course, but you must determine whether to send I/O requests to the device as byte, word, or block counts. If, for example, your program deals in terms of words and the device is character-oriented, you may have to convert the word count to a byte count in the service routine.
- Does the device want a positive or negative byte count?

  Some devices require a negative byte or word count. If your device is one of those, you may need to negate the count in the service routine.
- What is the device structure, or geometry?

  If the device is a disk, find out how the cylinders, tracks, and sectors are structured. Determine their size. Find out if the device requires interleaving, and, if so, learn how to optimize for speed. (*Interleaving* describes the process for writing data to a spinning device that requires program intervention between sectors. The disk is constantly moving; data is written into one sector, the program intervenes as the adjacent sector spins past, then more data is written into the next available sector.)
- What is the buffering arrangement? .

  Some devices transfer data to your program one character at a time. Others buffer data internally in a silo, or send it in packets. Decide how to buffer the data in your program. Make sure the buffer space you allocate is large enough.
- How do you calculate the address of the data on the device?

  This relates to the device's structure. Study the device now and determine how to find the data you want on it. Note that RT–11 block numbers must be converted to device-specific addresses. Note also that some processors have no multiply or divide instructions.

- What "housekeeping" operations does the device require?

  Some devices require a drive reset before a retry. Others require that the device be selected or that a disk pack be acknowledged before you can perform any operations on it. You must do a drive reset after a seek incomplete or a drive error, for example.

- How will you handle errors and exception conditions?

  First you must decide which errors are hard and will abort the transfer, and which errors are soft and will retry the transfer. Some typical soft errors include checksum errors, data late errors, and timing errors. Decide how many times you will retry the transfer for soft errors, and how you will handle a hard error condition.

- What are the abort considerations?

  Consider whether the device is relatively fast or slow. Keep in mind that you do not want to issue a controller reset if only one unit of a two-unit controller is affected by a program's abort because this can interfere with the operation of the second unit. Similar considerations may apply to dual-ported devices.

6.4.2 Study the Structure of an Interrupt Service Routine

Section 6.5 describes the structure of an interrupt service routine. Read this section carefully. Appendix C contains a sample application program that does in-line interrupt service. Read that program, too, and study its structure.

6.4.3 Study the Skeleton Interrupt Service Routine

Section 6.6 contains a skeleton outline of a foreground job with an in-line interrupt service routine. Study this outline to be sure you understand the flow of execution.

6.4.4 Think About the Requirements of Your Program

Remember that the interrupt service routine is part of your program and decide where to place it in the program. Review the material in Chapter 2 on swapping the USR. If you plan to execute your program in an XM system, read Section 6.7 for XM considerations.

6.4.5 Prepare a Flowchart of Your Program

Many experienced programmers prepare flowcharts after all their programs are written, or they omit them entirely. However, flowcharting a system with the complexities of interrupt service can help you find loose ends and point out errors in your logic. Flowcharts are not much help, unfortunately, in pointing out potential race conditions. (A race condition is a situation in which two or more 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. It may be caused by a device interrupting while its interrupt service routine is running, due to improper processor priority.) When you design your program, examine every step carefully; keep in mind what would happen if an interrupt occurred at each instruction. This kind of planning can help you avoid race conditions later.

Spend enough time to design a clean and straightforward way of handling error conditions; if your program can handle error conditions well, you will probably find that the rest of your program design works well too.

6.4.6 Write the Code

If you have followed the recommended steps so far, writing the code for the interrupt service routine itself should be relatively simple. You can borrow as much code as possible from other interrupt service routines you have studied. Start with a general outline, then add details to reflect the specifics of your particular device. When you are satisfied with the code, have checked it thoroughly for logic errors, and it assembles properly, you are ready to test and debug it.

6.4.7 Test and Debug the Program

The only way to test a program with in-line interrupt service is to try executing it. If the program is operating correctly, it should be able to read or write data accurately, should not lose any data, and should handle error conditions properly. Try executing the program in a test situation with data you have prepared. If you find errors, link the program with ODT (not VDT) and try running it step by step. Make coding corrections, reassemble the program, and retry it as necessary.

6.5 Structure of an Interrupt Service Routine

The following sections outline the general structure of an in-line interrupt service routine. Read them carefully and determine which items apply to your own situation.

6.5.1 Protecting Vectors: .PROTECT

In FB or XM systems where more than one job can be running, you should use the .PROTECT programmed request to protect an interrupt vector before you move a value to it. This process makes sure that the vector does not already belong to the monitor or to another job. It gives ownership of the vector to your job, and protects it from interference from another job or the monitor by setting bits in the monitor bitmap. (Chapter 3 describes the low-memory bitmap in detail.) Your job should abort immediately if the .PROTECT request fails; your job must not access a vector that is already in use. See Sections 6.5.2 and 6.6 for examples of how to use .PROTECT.

See the *RT–11 Programmer's Reference Manual* for the format of the .PROTECT programmed request.

Even though the .PROTECT request has no meaning in an SJ system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet using it simplifies conversion later if your program needs to run in an FB environment.

6.5.2 Setting Up the Interrupt Vector

Your program must take care of moving the address of your interrupt service routine to the first word of the interrupt vector. RT–11 requires all interrupts to raise the processor priority to 7, so your program must fill in the second word of the interrupt vector with 7 as the new priority. The following lines of code show a typical way for a program to set up the two-word interrupt vector. Note that a program should not set up a vector until the vector is protected. For this example, assume the device name is XX, and the interrupt vector is at 220 and 222.

```
        XXVEC = 220                   ;DEFINE THE VECTOR
        PR7 =   340                   ;PRIORITY 7 = 340
;
; The entry point for the interrupt service routine is ISREP:
;
        .PROTECT #AREA,#XXVEC         ;PROTECT THE VECTOR
        BCS     NOVEC                 ;VECTOR IN USE
        MOV     #ISREP,@#XXVEC        ;SET UP FIRST WORD
        MOV     #PR7,@#XXVEC+2        ;SET UP SECOND WORD
```

6.5.3 Stopping Cleanly: .DEVICE

The .DEVICE programmed request is meaningful only in FB and XM systems. Its purpose is to turn off a device (by clearing its interrupt enable bit) if its associated program is aborted with CTRL/C, or when the program exits. (See the *RT–11 Programmer's Reference Manual* for the format of the .DEVICE programmed request. See Section 6.6 of this manual for an example using .DEVICE.)

This request is not required in an SJ environment. However, even though the request has no meaning in an SJ system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet using it simplifies conversion later if your program needs to run in an FB environment.

When a program in SJ exits, the monitor waits for all I/O to finish if there is an active queue element outstanding. In FB, when a program exits, the monitor not only waits if there is an active queue element outstanding, but in addition, it enters the device handler at its abort entry point. If a job is aborted with CTRL/C, or if it issues a .HRESET request, the SJ monitor executes a hardware reset to stop I/O on all devices. If you are designing the hardware for your device, make sure that it stops cleanly when it receives the bus-initialize signal.

6.5.4 Lowering Processor Priority: .INTEN

When an interrupt occurs, control passes to your interrupt service routine entry point, the address you supplied as the first word of the interrupt vector. At this point, the processor priority is 7, and all other interrupts are prohibited. If you need to do anything with all interrupts disabled, this is where the code belongs. It should be as short and efficient as possible and should not destroy the contents of any registers. If this code needs to use registers, it must save them and restore them before issuing the .INTEN call. If the code executed at priority 7 is too long, system interrupt latency (a measure of how quickly the system can respond to an interrupt) will suffer. A good guideline is to spend no more than 50 microseconds at priority 7.

You should lower the processor priority to that of the device as soon as possible. This means that only devices with a higher priority than this one will be able to interrupt its service routine. To lower the priority, use the .INTEN programmed request. The stack pointer and general registers R0 through R5 must contain the same values when your interrupt service routine issues the .INTEN request as they did at the interrupt entry point. If your interrupt service routine is not written in Position-Independent Code (PIC), use the following format:

.INTEN prio

The .INTEN call generates the following code:

```
JSR     R5,@54
.WORD   ^C<PRIO*40>&340
```

If your interrupt service routine is written in PIC, use the .INTEN call with a second argument, *PIC*. (The argument can actually be anything at all, as long as it is not blank.)

.INTEN prio,PIC

The second format generates Position-Independent Code:

```
MOV     @#54,-(SP)
JSR     R5,@(SP)+
.WORD   ^C<PRIO*40>&340
```

Both formats cause a JSR to the monitor's INTEN routine, which lowers the processor priority, and, in FB and XM, switches to system state. The monitor then calls the interrupt service routine back as a co-routine. R4 and R5 are available for use on return from the call. You must not destroy the contents of any other registers. If you need R0 through R3, save them on the stack or in memory and restore them before you exit. If you need to preserve values across the .INTEN request, you must save them in memory before the call and restore them after it. Likewise, if the contents of the PS are important, such as the values of the condition bits, you should save them before issuing the .INTEN call. Since .INTEN causes a switch to the system stack in FB and XM, you should avoid using the stack excessively once you are in your interrupt service routine. Save and restore registers and the PS, as necessary, by using memory locations instead of the stack.

NOTE

Saving values in memory locations may prevent your interrupt routine from being re-entrant. If you intend to use the routine for multiple devices, be careful about re-entrancy when you design it.

(See the *RT–11 Programmer's Reference Manual* for more information on .INTEN. See Section 6.6 of this chapter for an example using .INTEN. See Section 6.5.7 for a summary of the interrupt service routine macro calls.)

6.5.5 Issuing Programmed Requests: .SYNCH

The .SYNCH call is useful mainly in the FB and XM environments. Its purpose is to make sure that the correct job is running when an interrupt service routine executes a programmed request. Even though the .SYNCH call has no meaning in an SJ system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet using it simplifies conversion later if your program needs to run in an FB environment. (For the complete expansion of this macro, see the listing of the system macro library in the *RT–11 Programmer's Reference Manual.*) See the *RT–11 Programmer's Reference Manual* for the format of the .SYNCH request.

If you need to issue one or more RT–11 programmed requests from the interrupt service routine, you must first issue the .SYNCH call. Remember that the .INTEN call switched execution to system state, and programmed requests can only be made in user state. The .SYNCH call itself handles the switch back to user state. Note that you should never issue programmed requests requiring the USR from within an interrupt service routine, even after using .SYNCH. You can also issue .SYNCH after .FORK, which is covered in Section 6.5.6. When you issue the .SYNCH call, R0 through R3 and the stack pointer must contain the same values as they did when the .INTEN request returned to you.

Table 6–1 illustrates the format of the synch block, which acts like a completion queue element. The information in the seven-word synch block is placed at the head of the appropriate job's completion queue. Therefore, the code following the .SYNCH request executes as a completion routine, in user state, at priority 0. Because of this, your program must either disable interrupts before the .SYNCH call, or it must be prepared for the device to interrupt again before the .SYNCH code executes. The synch block is available for reuse when Q.COMP (offset 14 octal) is 0. You can test the synch block easily by issuing another .SYNCH. If control passes to the error return (the word following the .SYNCH call), the block is still in use.

Table 6–1: Synch Block

Offset	Name	Agent	Contents
0	Q.LINK	–	Reserved
2	Q.CSW	user	Job number
4	Q.BLKN	–	Reserved
6	Q.FUNC	–	Reserved
10	Q.BUFF	user	R0 argument to pass
12	Q.WCNT	monitor	–1
14	Q.COMP	user	Assemble a value of 0 here; the monitor then maintains the contents of this word

In general, a long time can elapse between the .SYNCH call and the return. First, the monitor switches to user state, and a scheduling pass is required to determine whether or not a context switch is also necessary. Then a background completion routine may have to wait for a compute-bound foreground job to become blocked. So, it may take a considerable amount of time before the code following the .SYNCH actually executes.

In the code following the .SYNCH call, R0 and R1 are free for use, as they are in any completion routine. However, you must preserve R2 through R5 if your .SYNCH routine uses them. This poses a problem for R4 and R5, which are not preserved across the call. If their contents are important, save them in memory before the .SYNCH call. You can use Q.BUFF in the synch block to pass a value into R0 for the synch routine.

The .SYNCH call has an unusual error return. The first word after .SYNCH is the return address on error; the second word after .SYNCH is the return on success. See Section 6.6 for an example using .SYNCH. See Section 6.5.7 for a summary of the interrupt service routine macro calls.

In the SJ environment, routines following .SYNCH calls (and, in fact, completion routines in general) are nested (that is, they can interrupt each other). They are serialized in FB and XM. In SJ, the .SYNCH mechanism simulates the FB and XM scheme but does not duplicate it.

6.5.6 Running at Fork Level: .FORK

The .FORK programmed request gives you another way to lower the processor priority. (See the *RT–11 Programmer's Reference Manual* for the format of the .FORK programmed request. For the complete expansion of this macro, see the listing of the system macro library in that manual.)

When you issue a .FORK call, the fork block is added to a fork queue, which is a first-in, first-out list. Fork routines (all the code following a .FORK call) execute in system state at priority 0, after all interrupts have been serviced, but before the monitor switches to user state. Context switching is inhibited as well during the time fork routines are executing. (See Figure 6–1 for a review of RT–11 priority levels.)

R4 and R5 are preserved across the .FORK call. In addition, R0 through R3 are free for use after the call. Like .SYNCH, the .FORK call assumes you have not changed R0 through R3 or the stack since the .INTEN call returned to you. See Section 6.5.7 for a summary of the interrupt service routine macro calls. Note that you cannot issue .FORK without a prior .INTEN call.

You must provide a four-word block of memory for the fork queue element, the last three words of which will contain R4, R5, and the return PC. The first word is a link word, which must be 0 when you issue the .FORK request. Because a .FORK routine should not be re-entrant, make sure that the device cannot interrupt between the time you issue the .FORK call and the time the .FORK routine (the code following the call) begins to execute.

You may not re-use a fork block until the fork routine has been entered. It is safe to assume that the fork block is free when the call that used it returns. See Table 6–2 for an illustration of the fork block.

Table 6–2: Fork Block

Offset	Name	Agent	Contents
0	F.BLNK	monitor	Link word
2	F.BADR	monitor	FORK routine address
4	F.BR5	monitor	R5 save area
6	F.BR4	monitor	R4 save area

Generally, .FORK is used in device handlers. To use it in an interrupt service routine, you must first set up a pointer, called $FKPTR. The recommended way to do this in a main program is as follows:

```
          MOV     @#54,R4
          ADD     402(R4),R4
          MOV     R4,$FKPTR
           .
           .
           .
$FKPTR:   .WORD   0
XXFBLK:   .WORD   0,0,0,0
```

Then, in the interrupt service routine, you can use the normal form of the .FORK macro:

.FORK XXFBLK

The .FORK macro expands as follows:

```
JSR     R5,@$FKPTR
.WORD   XXFBLK-.
```

In an SJ system, there is no real support for .FORK unless you select timer support as a special feature at system generation time. Instead, the monitor simulates the process by saving registers R0 through R3 before calling the interrupt service routine back. Beyond that, it does not attempt to serialize fork routines. Note that in your interrupt service routine, no registers are free for use before the .INTEN call. After the .INTEN, you can safely use R4 and R5. See Section 6.5.7 for a summary of the interrupt service routine macro calls.

The .FORK request has several applications in a real-time environment because it permits lengthy but noncritical interrupt processing to be postponed until all other interrupts are dismissed.

For example, the CR11 card reader handler internally buffers 80 columns of data. It receives an interrupt once per column, and translates and moves the character into its internal buffer at interrupt level. It then moves its internal buffer to the user buffer, a process that can take up to 2.5 msec. In RT–11 Version 2C, this process took place at priority level 6, which meant that interrupts at this priority and lower could be locked out for this time. This can cause data late errors on communications devices when the card reader is active at the same time.

This problem is not solved by simply dropping priority to 0, since the card reader could have interrupted a lower-priority device. Lowering priority causes problems in the other device handlers that are re-entrant. Using a .SYNCH does not always solve the problem, either, since the SJ monitor only simulates a .SYNCH and drops priority to 0, which produces the same problems for re-entrant handlers. The FB monitor must perform a context switch since .SYNCH returns to the caller in user context, running on the user stack. The context switch is a lengthy process and does not occur at all if there is a compute-bound foreground job.

The .FORK request is the solution to the problem. It returns at priority 0, but only when all other interrupts have been dismissed and before control is returned to the interrupted user program. (Note that you dismiss an interrupt when you leave interrupt level, by any one of several means.)

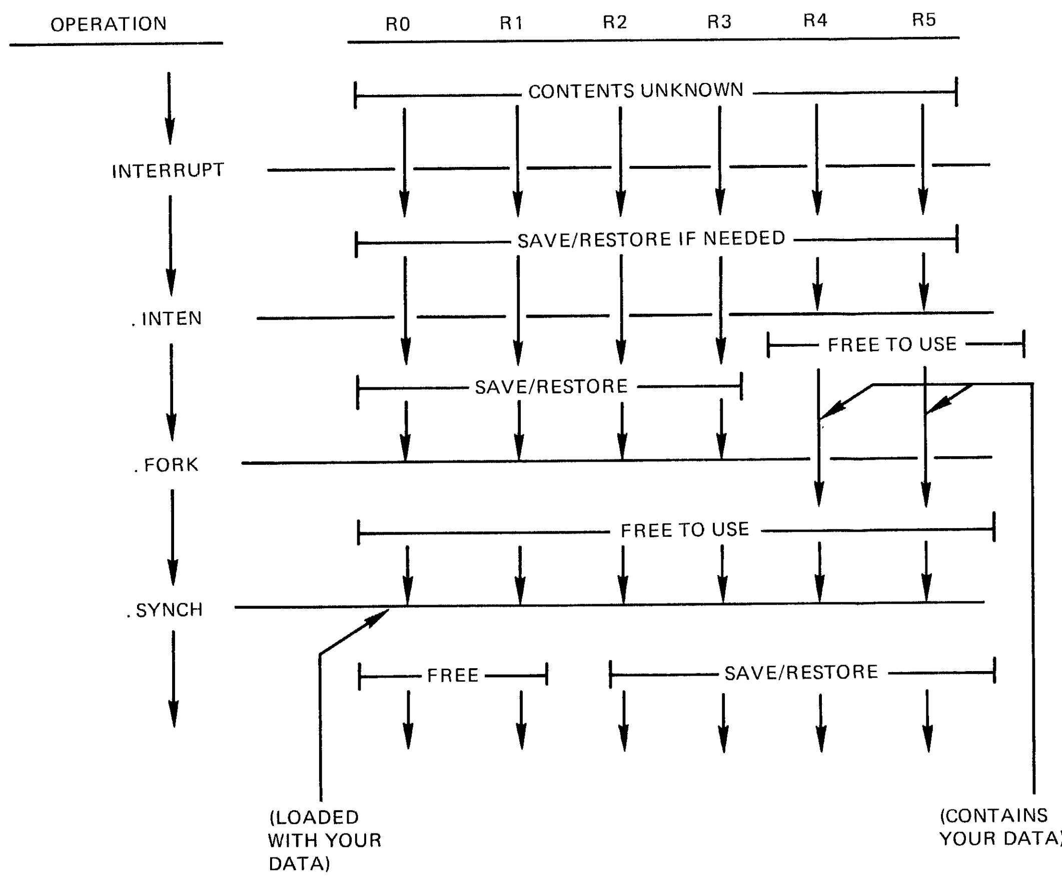
6.5.7 Summary of .INTEN, .FORK, and .SYNCH Action

Table 6–3 summarizes the effects of the .INTEN, .FORK, and .SYNCH macro calls. Figure 6–4 describes the status of the registers for each call.

Table 6–3: Summary of Interrupt Service Routine Macro Calls

Macro Call	New Priority	New Stack	Registers Available to Use After Call	Your Data Preserved Across Call In
.INTEN	device's	System	R4, R5	none
.FORK	0	System	R0-R5	R4, R5
.SYNCH	0	User	R0, R1	R0

Figure 6–4: Summary of Registers in Interrupt Service Routine Macro Calls



6.5.8 Exiting from Interrupt Service: RTS PC

The .INTEN request causes the monitor to call your interrupt service routine as a co-routine. At the end of your routine, when it is time to exit, use an RTS PC instruction. This returns control to the monitor, which restores R4 and R5 and then executes an RTI instruction.

You also exit from .FORK and .SYNCH routines with an RTS PC instruction. Be sure that the stack is the same as it was upon entry, and that any registers that must be preserved have their original contents.

6.5.9 Servicing Interrupts in FORTRAN: INTSET

The INTSET function is available in RT–11 to establish a FORTRAN subroutine that will be initiated via interrupt and that will run at interrupt level. See the SYSLIB routines in the *RT–11 Programmer's Reference Manual* for a more complete description of INTSET.

6.6 Skeleton Outline of an Interrupt Service Routine

Figure 6–5 shows a foreground main program that contains an in-line interrupt service routine. The foreground program performs some initialization tasks and then suspends itself. When data is available from a peripheral device the interrupt service routine collects it. When all the data is gathered, the interrupt service routine resumes the main program, which can then process the new information before suspending itself again. The main program's processing could involve some manipulation of the new data or it could be writing the data to a file shared by a background data analysis job. Note that because this example forces the job number to 2, it cannot execute properly in a system with the system job feature present.

For this example, *xx* represents the device name.

6.7 Interrupt Service Routines in XM Systems

If you are not planning to execute your program in an XM environment, you need not read this section.

Of the two kinds of jobs in an XM environment, virtual jobs and privileged jobs, virtual jobs cannot contain in-line interrupt service routines (see Chapter 4). By the very definition of virtual mapping, virtual jobs cannot access the device I/O page. Therefore, they cannot set a device's interrupt enable bit or move data to or from a device's data buffer register.

If a job containing an in-line interrupt service routine must run in the XM environment, it must run as a privileged job. Privileged mapping makes the low 28K words of memory and the I/O page available to the program and permits the program to map portions of the user virtual address space into extended physical memory if the program requires it.

In order to understand the restrictions that the XM environment imposes on interrupt service routines, you must understand that when an interrupt occurs in XM, its service routine executes with kernel, not user, mapping. This means that whether or not the program has mapped some of its virtual address space into extended memory, the interrupt service routine executes with the default kernel mapping to the low 28K words of memory plus the I/O page. It makes sense, therefore, that the first XM restriction demands that the mapping for your interrupt service routine plus any data it uses must be identical to kernel mapping at any time that an interrupt could occur.

Figure 6–5: Skeleton Interrupt Service Routine

```
** MAIN PROGRAM **

        xxVEC = vvv                             ;THE DEVICE VECTOR
        PR7 =   340                             ;PRIORITY 7
        DEVPRI= 5                               ;DEVICE PRIORITY = 5
                                                ;(0-7, NOT 000-340)
        xxCSR = nnnnnn                          ;THE DEVICE CONTROL REGISTER
        IENABL =100                             ;INTERRUPT ENABLE BIT

START:  .PROTECT #LIST,#xxVEC                   ;PROTECT THE VECTOR
        BCS     ERROR                           ;HANDLE .PROTECT ERROR
        MOV     #ISREP,@#xxVEC                  ;SET UP FIRST WORD
                                                ;OF VECTOR
        MOV     #PR7,@#xxVEC+2                  ;SET UP SECOND WORD
                                                ;OF VECTOR
        .DEVICE #LIST,#DEVLST                   ;TO DISABLE DEVICE ON

                                                ;EXIT OR ABORT
;
; Lines of code here initialize input buffers in the service routine;
; initialize other pointers and flags
;
SPND:   BIS     #IENABL,@#xxCSR                 ;ENABLE INTERRUPTS
        .SPND                                   ;WAIT UNTIL THERE IS SOME DATA
;
; Lines of code here store the data;
; reset some flags
;
        BR      SPND                            ;WAIT FOR MORE DATA
DEVLST: .WORD   xxCSR                           ;LIST FOR .DEVICE
        .WORD   0
        .WORD   0
LIST:   .BLKW   3                               ;EMT ARG BLOCK
ERROR:  .                                       ;ROUTINES TO HANDLE ERRORS
        .
        .

                ** INTERRUPT SERVICE ROUTINE **

ISREP:  .                                       ;THE INTERRUPT ENTRY POINT;
        .                                       ;PRIORITY IS 7
        .
        .INTEN  DEVPRI                          ;NOTE: NOT #DEVPRI.
                                                ;LOWER TO DEVICE PRIORITY;
                                                ;WE ARE IN SYSTEM STATE
                                                ;WITH R4 AND R5 AVAILABLE.
;
; If there is more data to collect:
;
        BR      RETURN
;
; If there is no more data to collect:
;
        .SYNCH  #SYNBLK                         ;GO BACK TO MAIN PROGRAM
                                                ;TO PROCESS DATA.
        BR      SYNERR                          ;SYNCH RETURNS HERE ON ERROR
        .RSUM                                   ;WAKE UP MAIN PROGRAM
RETURN: RTS     PC                              ;WAIT FOR ANOTHER INTERRUPT
        .
        .
        .
SYNBLK: .WORD   0,2,0,0,0,-1,0                  ;NOTE: 2 IS THE JOB NUMBER
                                                ;FOR THE FOREGROUND JOB.
SYNERR:                                         ;PROCESS SYNCH ERROR
```

Figure 6–6 shows the default kernel mapping scheme, which provides access to the low 28K words of memory plus the I/O page. This is also the mapping scheme for a privileged job when it first begins execution. And, this is the mapping scheme that takes effect whenever an interrupt is serviced. (The shaded areas in the figure represent memory that the user job cannot access.) In Figure 6–6, the interrupt vector at 200 and 202 contains the entry point, called ISREP:, of the interrupt service routine, and the value 340, which represents the new PS. When an interrupt occurs, the system uses kernel mapping to locate the interrupt service routine. In this example, it should start at address 120000. Since privileged mapping and kernel mapping are identical in this diagram, the interrupt service routine is located in physical memory exactly where the kernel mapping points, so it can execute correctly.

Figure 6–6: Kernel and Privileged Mapping

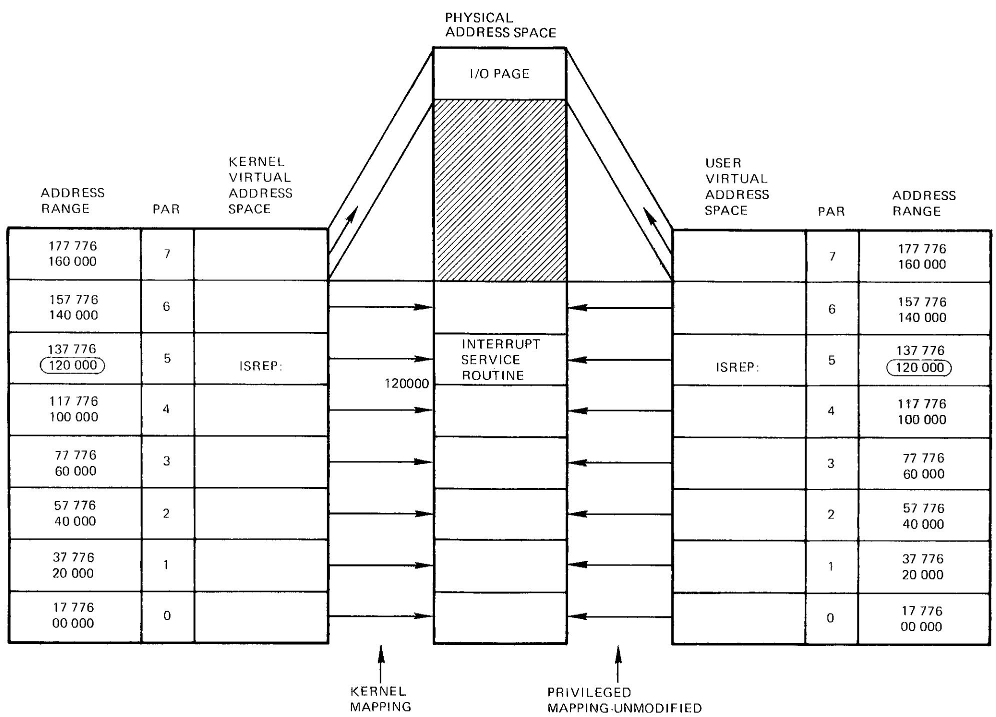
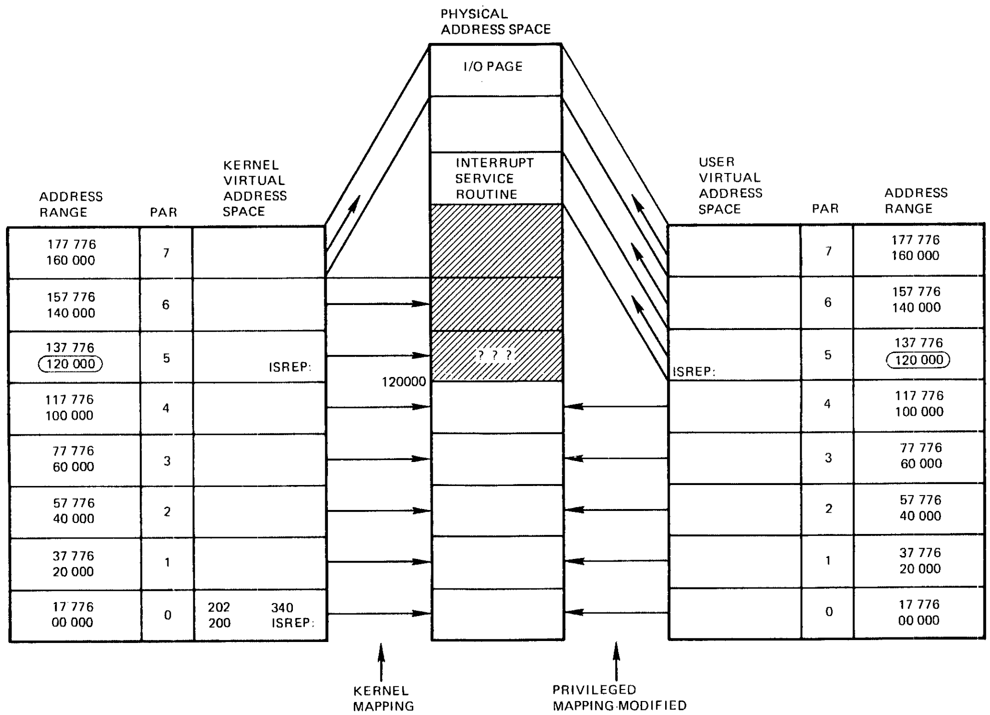


Figure 6–7 shows a privileged job that changes the user virtual address mapping. (The shaded areas in the figure represent memory that the user job cannot access.) You can see from the example that the interrupt service routine cannot execute correctly when an interrupt occurs because the interrupt service routine is not located in physical memory where it should be. The memory area pointed to by the kernel mapping contains random data or instructions.

Figure 6–7: Interrupt Service Routine Mapping Error



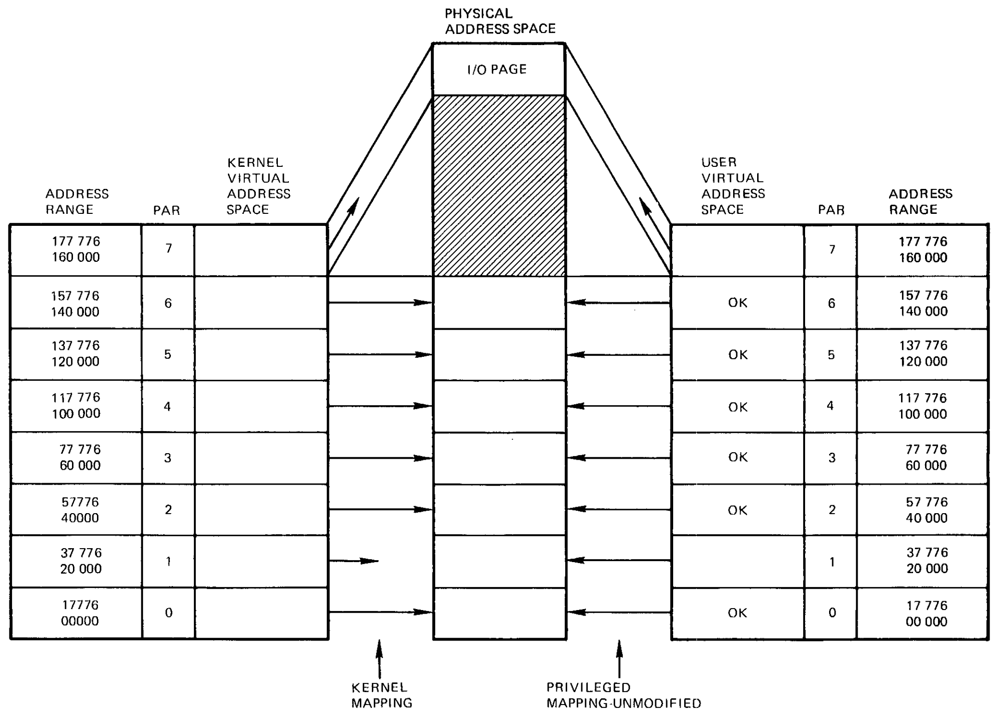
The second restriction for interrupt service routines in XM relates to the way the monitor uses Page Address Register (PAR) 1 with kernel mapping. PAR1 controls the mapping for virtual addresses 20000 through 37776. When XM is first bootstrapped with kernel mapping, the virtual addresses map directly to the same physical addresses. However, the monitor itself uses PAR1 to map to EMT area blocks and to user data buffers. So, whenever the system is running, the kernel virtual addresses in the PAR1 range can be mapped just about anywhere in physical memory and you have no way of controlling it. You must be sure that your interrupt service routine and any data it needs are not located in the virtual address range mapped by PAR1. Figure 6–8 illustrates this restriction. Valid locations for interrupt service routines, assuming that privileged mapping is identical to kernel mapping at the time of the interrupt, are marked on the diagram as "OK".

If your interrupt service routine needs a window into memory, it can borrow PAR1 the same way the monitor does. It must save the contents, set the value it needs, and restore the original contents before exiting. It can do this at .INTEN or fork level, but not at synch level.

NOTE

> If your system uses the MQ handler to communicate among system jobs and you have defined the conditional assembly parameter MQH$P2=1 during system generation, all the restrictions for PAR1 also apply to PAR2 – the range of addresses from 40000 through 57777.

Figure 6–8: PAR1 Restriction for Interrupt Service Routines



One final piece of information is important if you use .SYNCH in your interrupt service routine. The lines of code following .SYNCH execute almost like a completion routine. Completion routines in XM execute with the user registers, the user stack, and with user mapping. But, since the code following .SYNCH is still part of an interrupt service routine, it executes with the user registers, but with *kernel* mapping. So, the code following a .SYNCH call in XM must observe the same restriction as the main body of the service routine: its mapping must be identical to kernel mapping at any time that an interrupt could occur, or any time the completion routine could be executing. Of course, it must observe the PAR1 and PAR2 restrictions as well.

Chapter 7
Device Handlers

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. The structure of a standard handler 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 time-out support, special functions, error logging, and special services available in XM 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 read Chapter 6, Interrupt Service Routines, before you read about device handlers. Section 6.3 of that chapter can help you decide whether you need to write an in-line interrupt service routine or a device handler.

7.1 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

7.1.1 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 material in Section 6.4.1 so that you can answer all the pertinent questions about your device before you attempt to write a handler for it.

7.1.2 Study the Structure of a Standard Device Handler

Section 7.2 describes the structure of a standard device handler. Read this section carefully; your handler must conform to this structure.

7.1.3 Study the Skeleton Device Handler

Section 7.3 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.

7.1.4 Think About Using the Special Features

Sections 7.4 through 7.9 describe the special features available to device handlers. Read these sections carefully to determine whether any of the features are applicable to your handler.

7.1.5 Study the Sample Handlers

Appendix A contains assembly listings of three RT–11 device handlers (RK, DX, and PC) 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.

7.1.6 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.

7.1.7 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 7.2 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.

7.1.8 Install, Test, and Debug the Handler

Sections 7.11 and 7.12 show how to install a new device handler and how to begin testing and debugging it.

7.2 Structure of a Device Handler

An RT–11 device handler consists of the following six sections:

- Preamble
- Header
- I/O initiation
- Interrupt service
- I/O completion
- Handler termination

Each section is a separate logical unit, containing code for a particular purpose. Because the RT–11 system macro library provides special macros to generate much of the required code for these sections, the actual lines of code that you write yourself are not too complex.

Before you read ahead, take a minute to glance over the sample device handlers in Appendix A and get a feel for the overall structure of the handlers. Also refer to Figure 7–12, which illustrates the layout of the .SYS image of a device handler.

7.2.1 Preamble Section

The device handler source file begins with the preamble section, which includes an .MCALL directive for the .DRDEF macro and any other macros you need that this chapter does not explicitly mention. The preamble also provides definitions for symbols that you will use later. Much of the work in the preamble is done by the .DRDEF macro.

7.2.1.1 .DRDEF Macro — Use the .DRDEF macro near the beginning of your device handler. This macro performs most of the work of the preamble section. Its 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

dd represents the two-character device name. The format of the .DRDEF macro call is as follows:

.DRDEF name,code,stat,size,csr,vec

name is a two-character device name, such as RK for the RK05 disk handler.

code is the octal numeric value that uniquely identifies the device. See Section 7.2.1.2.

stat is the device status bit pattern. Your value for *stat* can use the following symbols (described in Section 7.2.1.3):

FILST$	WONLY$	HNDLR$
RONLY$	SPECL$	SPFUN$

size is the size of the device in 256-word blocks; use a value of 0 if the device is not file-structured (see Section 7.2.1.4).

csr is the default value for the device's control and status register.

vec is the default value for the device's interrupt vector.

.MCALL Directive

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

.DRAST	.DRBEG	.DRFIN
.DRBOT	.DREND	.DRSET
.DRVTB	.FORK	.QELDF

In addition, if you assemble your handler with the conditional TIM$IT set to 1, .DRDEF issues an .MCALL directive for these macros:

.TIMIO and .CTIMIO

System Generation 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, you 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$IT (for device time-out), MMG$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.

Queue Element Offsets

The .DRDEF macro invokes .QELDF to define queue element offsets symbolically. The following example shows the queue element offsets generated. (See Section 7.9.3 for the queue element in XM systems.)

```
Q.LINK=0          (Link to next queue element)
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 buffer address)
Q.WCNT=^O12       (Word count)
Q.COMP=^O14       (Completion routine code)
Q.ELGH=^O16       (Length of queue element)
```

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

```
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=10
```

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 RT–11 device handlers follow.

To define an internal symbol for line feed (ASCII 12):

```
LF              = 12            ;ASCII FOR LINE FEED
```

To define other device registers:

```
RKDS            = RK$CSR        ;DRIVE STATUS REGISTER
RKER            = RKDS+2        ;ERROR REGISTER
RKCS            = RKDS+4        ;CONTROL STATUS REGISTER
RKWC            = RKDS+6        ;WORD COUNT REGISTER
```

The .DRDEF macro defines the following symbols for you:

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

7.2.1.2 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 currently defined in octal as Table 7–1 shows.

Table 7–1: Device-Identifier Byte Values

Name	Code	Device
RK	0	RK05 Disk
DT	1	TC11 DECtape
EL	2	Error Logger
LP	3	Line Printer
TT,BA	4	Console Terminal or Batch Handler
DL	5	RL01/RL02 Disk
DY	6	RX02 Diskette
PC	7	PC11 Reader/Punch
	10	Reserved (V2 PP handler)
MT	11	TM11/TMAll/TU10/TS03 MAGtape
RF	12	RF11 Disk
CT	13	TA11 DECassette
CR	14	CR11/CM11 Card Reader
	15	Reserved
DS	16	RJS03/RJS04 Fixed-Head Disk
	17	Reserved
MM	20	TJU16/TU45 MAGtape
DP	21	RP11/RP02/RP03 Disk
DX	22	RX11/RX01 Diskette
DM	23	RK06/RK07 Disk
	24	Reserved
NL	25	Null Device
	26–30	Reserved (DECnet)
	31–33	Reserved (CTS–300)
DD	34	TU58 DECtape II
MS	35	TS11/TS04 MAGtape
PD	36	PDT–11/130
PD	37	PDT–11/150
	40	Reserved

(Continued on next page)

Table 7–1: Device-Identifier Byte Values (Cont.)

Name	Code	Device
LS	41	Serial Line Printer
MQ	42	Internal Message Handler
DR	43	DRV11J Interface (MRRT)
XT	44	Reserved (MRRT)
	45	Reserved
LD	46	Logical Disk Handler
VM	47	KT11 Pseudo-Disk Handler
DU	50	MSCP Disk Class Handler (RA80, RC25)
SL	51	Single-Line Editor

To create device-identifier codes for devices that are not already supported by RT–11, start by using code 377 octal 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.

7.2.1.3 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 value of the status word is stored in block 0 of the handler file and in the $STAT table when the device is installed; the .DSTATUS programmed request returns this value to a running program. The .DRDEF macro sets up the device status word based on the arguments *code* and *stat*.

Table 7–2 shows the meaning of the bits in the device status word. The .DRDEF macro uses the symbol *dd*STS to represent the device status word.

Note that bit 11 in the status word should be set for device handlers that remove the queue element on entry and queue internally, and for devices such as magtape that have internal data that could need modification on abort. See Section 7.4 for more information on device handlers that do their own queuing. See Section 7.8.5 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 mnemonics 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   ;NO DIRECTORY
HNDLR$    ==   4000   ;ENTER HANDLER ON ABORT
SPFUN$    ==   2000   ;ACCEPTS SPECIAL FUNCTIONS
ABTIO$    ==   1000   ;ALWAYS TAKE ABORT ENTRY
VARSZ$    ==    400   ;HANDLER SUPPORTS VARIABLE-SIZE  VOLUMES
```

Table 7–2: Device Status Word

Bit	Symbol	Meaning
0–7	–	Device-identifier byte (see Section 7.2.1.2)
8	VARSZ$	0 = .SPFUN 373 requests are invalid for this handler 1 = .SPFUN 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 This bit is ignored in SJ systems.
12	SPECL$	1 = Special directory-structured device (examples are MT, CT)
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 MT, CT, PC, LP) 1 = This is a random-access device (examples are RK, DX)

For example, form the *stat* argument for the RK, MT, and LP handlers as follows:

For RK: FILST$
For MT: SPECL$!SPFUN$
For LP: WONLY$

7.2.1.4 Device Size Word — The *size* argument for the .DRDEF macro defines the size of the device in 256-word blocks. The .DRDEF macro puts this value into *dd*DSIZ. If the device is not random access, place the value 0 in *size*. The size of the RK device is 4800 decimal blocks (11300 octal); the size for the PC (paper tape) device is 0, since it is not random access.

The .DSTATUS 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.

7.2.2 Header Section

The second part of an RT–11 device handler is the header section. In the header section you invoke the .DRBEG macro to set up the first five words of the handler. This macro also stores five words of information in block 0 of the handler file, in locations 52 through 60, 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.

7.2.2.1 Information in Block 0 – Table 7–3 shows the five words in block 0 that the .DRBEG macro sets up by using the .ASECT directive. It also shows the three words .DRBOT sets up for bootable devices (see Section 7.10.2.6). In the table, the associated mnemonics are shown in square brackets, and the two-character device name is represented by *dd*. The installation verification code, which is optional, is described in Section 7.11.3.5.

Table 7–3: Information in Block 0

Location	Contents [and Mnemonic]
52	Size of the handler in bytes [ddEND-ddSTRT]
54	Size of the device in 256-word blocks [ddDSIZ]
56	Device status word [ddSTS]
60	A status word to reflect current system generation features [ERL$G + <MMG$T2> + <TIM$IT4>]
62	A pointer to the start of the primary driver (from .DRBOT)
64	The length of the primary driver, in bytes (from .DRBOT)
66	The offset from the start of the primary driver to the start of the bootstrap read routine (from .DRBOT)
176	CSR address [dd$CSR]
200	Start of installation verification code

7.2.2.2 First Five Words of the Handler – Table 7–4 shows the five words that the .DRBEG macro generates at the start of the handler's p-sect. In the table, *dd* represents the two-character device name.

7.2.2.3 .DRBEG Macro – Use the .DRBEG macro to set up the information in block 0 and the first five words of the handler. This macro also generates the appropriate global symbols for your handler. Before you use .DRBEG, you

Table 7–4: Handler Header Words

Word	Symbol	Contents
1	ddSTRT::	Device vector (for single-vector devices); Offset to table of vectors (for multi-vector devices)
2	–	Offset to interrupt service entry point
3	–	Priority (340)
4	ddLQE::	Pointer to the last queue element
5	ddCQE::	Pointer to the current queue element

must have invoked .DRDEF to define *dd*$CSR, *dd*$VEC, *dd*DSIZ, and *dd*STS. The format for .DRBEG is as follows:

.DRBEG name

name is the two-character device name.

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

7.2.2.4 Multi-Vector Handlers: .DRVTB Macro — An RT–11 device handler can service a device that has more than one vector. The PC handler, for example, services interrupts through vector 70 for the paper tape reader, and through 74 for the paper tape punch.

If your device has more than one interrupt vector associated with it, the handler must contain a table of three-word entries for each vector. The entry for each vector consists of the vector location, the interrupt entry point, and the Processor Status, or PS, value.

To set up the handler header for a multi-vector device, simply invoke the .DRVTB macro two or more times. The .DRVTB macro sets up the table of three-word entries for each vector of a multi-vector device. Place it in your handler 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.

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.

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 uses two vectors is the PC handler. The following example shows the source lines and the code the macros generate.

```
; PUNCH-READER VECTOR TABLE
_.IF EQ PR11$X                                  ;IF BOTH READER AND PUNCH
        .DRVTB   PC,PC$VEC,PCINT                ;TABLE FOR READER
        .DRVTB   ,PP$VEC,PPINT                  ;TABLE FOR PUNCH
_.ENDC
```

The vector table generated by the .DRVTB macros is as follows:

```
        .WORD    PC$VEC_&_^C3,PCINT-.,340!0     ;TABLE FOR READER
        .WORD    PP$VEC_&_^C3,PPINT-.,340!0     ;TABLE FOR PUNCH
        .WORD    0
        ;TO END THE TABLE
```

As you see in the example above, the priority bits of the PS are always set to 7, even if you omit the *ps* argument.

7.2.2.5 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 PC handler has separate interrupt entry points for its two vectors, so it can easily determine the source of the interrupt. Interrupts through vector 70 go to the routine at PCINT:; interrupts through 74 go to PPINT:.

Suppose that the PC handler had only one interrupt entry point, called PCINT:. In this case, the handler could distinguish which vector took the interrupt by setting the condition codes in the PS for the vectors. For the reader vector at 70, it could leave the C bit clear. For the punch vector at 74, it could set the C bit. Then, at PCINT:, control could pass to different routines based on the value of the C bit in the new PS. The following example shows how to invoke the .DRVTB macro and place values in the condition codes of the PS.

```
; PUNCH-READER VECTOR TABLE
.IF EQ PR11$X                                   ;IF BOTH READER AND PUNCH
        .DRVTB   PC,PR$VEC,PCINT                ;C BIT CLEAR
        .DRVTB   ,PP$VEC,PCINT,1                ;C BIT SET
.ENDC
```

7.2.3 I/O Initiation Section

The I/O initiation section contains the first executable instructions of the handler. 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 JSR PC instruction. The monitor calls the handler at the handler's sixth word — that is, the first word immediately after the five-word header. It 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 of the 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.

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 represent guidelines for a generalized I/O initiation section.

1. You should already have 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 find it necessary 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 is the step that 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
```

7.2.4 Interrupt Service Section

Control passes to the interrupt service section of the handler when a device interrupts or when the program requesting the I/O transfer aborts. 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 Table 6–3).

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. The various situations are discussed in the following sections.

7.2.4.1 Abort Entry Point — There are a number of situations that cause an abort in the queued I/O system: (1) a double CTRL/C can abort a running program; (2) the .HRESET programmed request causes an abort; (3) a trap to 4 or 10, or any other condition that produces the *?MON-F-* type of fatal error message, also causes an abort. On abort, whether or not the handler is entered at all depends on two factors. The handler is always entered at the abort entry point (the word immediately before the normal interrupt entry point) if an active queue element exists and it belongs to the aborting job. In FB and XM, the handler is also entered regardless of the existence of a queue element if HNDLR$ (bit 11) is set in the device status word. If HNDLR$ is set, the abort routine must consider two cases: there is pending I/O to abort; there is no I/O to abort. The SJ monitor ignores this bit. Additionally, handlers are never entered when a job aborts in the SJ environment; the SJ monitor simply performs a RESET instruction. In all environments, on entry to the handler, R4 always contains the job number of the aborting job. R0-R3 must be saved and restored.

When an abort occurs, it is important to stop I/O on some devices. Character-oriented devices, such as the paper tape reader/punch, fall into this category. On abort, the handler must stop the device in order to prevent a tape runaway condition, for example. It must also make sure that the device cannot interrupt again. So, character-oriented devices generally contain an abort routine; the abort entry point is simply a branch instruction to that routine. The PC handler, for example, has an abort routine that disables interrupts on the paper tape reader/punch. Then the handler exits to the monitor in the I/O completion section. The following lines are from the PC handler:

```
PCDONE: CLR     @#PC$CSR        ;TURN OFF THE READER INTERRUPT
        CLR     @#PP$CSR        ;TURN OFF THE PUNCH INTERRUPT
```

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, an RTS PC instruction is located at the abort entry point, which simply returns control to the monitor.

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.

7.2.4.2 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.

7.2.4.3 .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 macro also sets up a global symbol $INPTR, which contains a pointer to the $INTEN routine in the Resident Monitor. This pointer is filled in by the 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, as well.

abo is an optional argument that represents the label of an abort entry point. If you omit this argument, the macro generates an RTS PC instruction at the abort entry point, which is the word immediately preceding the interrupt entry point.

The following example from the PC handler shows the .DRAST macro call and the code it generates.

```
         .DRAST   PP,4,PCDONE

         .GLOBL   $INPTR             ;MAKE THIS SYMBOL GLOBAL
         BR       PCDONE             ;THE ABORT ENTRY POINT
PPINT::  JSR      R5,@$INPTR         ;JUMP TO MONITOR INTEN CODE
         .WORD    ^C<4*^O40>&^O340   ;NEW PRIORITY
```

The next example, from the RK handler, does not have an abort routine.

```
         .DRAST   RK,5

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

7.2.4.4 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.

   If the error is soft, the handler should prepare to retry the transfer. It should decrement the count of available retries. Then, 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.

   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 you learned in Chapter 6, the .FORK macro causes a return to the Resident Monitor, which dismisses the current interrupt. (Review Section 6.5.6 for details on the .FORK macro.) 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 an RTS PC 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.

7.2.5 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. Although the other sections of the handler are distinct, separate parts, the I/O completion section is actually an extension of the interrupt service section and the dividing line between these two sections is artificial. Control does not pass to the I/O completion section as a result of a monitor call, a subroutine call, or a jump, but rather as a result of normal flow of execution through the interrupt service section. Execution passes to the I/O completion section when a hard error is detected, when a soft error condition exhausts the number of retries allowed for it, or when a data transfer completes. (Note that you can branch directly to this section from the I/O initiation section if you detect a hard error immediately.)

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, 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 RK handler. They illustrate how the handler sets the hard error bit and jumps back to the monitor.

```
BIS       #HDERR$,@-(R5)  ;SET HARD ERROR BIT
.                         ;(R5 POINTS TO THIRD WORD OF
.                         ;QUEUE ELEMENT; POINTER TO
.                         ;CSW IS SECOND WORD.)
.DRFIN    RK              ;JUMP TO MONITOR
```

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. As the example in point 2 shows, the .DRFIN macro generates the code to perform the jump.

For a character- or word-oriented device, such as paper tape, 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. Examples of conditions that cause end-of-file are absence of tape in the paper tape reader, and detection of CTRL/Z typed on the console terminal. 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.

The PC handler uses the reader/punch ready bit in the device status register as the internal EOF flag. The following example shows how the PC handler zero-fills the user buffer when it detects end-of-file, sets an internal EOF flag, and jumps back to the monitor.

```
1$:     CLRB    @-(R4)          ;CLEAR A BYTE
        INC     (R4)+           ;BUMP BUFFER ADDRESS
        DEC     @R4             ;COUNT DOWN BYTES REMAINING
        BNE     1$              ;LOOP UNTIL DONE
PCDONE: CLR     @#PC$CSR        ;TURN OFF THE READER INTERRUPT
        CLR     @#PP$CSR        ;TURN OFF THE PUNCH INTERRUPT
        CLR     PCFBLK+2        ;CLEAR FORK BLOCK TO AVOID DISPATCH
PCFIN:  .DRFIN  PC              ;GO TO I/O COMPLETION
```

When the handler is called again with a new queue element for another READ operation, it first checks its internal EOF flag. Finding it set, the handler sets the EOF bit of the Channel Status Word, bit 13, and jumps back to the monitor. The Resident Monitor eventually clears this bit, when the next I/O request is made for this channel.

The following example shows how the PC handler tests the device ready bit — which it uses as its internal EOF flag — sets the EOF bit for the user program, and jumps back to the monitor.

```
        MOV     #PC$CSR,R5          ;READ REQUEST, GET THE CSR
        TST     (R5)+               ;IS READER READY?
        BPL     PCGORD              ;YES, START TRANSFER
        BIS     #EOF$,@-(R4)        ;NOT READY ON ENTRY, SET EOF
        BR      PCFIN               ; AND COMPLETE OPERATION
```

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.

7.2.6 Handler Termination Section

The purpose of the handler termination section is to declare some global symbols and to establish a table of pointers to offsets 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.

7.2.6.1 The .DREND Macro — The format of the .DREND macro is as follows:

.DREND name

name is the two-character device name.

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

7.2.6.2 Pseudo-Devices — You can write a device handler for a pseudo-device (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 pseudo-devices are NL (the null device) and MQ (the message queue 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 pseudo-devices do not interrupt, the handler needs no interrupt service section, and no .DRAST macro call.

7.3 Skeleton Outline of a Device Handler

The skeleton outline in Figure 7–1 provides the structure for a simple device handler. In the figure, *SK* is the device name.

Figure 7–1 Skeleton Device Handler

```
.TITLE  SK      V05.00

;  SK DEVICE HANDLER

.IDENT  /V05.00/

.SBTTL  PREAMBLE SECTION

.MCALL  .DRDEF
.DRDEF  SK,377,WONLY$,0,177514,200

SKBR    = SK$CSR+2              ;SK BUFFER REGISTER
SKIE    = 100                   ;INTERRUPT ENABLE BIT

.SBTTL  HEADER SECTION

        .DRBEG  SK

.SBTTL  I/O INITIATION SECTION

        MOV     SKCQE,R4        ;R4 POINTS TO CQE
        ASL     Q$WCNT(R4)      ;MAKE WORD COUNT BYTE COUNT
        BEQ     SKDONE          ;A SEEK COMPLETES IMMEDIATELY
        BCC     SKERR           ;THIS IS A WRITE-ONLY DEVICE -
                                ;A READ REQUEST IS ILLEGAL
RET:    BIS     #SKIE,@#SK$CSR#;ENABLE INTERRUPTS
        RTS     PC              ;WAIT FOR ONE

.SBTTL  INTERRUPT SERVICE SECTION

        .DRAST  SK,4,SKDONE
        MOV     SKCQE,R4        ;R4 POINTS TO CQE
        BIT     #100200,@#SK$CSR ;ERROR OR READY?
        BMI     RET             ;ERROR - HANG UNTIL CORRECT
        BEQ     RET             ;NOT READY - EXIT AND WAIT
        BIC     #SKIE,@#SK$CSR ;DISABLE INTERRUPTS
        .FORK   SKFBLK          ;PROCESS REMAINING CODE AT
                                ;FORK LEVEL
        ADD     #Q$WCNT,R4      ;OFFSET QUEUE ELEMENT POINTER

SKNEXT: TSTB    @#SK$CSR        ;READY FOR NEXT CHAR?
        BPL     RET             ;NO - BRANCH BACK
        TST     @R4             ;ANY LEFT TO PRINT?
        BEQ     SKDONE          ;NO - TRANSFER IS DONE
        MOVB    @-(R4),R5       ;GET A CHARACTER
        INC     (R4)+           ;BUMP BUFFER POINTER
        INC     @R4             ;BUMP CHARACTER COUNT
        BIC     #^C<177>,R5     ;7-BIT ASCII
        MOVB    R5,@#SKBR       ;SEND CHAR TO DEVICE
        BR      SKNEXT          ;TRY FOR ANOTHER

.SBTTL  I/O COMPLETION SECTION

SKERR:  BIS     #HDERR$,@-(R4)  ;SET ERROR BIT IN CSW
SKDONE: BIC     #SKIE,@#SK$CSR    ;DISABLE INTERRUPTS
        .DRFIN  SK              ;JUMP TO MONITOR

SKFBLK:  .WORD  0,0,0,0         ;FORK QUEUE ELEMENT

.SBTTL  HANDLER TERMINATION SECTION

        .DREND  SK
.END
```

7.4 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.

NOTE

> Although internal queuing may be possible in some cases, much depends on the individual situation. For some handlers and devices, internal queuing is impractical or impossible. DIGITAL does not recommend the use of internally queued handlers with RT–11.

As an example, consider a process controller with input counters and an A/D converter. Since the converter is a slow device, the IP–11 controller can read an input counter while running the converter. Another example is the RT–11 message queue, implemented through the MQ handler, for system job communication. If one job sends a message to a second job, and the second job does not accept the message, the MQ handler waits. However, if a receive request for the job is next in the queue, the MQ handler processes it. To do this, it takes the original send request from the monitor/handler queue, queues it internally, and then services the receive request.

In general, the handler follows a procedure to implement internal queuing. When an I/O request is made for the handler, it is always the first and only request in the monitor/handler queue. As soon as it comes in, the handler queues it internally and clears *dd*CQE and *dd*LQE to "remove" the request from the monitor/handler queue. Note that the queue element is still busy — it is still in use by the handler.

7.4.1 Implementing Internal Queuing

When the handler is first entered for a request, at the sixth word, it must check the queue element for validity. An invalid request causes an immediate fatal error.

If the request is for a procedure that completes very quickly, such as a seek on paper tape, the handler performs the operation. Then it issues the .DRFIN macro call to release the queue element and inform the requesting program that the operation completed. In summary, the handler performs the operation if it is one that can be taken care of both synchronously and quickly.

If the request is for a procedure that requires calculation and some time to complete, the handler places the request on its internal queue by using the queue element's link word. The link word is 0, because this element is the first and only element on the monitor's queue for the handler.

In summary, the handler queues the request internally if it is one that requires some work and time, and must be taken care of asynchronously. If the request is the first one on the internal queue, the handler starts the operation, waits for it to complete, and exits with an RTS PC instruction. If the request is not the first one on the internal queue, the handler does not start the operation and simply exits with an RTS PC instruction.

7.4.2 Interrupt Service for Handlers That Queue Internally

When an operation completes, the handler is 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. If the operation is not complete, the handler restarts it and returns to the monitor. If the transfer is complete, the handler must put the internally queued request back on the monitor/handler I/O queue by setting *dd*CQE and *dd*LQE. In this situation, the handler needs to return the request to the main I/O queue, but it also needs to continue execution (rather than return immediately to the monitor) to check its internal queue in case there is another outstanding request.

To return the request to the monitor without exiting, the handler must perform a .DRFIN substitute. The following example illustrates how a handler does this.

R4 points to the queue element on the internal queue, at its third word.

```
MOV     ddCQE,-(SP)     ;IN CASE THE MONITOR/HANDLER QUEUE
                        ;HAS AN ELEMENT WHEN WE TAKE THIS
                        ;INTERRUPT
MOV     R4,ddCQE        ;PUTS INTERNAL QUEUE ELEMENT ON THE
MOV     R4,ddLQE        ;MONITOR/HANDLER QUEUE
CLR     Q$LINK(R4)
MOV     PC,R4           ;JSR
ADD     #ddCQE-.,R4     ;  VERSION
MOV     @#54,R5         ;    OF
JSR     PC,@270(R5)     ;      .DRFIN
MOV     @SP,ddCQE       ;RESTORE POSSIBLE OTHER
MOV     (SP)+,ddLQE     ;QUEUE ELEMENT
.
.
.
.
```

(Check the internal queue now and start another operation if necessary.)

```
.
.
.
RTS     PC              ;RETURN
```

7.4.3 Abort Procedures for Handlers That Queue Internally

Whether or not your handler queues I/O requests internally, RT–11 maintains a count of outstanding I/O requests for each channel. There is one counter in each channel; the total of outstanding I/O requests is in the Resident Monitor. When a job aborts, any outstanding I/O requests it has must be removed from the counters. This occurs automatically if the handler relies strictly on the monitor/handler I/O queue.

If, however, the handler implements an internal I/O queue, it must follow a procedure to reduce the count of outstanding I/O requests. This procedure involves making sure that the handler will be entered when any job aborts, whether or not the handler appears to have an active queue element, by having the handler set bit 11, HNDLR$, in the device status word *dd*STS when it invokes .DRDEF. In FB and XM systems, this forces the handler to be entered on all aborts, even if there is nothing on its monitor/handler queue. (The SJ monitor ignores this bit, since there is no problem in a single-job system.)

If the handler is entered at the abort entry point, then, it must check its internal queue for elements belonging to the aborted job. (Remember that R4 always contains the job number of the aborting job.) The handler should purge its internal queue of these elements and use one of the following procedures to reduce the monitor's count of outstanding I/O requests. R0 through R3 must be saved and restored.

If *dd*CQE has a non-zero value:

1. Remove any internal elements for the aborting job.
2. Link the elements together via the element's link word; the last element's link word must be 0. Set *dd*LQE to point to the last element in the aborting job.
3. If *dd*CQE points to an element belonging to the aborting job, halt I/O and issue a .DRFIN. If you cannot halt I/O, then issue an RTS PC instruction, wait for an interrupt, and then issue a .DRFIN.

   If *dd*CQE does not point to an element belonging to the aborting job, simply issue the RTS PC instruction.

If *dd*CQE has the value 0:

1. Remove any internal queue elements that belong to the aborting job. If there are none, simply issue the RTS PC instruction.
2. Link the elements together, as described in 2 above, setting *dd*CQE to point to the first element, and *dd*LQE to point to the last element. Note that the last element's link word must be 0.
3. Issue the .DRFIN macro.

7.5 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 XM), where *dd* is the two-character device name. For example, the following command changes the column width for the line 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 the line 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 Chapter 4 of the *RT–11 System User's Guide* 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.

7.5.1 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 *dd*.SYS on the system device (*dd*X.SYS in XM). This handler need not be installed in the running system. The monitor then reads blocks 0 and 1 of the handler into the USR buffer area in memory. 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 are changed. However, any memory-resident copy of the handler is not affected.

7.5.2 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 with a zero word. Use the .DRSET macro, described below, to generate the table.

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 the line 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 Radix–50 code for the option name, 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 a pointer 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.

Figure 7–2 shows a summary of the SET option table.

Figure 7–2: SET Option Table

VALUE TO PASS IN R3 TO THE SET ROUTINE	
RADIX-50 FOR OPTION NAME (TWO WORDS)	
CODE FOR VALID SET COMMAND TYPES	POINTER TO SET ROUTINE

7.5.3 .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 use 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 should not contain any embedded spaces or tabs.

val is a parameter that will be passed in R3 to the routine. 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 one in block 0 or 1 of the handler. It 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 must not go above address 776.

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.

7.5.4 Routines to Modify the Handler

Your handler needs one routine for each SET option that is valid. 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.

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 decimal 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. 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.

The routine can indicate that a command is illegal by returning with the carry bit set. For example, the line 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!
```

You terminate this section with an .ASECT directive, after which you set the location counter to 1000. Then you can continue with the rest of the handler code, starting with the .DRBEG macro, which establishes the handler header.

7.5.5 Examples of SET Options

The following examples taken from a line 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 line printer WIDTH option is being established, that 30 decimal is a default value of some kind, that O.WIDTH is the routine that modifies code for it, and that it takes a numeric argument:

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

The next call indicates that the line printer CR option is being established, that "NOP" is to be passed to the routine, that O.CR is the name of the routine that modifies code for this 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
```

```
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     RO,COLCNT              ;MOVE VALUE FROM USER TO
        MOV     RO,RSTC+2              ;TWO CONSTANTS
        CMP     RO,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
        BEQ     RSTC-CROPT+.           ;A NEW INSTRUCTION
        MOV     R3,CROPT               ;ENTRY POINT FOR
                                       ;"NOCR" (O.CR+4);
                                       ;MOVE EITHER "NOP" OR
                                       ;PREVIOUS LINE TO CROPT
        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 manages to substitute 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 decimal 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.

7.6 Device I/O Time-Out

Through the optional feature device time-out, a handler can 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.

You can select the device time-out feature at system generation time. Time-out is used by parts of the RT–11 multi-terminal monitor. The option is automatically included in your system if you select multi-terminal time-out support or DZ modem support. Otherwise, if you need to use the feature in your handler, you must specifically include it at system generation time. It is also required for DECnet applications.

RT–11 provides two macros to help you implement device time-out 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 an .MCALL directive for the .TIMIO and .CTIMIO macros.

7.6.1 .TIMIO Macro

Use the .TIMIO macro in the handler I/O initiation section to issue the time-out 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 XM systems, the completion routine executes with kernel mapping, since it is still a part of the interrupt service routine. (See Section 6.7 for more information about interrupt service routines and the XM monitor.) 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 cancelled successfully.

The format of the macro is as follows:

.TIMIO tbk,hi,lo

tbk is the address of the timer block, a seven-word pseudo timer 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 7–5.

Table 7–5: 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 177000 through 177377.
12	C.SYS	monitor	–1
14	C.COMP	user	Address of the completion routine to execute if time-out 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 decimal ticks per second if your system is running with 60-cycle power. If your system is running with 50-cycle power, there are 50 decimal ticks per second. 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 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 two-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. Start with 177000 and work up to the highest valid sequence number, 177377. 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
```

7.6.2 .CTIMIO Macro

When the condition the handler was waiting for occurs, you should issue a cancel time-out call, which disables the completion routine. Use the .CTIMIO macro call in your handler to cancel the time-out 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 line 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 two-minute timer with .TIMIO, and returns to the monitor with an RTS PC 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 two-minute timer, and returns again to the monitor with RTS PC to wait for an interrupt. (See Figure 7–3 for the line 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. Since seeks interrupt twice, the handler should not cancel the timer after the first interrupt. When the second interrupt occurs, though, 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.

7.6.3 Device Time-out Applications

Device time-out 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.

7.6.3.1 Multi-terminal Service — The resident multi-terminal service in RT–11 that supports DZ11 and DZV11 modems uses device time-out 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. The last instruction in 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.

7.6.3.2 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 corrupted the operation.

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 an RTS PC 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 time-out. 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 an RTS PC instruction to await another interrupt.

If the handler decides that the time-out 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.

7.6.3.3 Line Printer Handler Example — The extended example shown in Figure 7–3 consists of excerpts from a version of the RT–11 line 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 two-minute timer. The handler then returns to the monitor with an RTS PC 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 two-minute timer, and returns to the monitor with an RTS PC instruction to await an interrupt.

Figure 7–3: Line 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
```

Figure 7–3: Line Printer Handler Example (Cont.)

```
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
    .
    .
    .

; 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$-.,R0    ; POINT TO TIMIO COMPLETION ROUTINE
         MOV      R0,TICMPL;SAVE IT
         .TIMIO   TIMBLK,0,2*60.*60.   ;SET A 2-MINUTE TIMER
2$:      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    177000+3   ;SEQUENCE NUMBER
         .WORD    0        ;MONITOR PUTS -1 HERE

TICMPL:  .WORD    0        ;ADDRESS OF COMPLETION ROUTINE
SYNBLK:  .WORD    0        ;SYNCH BLOCK
.FRKBLK:.WORD     0,0,0,0
SYJNUM:  .WORD    0        ;JOB NUMBER
         .WORD    0,0,0,-1,0 ;OTHER
MESSAG:  .ASCIZ   "?LP-W-LP off line - please correct"
         .EVEN
         .DREND   LP
```

7.7 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 can run in either the FB or XM environment. If your system has the capability to run system jobs, the error logger runs as a system job; otherwise, the error logger can run as an ordinary foreground job. 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 the SJ monitor.

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 User's Guide* 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.

7.7.1 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, and once again when the allotted number of retries has been exhausted.

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.

7.7.1.1 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 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.

7.7.1.2 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, no tape in paper tape reader, and so forth. A soft error that has exhausted its allotted number of retries is considered a hard error.

R5 must point to the third word 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.

7.7.1.3 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 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.

7.7.1.4 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 of 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 which reflect device reliability.

7.7.1.5 To Call the Error Logger — Once the required registers are set up, call the error logger as follows:

```
JSR     PC,@$ELPTR
```

$ELPTR is 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
```

The SRUN or FRUN command fills in the error logger entry point; the UNLOAD EL command zeroes $ELHND.

On return from the error logger call, R0 through R3 are restored in your handler, and R4 and R5 are destroyed.

7.7.2 Error Logging Examples

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

7.7.3 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:

```
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's general classification.

C2 is the two-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 two-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. This is also the default output file for EL itself. 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 Chapter 8 of this manual.

7.8 Special Functions

Sometimes handlers need to perform device-specific actions for which there are no corresponding RT–11 programmed requests. Examples of these actions include rewinding magtapes and reading or writing absolute sectors on diskettes. The .SPFUN programmed request provides a means for programs to initiate such special functions. When a program issues a .SPFUN request, it supplies a special function code as one of the arguments. This code tells the handler which special function it is to perform. For example, the code that tells the MT handler to perform an off-line rewind is 372.

7.8.1 .SPFUN Programmed Request

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

.SPFUN area,chan,func,buf,wcnt,blk[,crtn]

For a complete description of the arguments for .SPFUN programmed request, see the *RT–11 Programmer's Reference Manual.*

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* arguments depend on the particular special function the request invokes. Of course, if the request calls for a data transfer, the arguments have their usual meanings. Note, however, that 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.

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.

7.8.2 How to Support Special Functions in a Device Handler

To implement support for special function calls in your handler, you must specify SPFUN$ as one of the bits in the *stat* value you provide to the .DRDEF macro. This indicates that the handler can accept special functions.

Next, define symbolics 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 octal). Consult the *RT–11 Programmer's Reference Manual* for a complete list of RT–11 codes. 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 the *RT–11 Programmer's Reference Manual* 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. This policy should avoid conflicts with new RT–11 codes in the future.

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. Be sure to check that the code is valid for your device and if it is not, return a hard error immediately.

If this is a request for a special function, the handler should initiate that function and return with an RTS PC 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.

7.8.3 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 .DSTATUS programmed request.

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

7.8.4 Bad Block Replacement

If your handler is to support bad block replacement, you must implement special function codes 377, 376, and 374, as they are implemented for the DL handler. See the description of the RL01 device in Chapter 10 for more information.

DUP requires modification to correctly initialize and squeeze a device that supports bad block replacement. See the *RT–11 System Release Notes*.

7.8.5 Devices with Special Directories

The RT–11 monitor can interface to file-structured devices having nonstandard (that is, non-RT–11) directories. Examples of special devices are magtape and cassette. 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 .LOOKUP, .ENTER, .CLOSE, and .DELETE, as well as data transfers.

The monitor requests a special directory operation by placing a positive, nonzero value in the function code byte of the queue element. The positive function codes are standard for all devices. They are as follows:

Code	Function
1	Close
2	Delete
3	Lookup
4	Enter

These functions correspond to the programmed requests .CLOSE, .DELETE, .LOOKUP, and .ENTER, which are described in the *RT–11 Programmer's Reference Manual*. The .RENAME request is not supported for special devices.

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 device handler during directory operations are returned to the monitor. A unique error code is chosen for each type of error. This error code is directly returned by placing it in SPUSR (special device USR error), located at fixed offset 272 from the start of the Resident Monitor. 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 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, by-passing 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.

7.9 Device Handlers in XM Systems

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

7.9.1 Naming Conventions and the System Conditional

When you write a device handler, write a common source file called *dd*.MAC, where *dd* is the two-character device name enclosing the code that pertains to extended memory support in conditional assembly directives. The system generation conditional that represents extended memory support is MMG$T, which has a value of 0 if extended memory support is not selected and of 1 if extended memory support is selected. This means that the extended memory code is only assembled when the value of the conditional MMG$T is 1. Assemble your source file with the system conditional file, SYCND, and with XM.MAC, producing *dd*X.OBJ for XM systems, or *dd*.OBJ for SJ and FB systems. This procedure ensures that the system generation features that the handler supports match those of the current monitor.

7.9.2 XM Environment

In an XM system, handlers must reside within the low 28K words of physical memory. Further restrictions may also apply, depending on the presence or absence of .FETCH support in your XM monitor.

If your XM monitor does not have .FETCH support, handlers cannot reside within the area of physical memory mapped by PAR1, an area that includes the memory locations between 20000 and 37777. In addition, if your system uses the MQ handler and if you defined the SYSGEN parameter MQH$P2 = 1, handlers cannot reside in the PAR2 address space (40000–57777). Before you run a program that uses handlers, you must make the handlers resident with the LOAD monitor command, which enforces the address restriction.

If your XM monitor includes .FETCH support (a SYSGEN option), the only restriction on the location of handlers in memory is that they must reside in the low 28K words. With .FETCH support enabled, you need not load most XM handlers before running programs. Even with .FETCH support enabled, however, some handlers may still have to be loaded with the LOAD command. All Digital-supplied XM handlers are fetchable with the exception of the file-structured magtape handlers (MS, MT, and MM).

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 6). 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 with any kind of mapping

As you may suspect, the chief difficulty for handlers in XM systems is communicating with the user data buffer. This difficulty arises from the fact that 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 an XM 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 bits for XM (bits 16 and 17 of the 18-bit address, or bits 16–21 of the 22-bit address), which must be 0.

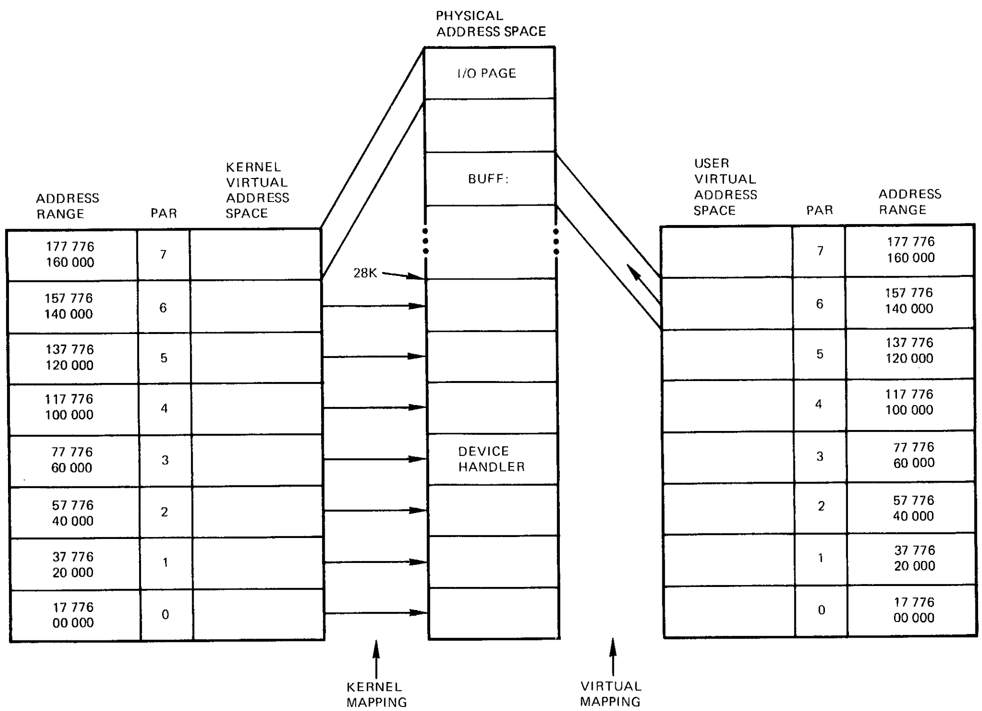
Figure 7–4 illustrates an XM system. The program that requests an I/O transfer has mapped its data buffer area into physical memory above the 28K word boundary.

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.

7.9.3 The Queue Element in XM

In order to locate the actual user buffer in physical memory, the handler requires an extra word of information in the queue element. This word is a value for PAR1 that, when combined with the user virtual buffer address,

Figure 7–4 Device Handler in XM



provides the physical address of the buffer. Although only one extra word is used for XM handlers, the queue element allows room for two words in addition to that. These two words, at offsets 20 and 22, are reserved for future use by DIGITAL and should not be used.

When the system conditional MMG$T is set to 1, the .QELDF macro invoked by .DRDEF in the beginning of your handler expands to generate the correct offsets for the XM queue element. The macro expansion is as follows:

```
Q.LINK=0          (Link to next queue element)
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)
Q$LINK=-4         (Symbols for easy reference:)
Q$CSW=-2
Q$BLKN=0
Q$FUNC=2
Q$JNUM=3
Q$UNIT=3
Q$BUFF=4
Q$WCNT=6
Q$COMP=^O10
Q.PAR=^O16        (PAR1 value)
Q$PAR=^O12
Q.ELGH=^O24       (Length of queue element)
```

7.9.4 DMA Devices: $MPPHY Routine

DMA devices – most disks, for example – usually work with 18- or 22-bit addresses so that their handlers need not map to the user buffer. These handlers use the monitor routine $MPPHY to find the user buffer in physical memory. $MPPHY uses the Q.PAR value from the queue element and the Q.BUFF virtual buffer address 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:

```
JSR     PC,@$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 BUFFER ADDRESS IN QUEUE ELT
        JSR     PC,@$MPPTR      ;CONVERT USER VIRTUAL ADDRESS TO PHYSICAL
        MOV     (SP)+,-(R4)     ;PUT LOW 16 BITS IN RKBA, HIGH BITS ON STACK
        MOV     (R5)+,-(R4)     ;PUT WORD COUNT INTO RKWC
        BEQ     7$              ;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
```

7.9.5 Character Devices: $GETBYT and $PUTBYT Routines

The handlers for character-oriented devices, such as paper tape and line printers, must transfer the data from the device to the user buffer area themselves. 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.

7.9.5.1 $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:

```
JSR     PC,@$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 buffer address (Q.BUFF) in the queue element is updated by 1. If a mapping overflow occurs, the monitor routine subtracts 20000 from the value in Q.BUFF and adds 200 to the value in Q.PAR. Mapping overflow is described in more detail in Section 7.9.7.

The following example from the PC handler shows how the handler gets a byte from the user buffer and punches it on paper tape.

```
        MOV     PCCQE,R4        ;POINT TO CURRENT QUEUE ELEMENT
        MOV     #PP$CSR,R5      ;POINT TO PUNCH STATUS REGISTER
        TST     (RT)+           ;ERROR?
        BMI     PPERR           ;YES, PUNCH OUT OF PAPER
        TST     Q$WCNT(R4)      ;ANY MORE CHARACTERS TO OUTPUT?
        BEQ     PCDONE          ;NO, TRANSFER DONE
        INC     Q$WCNT(R4)      ;DECREMENT BYTE COUNT (IT IS NEGATIVE)
        JSR     PC,@$GTBYT      ;GET A BYTE FROM USER BUFFER
        MOVB    (SP)+,@R5       ;PUNCH IT
```

7.9.5.2 $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:

```
JSR     PC,@$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 20000 from the value in Q.BUFF and adds 200 to the value in Q.PAR. Mapping overflow is described in more detail in Section 7.9.7.

The following example from the PC handler shows how the handler gets a character from the paper tape reader and moves it to the user buffer.

```
MOV     PRCQE,R4            ;R4 POINTS TO Q.BLKN
MOVB    @#PRB,-(SP)         ;GET A CHARACTER
JSR     PC,@$PTBYT          ;MOVE IT TO USER BUFFER
DEC     Q$WCNT(R4)          ;DECREASE BYTE COUNT
```

7.9.6 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:

```
JSR     PC,@$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 2. If a mapping overflow occurs, the monitor routine subtracts 20000 from the value in Q.BUFF and adds 200 to the value in Q.PAR. Mapping overflow is described in more detail in Section 7.9.7.

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
        JSR     PC,@$PTWRD          ;CALL THE ROUTINE
```

7.9.7 Handlers That Access the User Buffer Directly

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 Cyclical 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. However, the disk is not DMA, so the DX handler cannot use the $MPPHY monitor routine. Moreover, other monitor routines for character-oriented devices available to a silo device are too slow for practical use. 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. In RT–11 Version 4, handlers doing their own buffer mapping accessed kernel PAR1 directly, but no handler could load into the PAR1 area because the handler would be in danger of unmapping itself while executing. In Version 5, handlers map to the user buffer through the monitor routine $P1EXT.[1]

[1] Because all relevant code is executed outside the PAR1 area, one of the restrictions that prevented Version 4 handlers from loading into the PAR1 area no longer applies. The other major restriction, the problem of interrupt service in the PAR1 area, is handled in Version 5 XM monitors that include .FETCH support by a vector forwarding technique that is transparent to the handler.

$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 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. P1EXT is a monitor fixed offset containing a pointer to the routine $P1EXT.

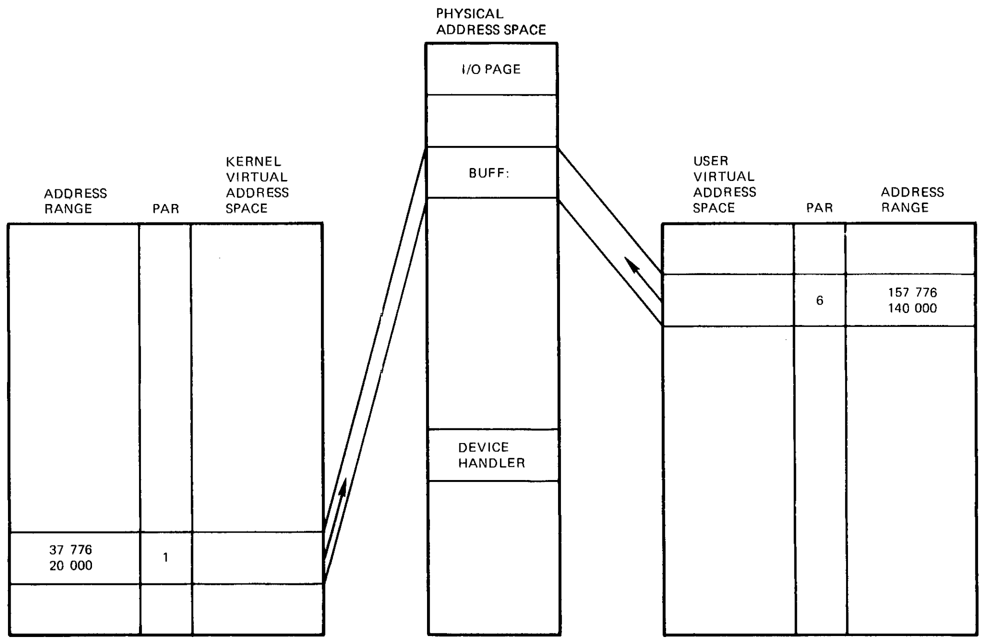
```
        MOV     @#SYSPTR,R4         ;R4 -> MONITOR BASE
        MOV     P1EXT(R4),(PC)+     ;GET ADDRESS OF EXTERNALIZATION ROUTINE
$P1EXT: .WORD   P1EXT               ;POINTER TO EXTERNALIZATION ROUTINE
        .
        .
        .
;--- Remove two lines below if not memory management
        JSR     R0,@$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 7–5 shows a virtual job in a typical XM 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.

Figure 7–5: Device Handler Mapping to User Buffer Area

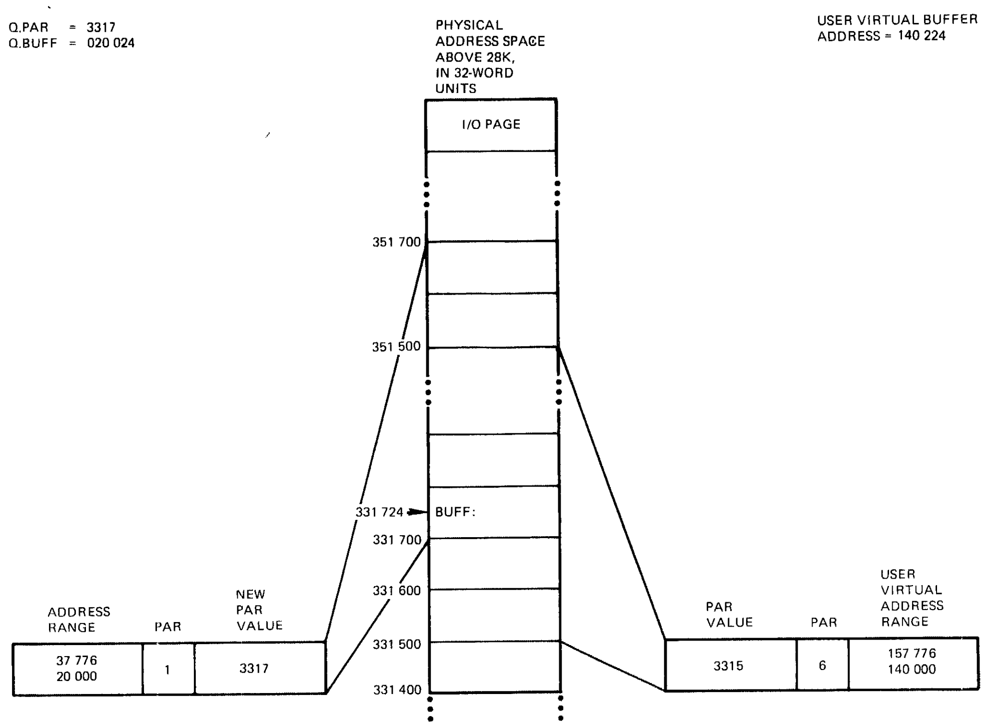


PAR1 maps to physical memory in units of 32-word decimal blocks and at most can map an area 4K words long. (Note that the page length of PDR1 is always set to map the entire page.) If the user buffer starts at a location in physical memory that is not an even multiple of 32 words, PAR1 maps to the first 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 7–6 shows how this mapping works.

Figure 7–6 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 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. The DX handler example just described transfers 64 words at a time, adding 2 to PAR1 after each transfer to avoid mapping overflow.

Figure 7–6: PAR1 Mapping



Another way to advance the buffer pointer is to modify the value of Q.BUFF by modifying the value in the queue element itself. In order 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.

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 7–6. 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.)

7.10 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.

7.10.1 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. RT–11 distributed monitors are named RT11BL.SYS, RT11SJ.SYS, and RT11FB.SYS. If you create a monitor through the system generation process, its name is RT11*xx*.SYG, where *xx* represents BL, SJ, FB, or XM. You must rename the monitor to .SYS before you use it.

Blocks 0 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.

The monitor file itself does not contain any device-specific code, nor does it have links to any specific device handler before bootstrap time. Instead, 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.

7.10.2 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.

A system device handler can contain SET options. So, if SET options are part of your handler, you need not remove them to create a system device handler.

7.10.2.1 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 two-character device name. The code executes at location 0 in physical memory.

7.10.2.2 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::.

7.10.2.3 Software Bootstrap — The software bootstrap executes 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 must perform the following functions in the order shown:

1. Set up the stack at location 10000.
2. Save the number of the device unit from which the system was just bootstrapped. (This is a value in the range 0 through 7.) 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. Call the bootstrap read routine to read in the rest of the bootstrap.
4. Put a pointer in B$READ to the bootstrap read routine.
5. Put the Radix–50 value for "B$DNAM" in B$DEVN.
6. Store the device unit number in B$DEVU.
7. Jump 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.)

7.10.2.4 Bootstrap Read Routine — The purpose of the bootstrap read routine 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 and by the software bootstrap 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 must be a non-interrupt routine 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 an RTS PC instruction, with the carry bit clear.

The bootstrap read routine should be located in your primary driver at location *dd*BOOT + 210. (Location 210 is the lowest address at which the read routine can be located.)

7.10.2.5 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.

7.10.2.6 .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 through 206 are not available for your use.

The format for the .DRBOT macro is as follows:

.DRBOT name,entry,read

name is the two-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 octal 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 octal bytes long or less, including the error routine and the locations from 60 through 206. You may have noticed that 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 non-system 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.

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

7.10.3 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:
```

7.10.3.1 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 two-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. (Note that the .SYS file type is both the default and the required file type.) It reads the first five blocks, blocks 0 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 7–6 for the primary driver and the secondary bootstrap.

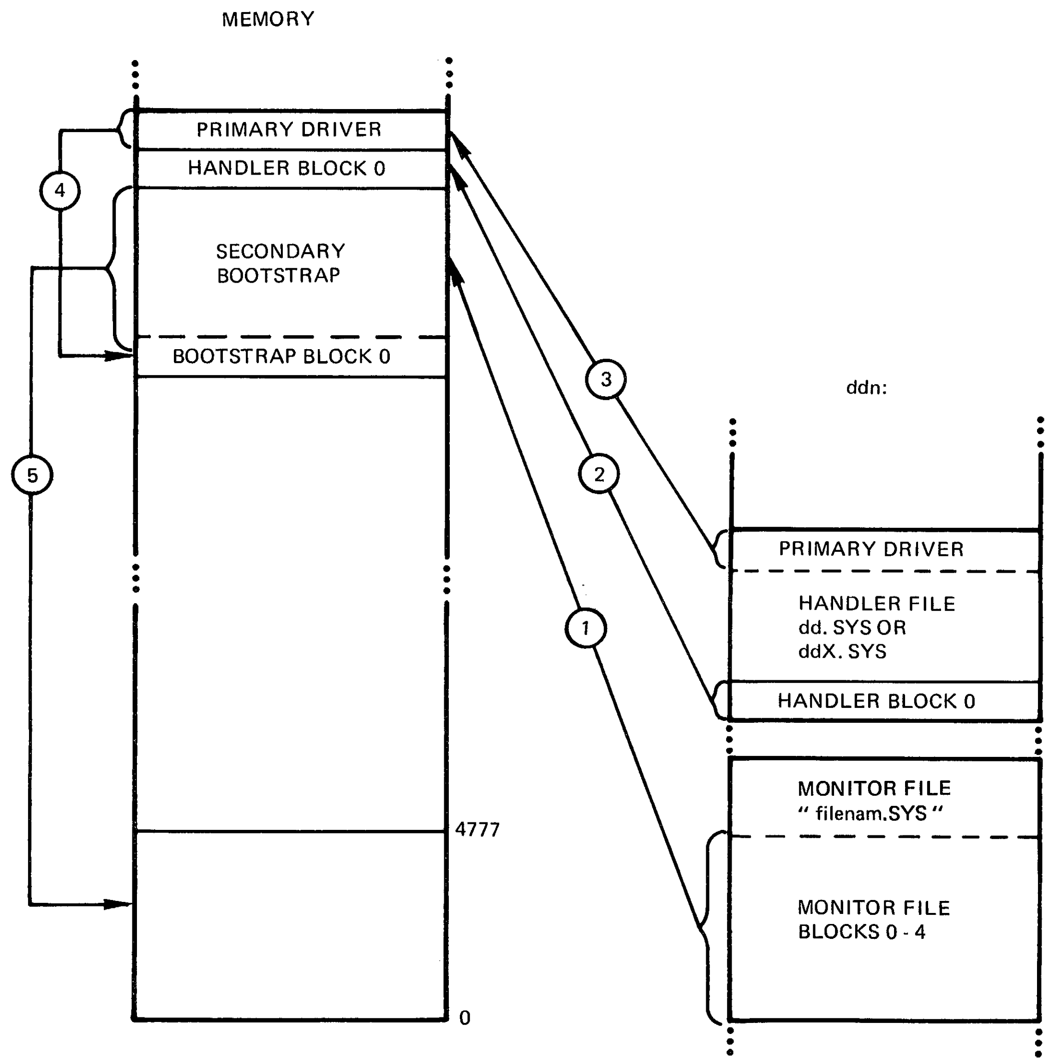
Table 7–6: DUP Information

Offset from Start of Memory Buffer	Contents
4722	Booted unit number
4724–4726	Booted file name in Radix–50
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 7–7 illustrates the entire procedure.

Figure 7–7: BOOT ddn:filnam Procedure



7.10.3.2 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 two-character name of the device that is to receive the bootstrap; *m* is its unit number.

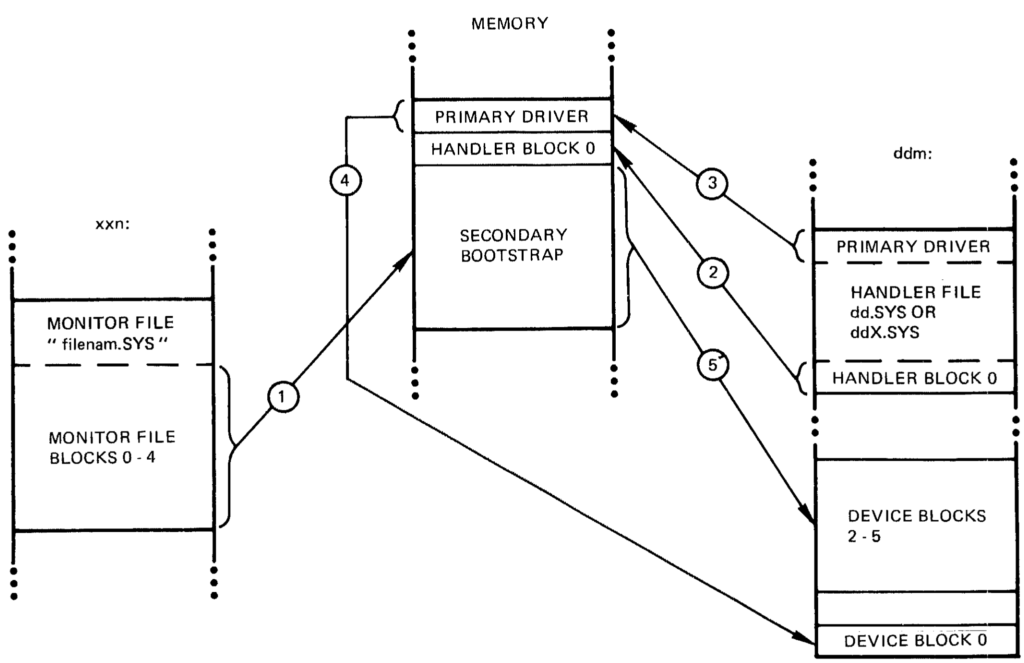
As soon as this command is issued, DUP checks that devices *xx* and *dd* are random-access devices. Next, it locates the monitor file *filnam*.SYS on the *xxn:* device. It reads the first five blocks of the monitor file, blocks 0 through 4, into a memory buffer. These blocks contain the secondary bootstrap 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 7–8 illustrates the entire procedure.

Figure 7–8: COPY/BOOT xxn:filnam ddm: Procedure



7.10.3.3 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 two-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 7–7 for the primary driver and the secondary bootstrap.

Table 7–7: 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.

Figure 7–9 illustrates the entire procedure.

7.11 How to Assemble, Link, and Install a Device Handler

Assembling, linking, and installing a new device handler are very simple procedures described in detail in the following sections.

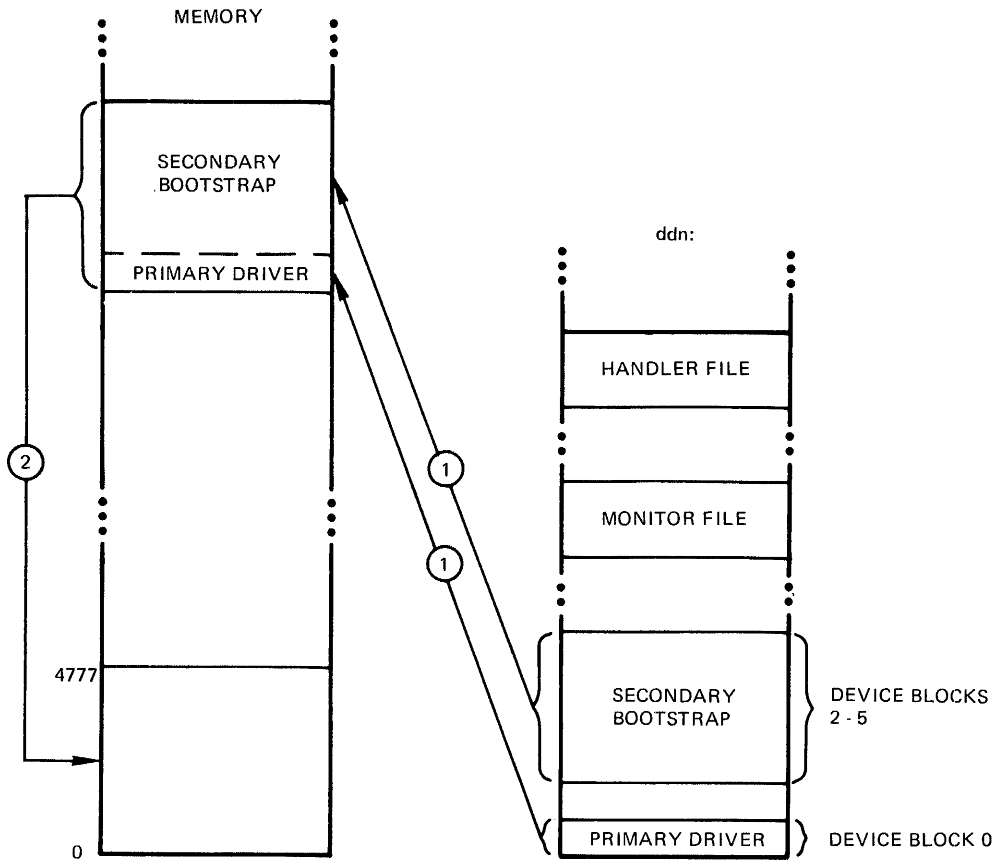
7.11.1 Assembling a Device Handler

Your MACRO–11 source file should be named *dd*.MAC, where *dd* is the two-character device name. For clarity, use the /SHOW:MEB assembler option to print the expansions of macros such as .DRBEG and .DRAST.

To assemble a handler for an SJ or FB system, use the following command:

```
MACRO/CROSSREFERENCE/SHOW:MEB/LIST  SYCND+dd/OBJECT
```

Figure 7–9: BOOT ddn: Procedure



To assemble a handler for an XM system, use the following command:

```
MACRO/CROSSREFERENCE/SHOW:MEB/LIST  XM+SYCND+dd/OBJECT:ddX
```

XM is a source file distributed with RT–11 that indicates that the extended memory feature is present, as is support for the foreground/background environment.

SYCND is the system conditional file. If your system was produced through the system generation process, you must use this file when you assemble your handler so that the handler conditionals will match the monitor conditionals and the handler will operate in the correct environment. Omit this file if you are assembling a device handler that will run with a distributed RT–11 monitor, or use the SYCND.DIS file that is part of the distribution kit.

7.11.2 Linking a Device Handler

Once your source file assembles without errors, you are ready to link it. To link a handler for an SJ or FB system, use the following command:

```
LINK/MAP/EXECUTE:dd.SYS  dd
```

To link a handler for an XM system, use the following command:

```
LINK/MAP/EXECUTE:ddX.SYS  ddX
```

7.11.3 Installing a Device Handler

Before you can use your new handler, you must add information about it to the monitor device tables described in Chapter 3 of this 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 7.11.3.6 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.

7.11.3.1 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.

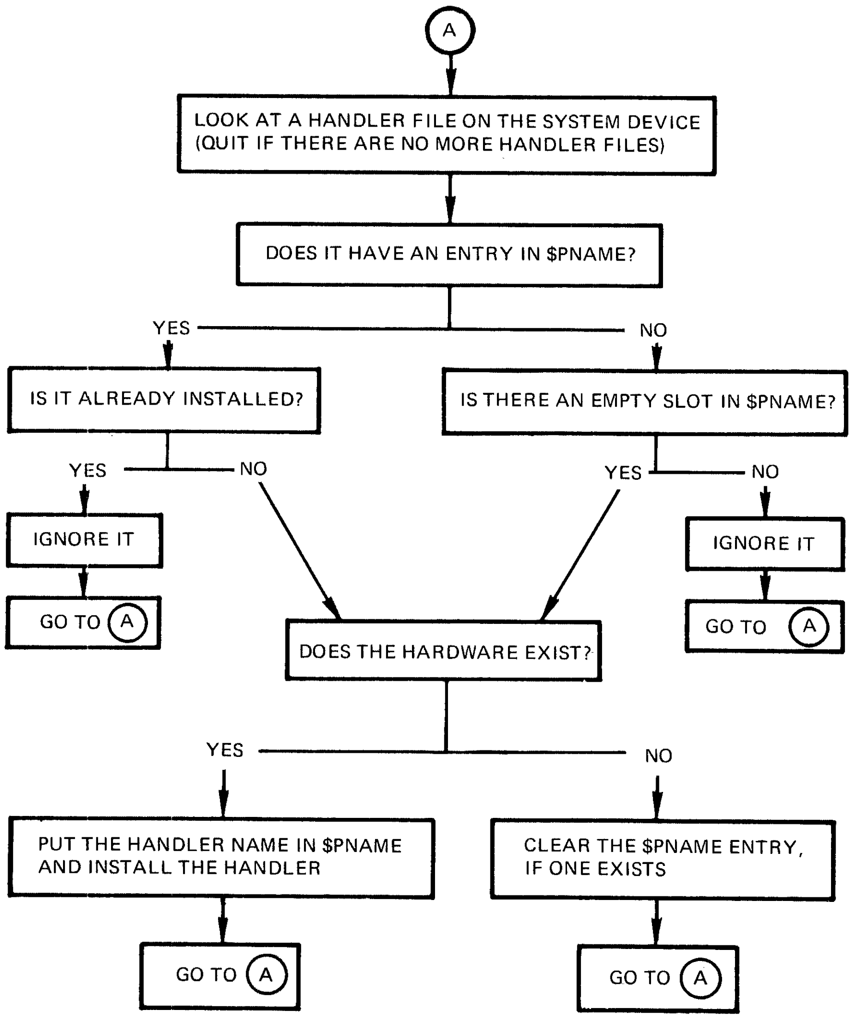
The only difficulty with this procedure occurs when there are more handler files than device slots. A distributed monitor reserves one device slot for each device RT–11 supports. A monitor you create through system generation reserves one slot for each device you request. In addition, it provides the 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 7–10 summarizes the algorithm the bootstrap uses to install device handlers.

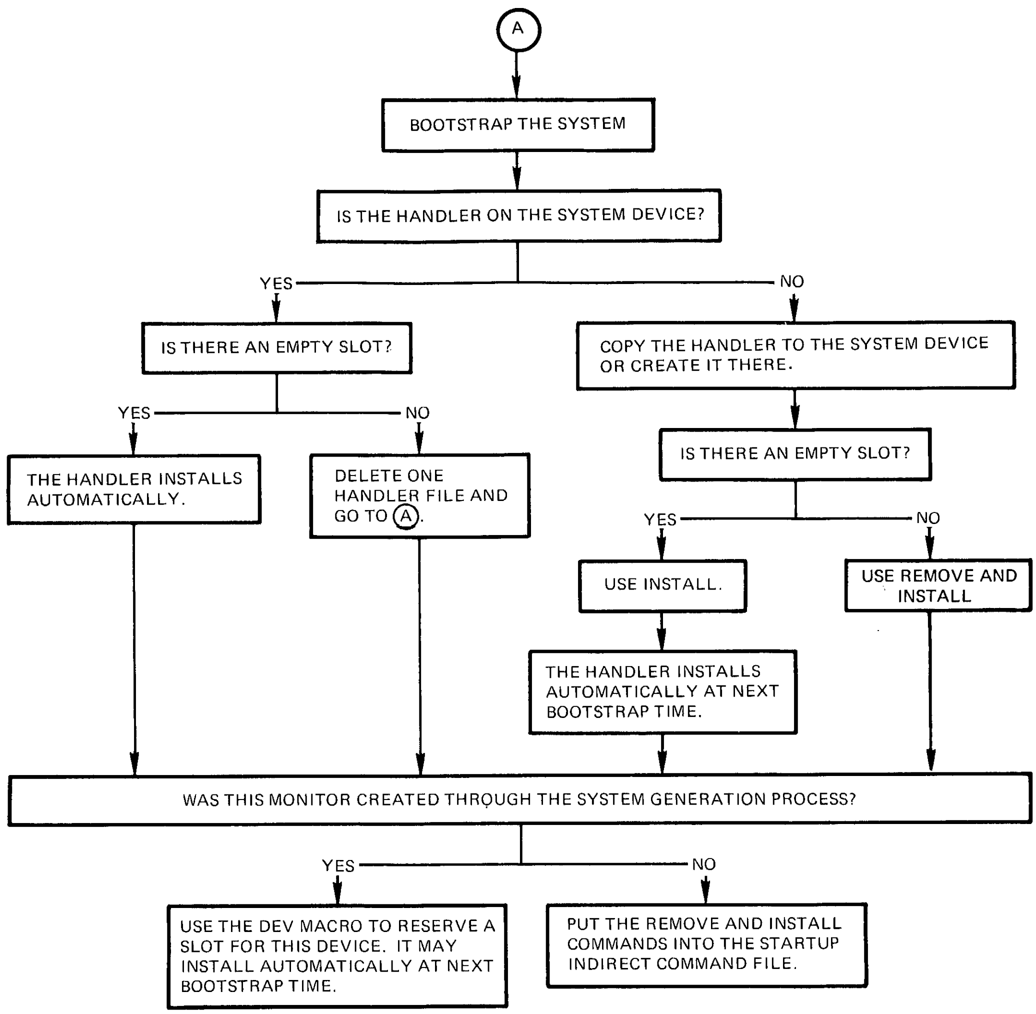
Figure 7–10: Bootstrap Algorithm for Installing Device Handlers



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 7.11.3.2) 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 7.11.3.3) to reserve a slot for a new device handler and give it priority for installation at bootstrap time. Figure 7–11 summarizes the ways you can install a new device handler.

Figure 7–11: Installing a New Device Handler



7.11.3.2 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 add by using the INSTALL command. The formats of these commands are documented in Chapter 4 of the *RT–11 System User's Guide.*

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 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.

7.11.3.3 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 SYSTBL.MAC, one of the files you assemble to create a monitor file.

Use the DEV macro in the file SYSTBL.MAC to add a new device to the system permanently. The format of the DEV macro is as follows:

DEV name,s

name is the two-character device name.

s represents the device status word (leave this argument blank).

The following examples are taken from the SYSTBL file:

```
DEV     RK              ;INSTALLS THE RK DISK
DEV     LP              ;INSTALLS THE LINE PRINTER
DEV     MT              ;INSTALLS MAGTAPE
```

After you edit SYSTBL.MAC to add the DEV macro call for your device, you must reassemble it. Use the following command:

```
MACRO/OBJECT:TBxx  xx+SYCND+SYSTBL
```

xx represents *SJ*, *FB*, or *XM*. SJ.MAC, FB.MAC, and XM.MAC are files distributed with RT–11; they define conditional assembly parameters which indicate whether or not foreground/background processing is permitted. XM.MAC also indicates that extended memory support is present. 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.

7.11.3.4 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 CSR for the device. If the hardware for the device is not present on the system, a bus time-out 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-Illegal 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 doesn't 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 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.

7.11.3.5 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 has registers R0 and R1 available for your use; other registers must be saved and restored.

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 356. 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
        .
        .
        .
ERROR:  SEC             ;SET CARRY TO PREVENT INSTALLATION
        RTS     PC      ;AND RETURN
```

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
                        ;FROM 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 reponds 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, SO 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.

7.11.3.6 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 bootstrap and INSTALL routines by running SIPP. You clear locations 176 and 200 in the handler file's block 0, then use the INSTALL command or reboot the system to install the device handler.

7.12 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 ODT to observe the handler as it processes a data transfer. Sections 7.12.1 and 7.12.2 describe how to do this.

2. Test the handler with keyboard monitor commands, with system utility programs, and with FORTRAN IV or BASIC–11. 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 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. If you are lucky, it will work perfectly the first time!

NOTE

> Handlers for magtape devices are slightly more difficult to interface to the system, since MDUP (which you need to build a bootable magtape) does not immediately recognize devices other than those supported by RT–11. See the *RT–11 Installation and System Generation Guide* for instructions on rebuilding MDUP; see the *RT–11 System Release Notes* for patches to DUP and MDUP. (Other utilities can use the new magtape handler.)

7.12.1 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 the SJ monitor, it is worthwhile to switch to FB 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. If you are using a VT11 or VS60 display terminal, issue the GT ON command now. This puts the GT handler high in memory, just under the Resident Monitor.

(The examples shown in this discussion were created by testing the DX handler on a PDP–11/05 with 28K words of memory and a VT11 display terminal.)

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.

7.12.2 Using ODT in XM

By following just a few special guidelines you can use ODT to debug an XM device handler.

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 7.12.1.

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 current mode you expect at the time the breakpoint occurs. The values are 11 for user mode, 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 ODT to perform an .EXIT request, 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.

7.13 Contents of .SYS Image of a Device Handler

Figure 7–12 below shows the layout of the .SYS image of a handler, after assembly and linking. Locations not otherwise identified are reserved for future use by DIGITAL.

Figure 7–12: Device Handler .SAV Image

Location	
0	
52:	Size of handler in bytes (*dd*END-*dd*STRT)
54:	Size of device in blocks (*dd*SIZE)
56:	Device status word (*dd*STS)
60:	SYSGEN options word
62:	Pointer to start of primary driver (from .DRBOT)
64:	Length of primary driver, in bytes (from .DRBOT)
66:	Offset from start of primary driver to start of bootstrap read routine (from .DRBOT)
70:	Offset to handler data area
110:	Release name in Radix–50
112:	Version number(s); –1 to terminate list
176:	CSR address (*dd*$CSR)
200:	Start of data device installation code, if any (or 0)
202:	Start of system device installation code, if any
356:	High limit of installation code
360:	Reserved for memory use bitmap (360–377)
400:	Start of SET option code (from .DRSET)
776:	High limit of SET option code
1000:	Vector address (*dd*$VEC) (from .DRBEG)
1002:	*dd*INT-. (from .DRBEG)
1004:	New PSW (from .DRBEG)
1006:	*dd*LQE (from .DRBEG)
1010:	*dd*CQE (from .DRBEG)
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$END = . (from .DREND; end of device handler
dd$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)

Figure 7–12: Device Handler .SAV Image

*dd*END:		
*dd*BOOT:	NOP	Start of primary bootstrap (from .DRBOT)
	BR entry	Label *entry* from .DRBOT
entry–14	020	(from .DRBOT)[1]
entry–12	Controller types	(from .DRBOT)[2]
entry–10	020	(from .DRBOT)[3]
entry–6	checksum	(from .DRBOT)[4]
entry–4	0	(from .DRBOT)
entry–2	diskette type	(from .DRBOT)[5]
entry:	BR . + 2 or BMI . + 2	(from DRBOT)[6]
	Start of primary bootstrap read routine	
662:	High limit of primary bootstrap	
664:	Start of bootstrap error code	
776:	End of bootstrap error code	

1 This byte identifies the type of CPU. A value of 20 indicates a PDP–11.

2 This byte indicates the type of controllers that the operating system 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 bus controller
110	MSCP UNIBUS controller
120	MSCP LSI–11 bus controller

3 This byte identifies the type of file structure on the disk. A value of 20 indicates RT–11 file structure.

4 The checksum byte is a checksum of the previous three bytes. It is computed as the complement of the sum of the bytes.

5 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

6 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.

Chapter 8
File Formats

This chapter describes the formats of various RT–11 files. It contains information on the following file types:

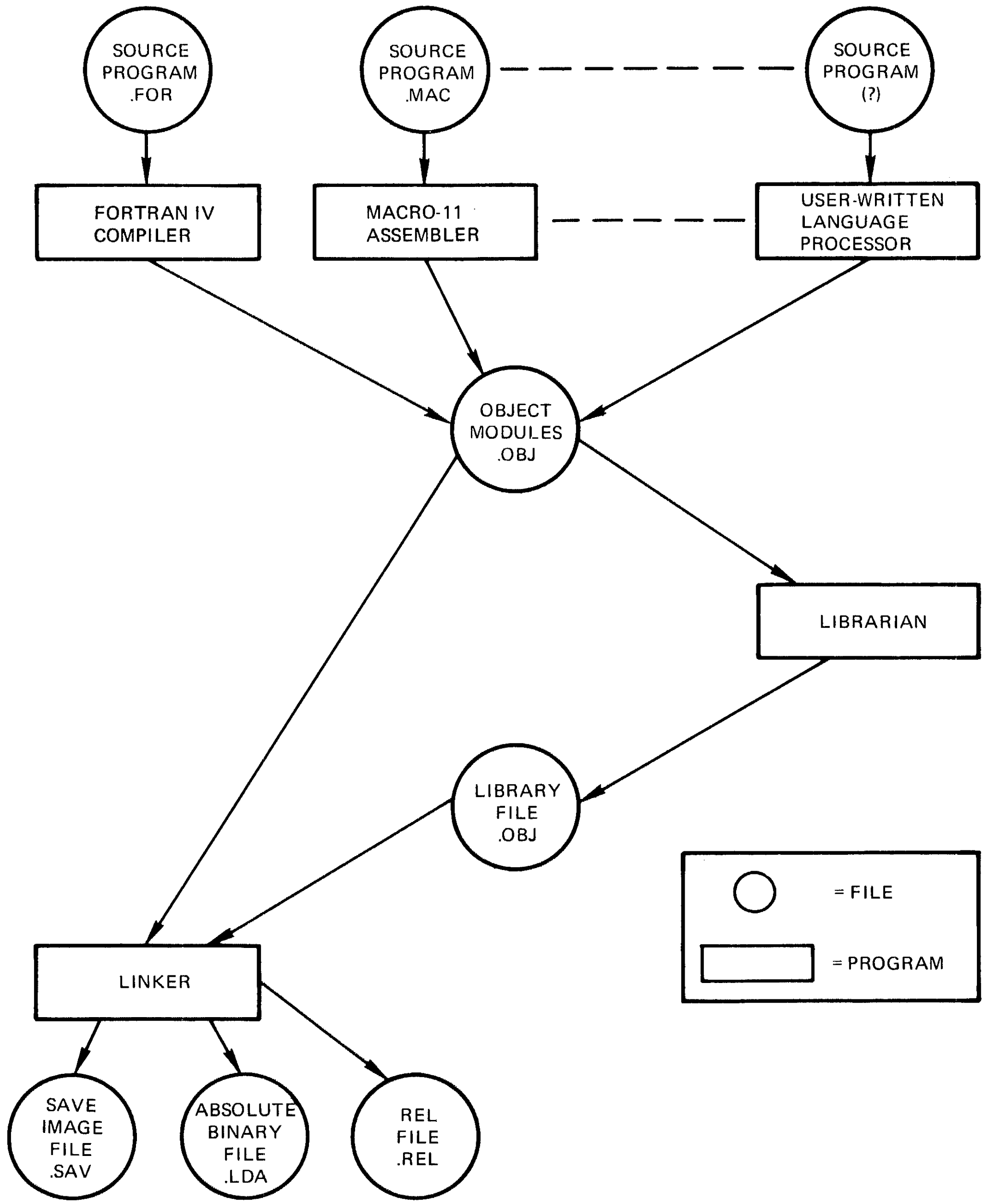
- Object files (OBJ)
- Symbol table definition files (STB)
- Library files (OBJ and MAC)
- Absolute binary files (LDA)
- Save image files (SAV)
- Relocatable files (REL)
- Stream ASCII files (such as MAC, FOR, and so on)
- CREF files
- Error log files

8.1 Object File Format (OBJ)

An object module is a file containing a program or routine in a binary, relocatable form. Object files normally have an .OBJ file type. In a MACRO–11 program, one module is defined as the unit of code enclosed by the pair of directives. MACRO–11 takes the module name from the statement. Language processors, such as MACRO–11 and FORTRAN IV, produce object modules; the linker processes object modules to make runnable programs in SAV, LDA, or REL format. The librarian can also process object files to produce library files, which the linker can then use. Figure 8–1 illustrates object module processing.

Although you can combine many different object modules to form one file, each object module remains complete and independent. However, when the librarian combines object modules into a library, the modules are no longer independent. Instead, they are concatenated and become part of the library's structure. The librarian concatenates modules by byte rather than by word in order to save space. For example, suppose a library is to consist of two modules, and the first module contains an odd number of bytes. The librarian adds the second module to the library behind the first module and positions the first byte of the second module as the high-order byte of the last word of the first module. As a result of this procedure one byte is saved in the library.

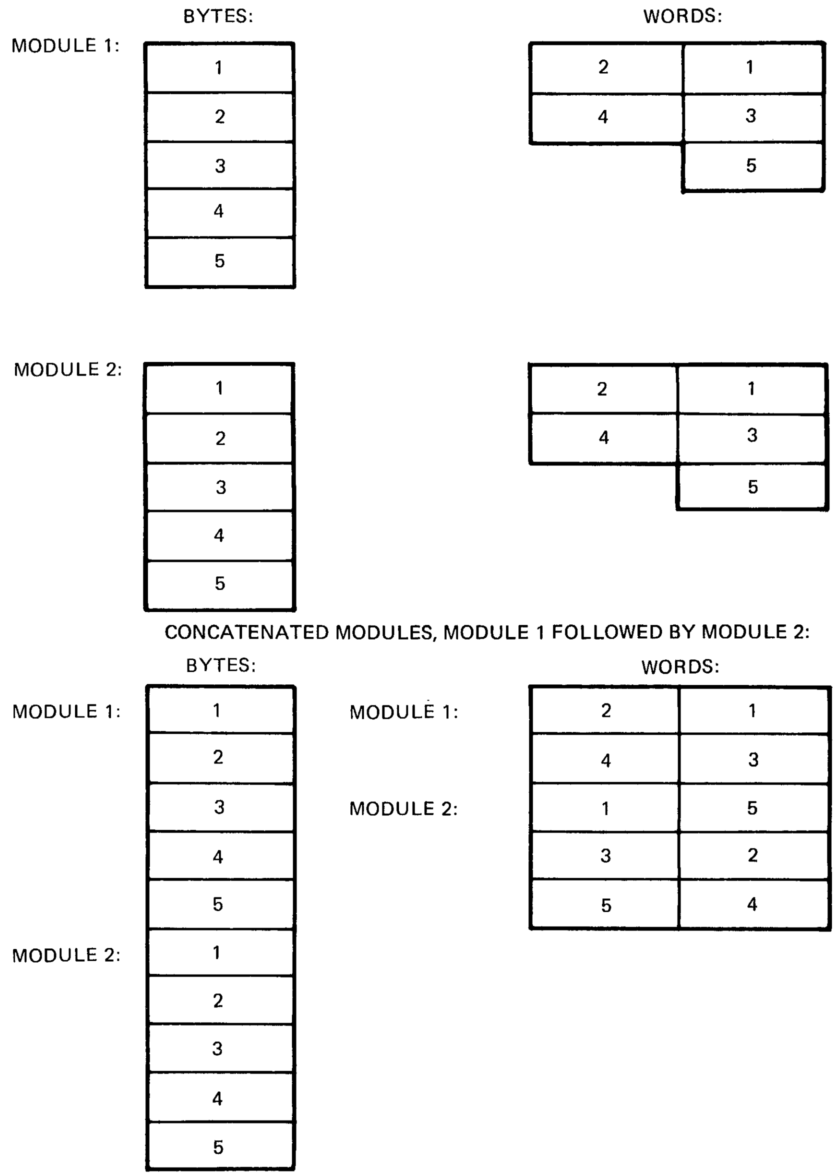
Figure 8–1: Object Module Processing



To understand byte concatenation, it is most helpful to think of the modules as a stream of bytes, rather than as a stream of two-byte words. Figure 8–2 shows how two five-byte modules would be concatenated. Module 1 and module 2 are shown both as bytes and as words.

The rest of Section 8.1 contains information on the composition of object modules that is more detailed than most programmers require. However, if you intend to write a language processor, a linker program, or a program to dump and interpret object modules, you should read this material carefully.

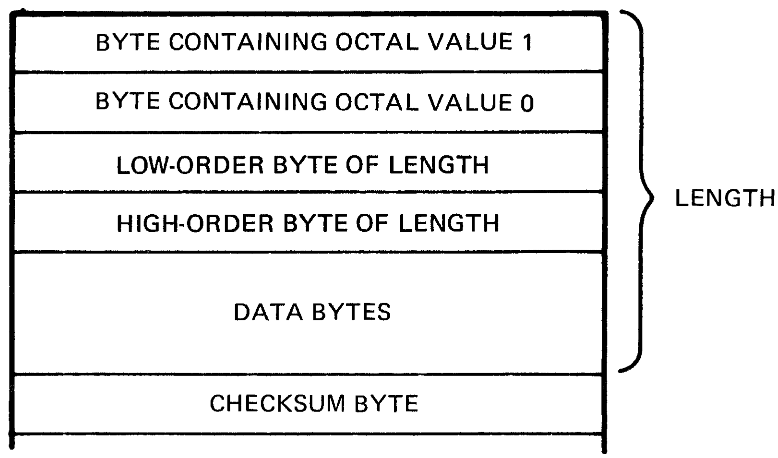
Figure 8–2: Modules Concatenated by Byte



If you are writing a language processor and want its output to be processed by the RT–11 linker, be sure that the processor produces object modules compatible with those described here. Since this section documents the object modules produced by MACRO–11 and FORTRAN IV, you could also use this information to write your own linker program.

Object modules are made up of formatted binary blocks. A formatted binary block is a sequence of eight-bit bytes (stored in an RT–11 file, on paper tape, or by some other means) that is arranged as shown in Figure 8–3.

Figure 8–3: Formatted Binary Format



Each formatted binary block has its length stored within it. The length includes all bytes of the block except the checksum byte. The checksum byte is the negative of the sum of all preceding bytes. Formatted binary blocks may be separated by a variable number of null (0) bytes.

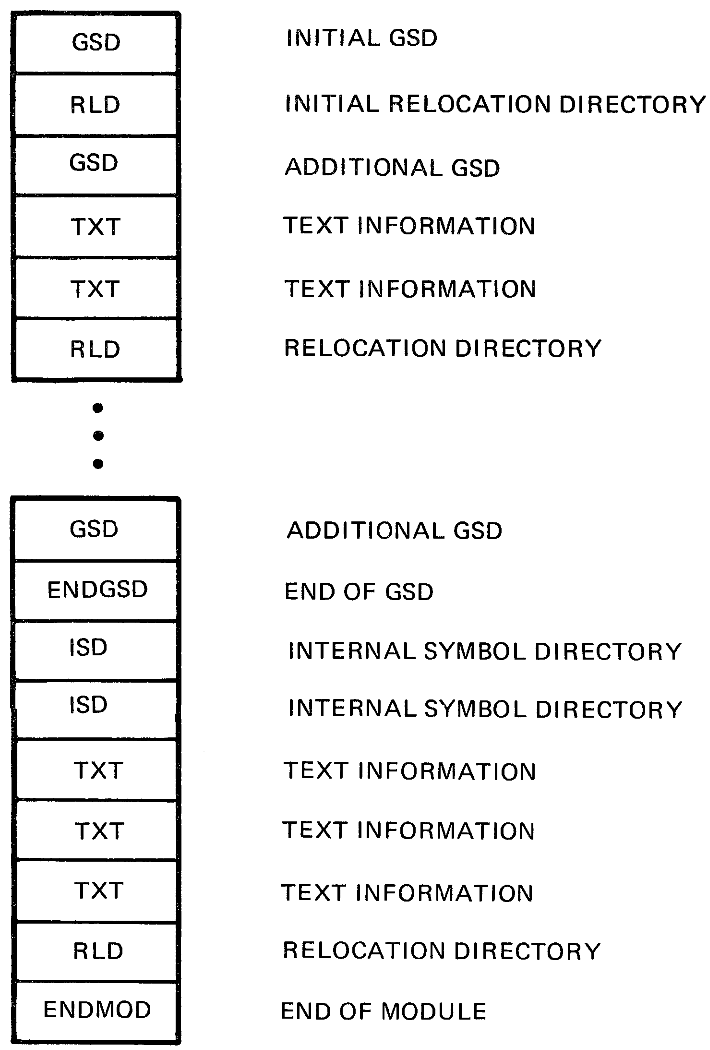
The "data bytes" portion of each formatted binary block contains the actual object module information. RT–11 uses and recognizes eight types of data blocks. The information in these blocks guides the linker as it translates object code into a runnable program. Table 8–1 lists the eight types of data blocks.

Table 8–1: RT–11 Data Blocks

Identification Code	Block Type	Function
1	GSD	Holds the Global Symbol Directory information
2	ENDGSD	Signals the end of Global Symbol Directory blocks in a module
3	TXT	Holds the actual binary text of the program
4	RLD	Holds Relocation Directory information
5	ISD	Holds the Internal Symbol Directory information (not supported by RT–11)
6	ENDMOD	Signals the end of the object module
7	Librarian header	Holds the status of the library file (see Section 8.3.1)
10	Librarian end	Signals the end of the library file (see Section 8.3.3)

An object module must begin with a Global Symbol Directory (GSD) block and end with an End of Module (ENDMOD) block. Additional GSD blocks can occur anywhere in the file, but must appear before an End of Global Symbol Directory (ENDGSD) block. An ENDGSD block must appear before the ENDMOD block, and at least one Relocation Directory (RLD) block must appear before the first Text Information (TXT) block. Additional RLD and TXT blocks can appear anywhere in the file. The Internal Symbol Directory (ISD) block can appear anywhere in the file between the initial GSD and ENDMOD blocks. Figure 8–4 shows a general scheme for an object module.

Figure 8–4: General Object Module Format



You must declare all program sections (PSECTs, VSECTs, and CSECTs) defined in a module in GSD items. The size word of each program section definition should contain the size in bytes to be reserved for the section. If you declare a program section more than once in a single object module, the linker uses the largest declared size for that section. All global symbols that are defined in a given program section must appear in the GSD items immediately following the definition item of that program section.

A special program section, called the absolute section (. **ABS**), is allocated by the linker beginning at location 0 of memory. Immediately after the GSD item that defines the absolute section, declare all global symbols that contain absolute (non-relocatable) values. If you do not want to allocate any memory space for the absolute section, specify zero as its size word. You can do this even if absolute global symbol definitions occur after it.

Global symbols that are referenced but not defined in the current object module must also appear in GSD items. These global references may appear in any GSD item except the very first, which contains the module name. In MACRO, referenced globals are seen in a GSD block under the . ABS. p-sect. They always have the p-sect definition preceding them.

Note that when a 16-bit word is stored as part of the information in a data block, it is always stored as two consecutive eight-bit bytes, with the low-order byte first.

Object module data blocks vary in length. The first byte in a data block is a code that identifies the type of data block. The codes range from 0 through 10 octal, as Table 8–1 shows. The format of the rest of the information in the data block depends on the type of data block.

The following sections describe in detail the format of the data blocks.

8.1.1 Global Symbol Directory Block (GSD)

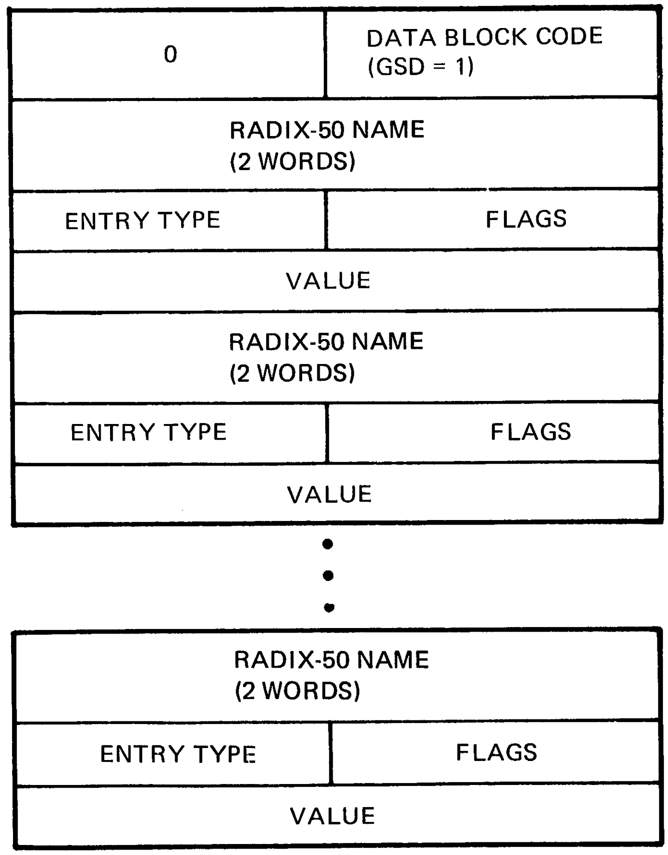
Global Symbol Directory blocks contain all the information the linker needs to assign addresses to global symbols and to allocate the memory a job requires. Table 8–2 shows the eight types of entries that GSD blocks can contain.

Table 8–2: Entries in GSD Blocks

Entry Type	Description
0	Module Name
1	Control Section Name (CSECT)
2	Internal Symbol Name
3	Transfer Address
4	Global Symbol Name
5	Program Section Name
6	Program Version Identification (IDENT)
7	Mapped Array Declaration (VSECT)

Each entry type is represented by four words in the GSD data block. The first two words contain six Radix–50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. Figure 8–5 illustrates the format of the GSD data block and the entry types.

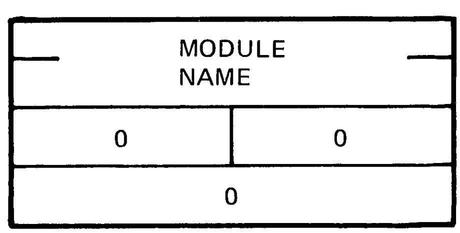
Figure 8–5: Global Symbol Directory Data Block



The following sections describe the entry types for GSD data blocks.

8.1.1.1 Module Name (Entry Type 0) – The module name entry (see Figure 8–6) declares the name of the object module. The name need not be unique with respect to other object modules because modules are identified by file, not module name. However, only one module name declaration can occur in a single object module.

Figure 8–6: Module Name Entry Format (Entry Type 0)



8.1.1.2 Control Section Name (Entry Type 1) – The control section name entry (see Figure 8–7) declares the name of a control section. The linker converts control sections – which include ASECTs, blank CSECTs, and named CSECTs – to p-sects. For compatibility with other systems, the linker processes ASECTs and both forms of CSECTs. See Section 8.1.1.6 for the entry the linker generates for a .PSECT statement.

You can define ASECT and CSECT statements in terms of PSECT statements, as follows:

For a blank CSECT, define a p-sect with the following attributes:

.PSECT ,RW,I,LCL,REL,CON

For a named CSECT, the p-sect definition is:

.PSECT name,RW,I,GBL,REL,OVR

For an ASECT, the p-sect definition is:

.PSECT . ABS.,RW,I,GBL,ABS,OVR

The linker processes ASECTs and CSECTs as PSECTs with the fixed attributes defined above.

Figure 8–7: Control Section Name Entry Format (Entry Type 1)

CONTROL SECTION NAME	
1	IGNORED
MAXIMUM LENGTH	

8.1.1.3 Internal Symbol Name (Entry Type 2) – The internal symbol name entry (see Figure 8–8) declares the name of an internal symbol with respect to the module. Because the linker does not support internal symbol tables, the detailed format of this entry is not defined. If the linker encounters an internal symbol entry while reading the GSD, it ignores it.

Figure 8–8: Internal Symbol Name Entry Format (Entry Type 2)

SYMBOL NAME	
2	0
UNDEFINED	

8.1.1.4 Transfer Address (Entry Type 3) – The transfer address entry (see Figure 8–9) declares the transfer address of a module relative to a p-sect. The first two words of the entry define the name of the p-sect. The fourth word indicates the relative offset from the beginning of that p-sect. If no transfer address is declared in a module, the transfer address entry must not be included in the GSD, or else a transfer address 000001 relative to the default absolute p-sect (. ABS.) must be specified.

To begin execution of a program within a particular object module of a program, specify the starting address to the linker as the transfer address. The linker passes the first even transfer address it encounters to RT–11 as the program's starting address. Whenever the resulting program executes, the start address indicates the first executable instruction. If there is no transfer address (if, for example, you did not specify one with the .END directive in a MACRO–11 program), or if all transfer addresses are odd, the resulting program does not self-start when you run it.

Figure 8–9: Transfer Address Entry Format (Entry Type 3)

SYMBOL NAME	
3	0
OFFSET	

NOTE

When the p-sect is absolute, **Offset** is the actual transfer address if it is not equal to 000001.

8.1.1.5 Global Symbol Name (Entry Type 4) – The global symbol name entry (see Figure 8–10) declares either a global reference or a definition. All definition entries must appear after the declaration of the p-sect under which they are defined, and before the declaration of another p-sect. Global references can appear anywhere within the GSD.

Figure 8–10: Global Symbol Name Entry Format (Entry Type 4)

SYMBOL NAME	
4	FLAGS
VALUE	

The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol. The fourth word contains the value of the symbol relative to the p-sect under which it is defined.

The flag byte of the symbol declaration entry has the bit assignments shown in Table 8–3. Bits 1, 2, 4, 6, and 7 are not used by the RT–11 linker.

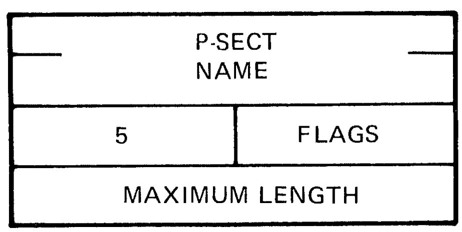
Table 8–3: Flag Bits for Global Symbol Name Entry

Bit	Meaning
0	Weak Qualifier 0 = Strong (normal) symbol 1 = Weak symbol
3	Definition 0 = Global symbol reference 1 = Global symbol definition
5	Relocation 0 = Absolute symbol value 1 = Relative symbol value

8.1.1.6 Program Section Name (Entry Type 5) — The p-sect name entry (see Figure 8–11) declares the name of a p-sect and its maximum length in the module. It also uses the flag byte to declare the attributes of the p-sect. The default attributes of the p-sect (blank or named with no attributed specified) are as follows:

.PSECT ,RW,I,LCL,REL,CON

Figure 8–11: P-sect Name Entry Format (Entry Type 5)



NOTE

The length of all absolute sections is zero.

GSD records must be constructed in such a way that once a p-sect name has been declared, all global symbol definitions pertaining to it must appear before another p-sect name is declared. Global symbols are declared by means of symbol declaration entries. Thus, the normal format is a series of p-sect names, each followed by optional symbol declarations.

Table 8–4 shows the bit assignments of the flag byte. Bits 1 and 3 are not used by the RT–11 linker.

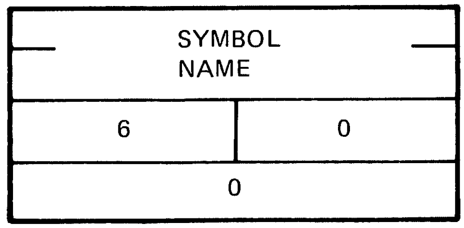
Table 8–4: Flag Bits for P-sect Name Entry

Bit	Meaning
0	Save (Root Allocation) 0 = P-sect scope is determined by the value of bit 6. 1 = P-sect is allocated in the root and its scope is global regardless of the value of bit 6.
2	Allocation 0 = P-sect references are to be concatenated with other references to the same p-sect to form the total amount of memory allocated to the section. 1 = P-sect references are to be overlaid. The total amount of memory allocated to the p-sect is the size of the largest request made by individual references to the same p-sect.
4	Access (not supported by RT–11 monitors) 0 = P-sect has read/write access. 1 = P-sect has read-only access.
5	Relocation 0 = P-sect is absolute and requires no relocation. 1 = P-sect is relocatable and references to the control section must have a relocation bias added before they become absolute.
6	Scope 0 = The scope of the p-sect is local. References to the same p-sect will be collected only within the overlay segment in which the p-sect is defined. 1 = The scope of the p-sect is global. References to the p-sect are collected across overlay segment boundaries.
7	Type 0 = The p-sect contains instruction (I) references. Concatenation of this p-sect will be by word boundary. Globals will be given overlay control blocks. 1 = The p-sect contains data (D) references. Concatenation of this p-sect will be by byte boundary. Globals will not go through the overlay handler.

8.1.1.7 Program Version Identification (Entry Type 6) — The program version identification entry (see Figure 8–12) declares the version of the module. The linker saves the version identification, or IDENT, of the first module that defines a nonblank version. It then includes this identification on the memory allocation map.

The first two words of the entry contain the version identification. The linker does not use either the flag byte or the fourth word because they contain no meaningful information.

Figure 8–12: Program Version Identification Entry Format (Entry Type 6)



8.1.1.8 Mapped Array Declaration (Entry Type 7) — The mapped array declaration (see Figure 8–13) allocates space within the mapped array area of the job's memory. The linker adds the array name to the list of p-sect names, and subsequent RLD blocks can reference it. The linker adds the length (in units of 32-word blocks) to the job's mapped array allocation. It rounds up the total amount of memory allocated to each mapped array to the nearest 256-word boundary. The contents of the flag byte are reserved and assumed to be zero. (Only the FORTRAN IV compiler produces this VSECT.)

The linker processes a VSECT as a PSECT with the following attributes:

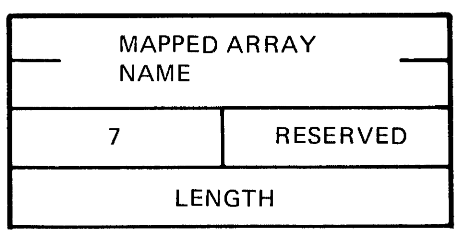
.PSECT . VIR.,RW,D,GBL,REL,CON

The size is equal to the number of 32-word blocks required. If the length is zero, the segment is the root. There must never be any globals under this section, which starts at a base of 0.

NOTE

> One additional address window is allocated whenever a mapped array is declared.

Figure 8–13: Mapped Array Declaration Entry Format (Entry Type 7)



8.1.2 End of Global Symbol Directory Block (ENDGSD)

The end of global symbol directory block (see Figure 8–14) declares that no other GSD blocks are contained in this module. Exactly one end of GSD block must appear in every object module. The length of the data block is one word.

Figure 8–14: End of GSD Data Block

0	DATA BLOCK CODE (ENDGSD = 2)

8.1.3 Text Information Block (TXT)

The text information block (see Figure 8–15) contains a byte string of information that the linker writes directly into the output file. The block consists of a load address followed by the byte string.

Text records can contain words or bytes of information whose final contents have not yet been determined. This information will be bound by a relocation directory block that immediately follows the text block. If the text block does not need modification, then no relocation directory block is needed. Thus, multiple text blocks can appear in sequence before a relocation directory block.

The load address of the text block is specified as an offset from the current p-sect base. At least one relocation directory block must precede the first text block. This RLD block must declare the current p-sect.

Figure 8–15: Text Information Data Block

0	DATA BLOCK CODE (TXT = 3)
LOAD ADDRESS	
TEXT	TEXT
TEXT	TEXT
TEXT	TEXT

•
•
•

TEXT	TEXT
TEXT	TEXT
TEXT	TEXT

8.1.4 Relocation Directory Block (RLD)

Relocation directory blocks (see Figure 8–16) contain the information the linker needs to relocate and link the preceding text information block. Every module must have at least one relocation directory block that precedes the first text information block. The first block does not modify a preceding text block. Instead, it defines the current p-sect and location.

Figure 8–16: Relocation Directory Data Block

0	DATA BLOCK CODE (RLD = 4)
DISPLACEMENT	COMMAND
INFORMATION	INFORMATION
INFORMATION	INFORMATION
INFORMATION	INFORMATION

•
•
•

COMMAND	INFORMATION
INFORMATION	DISPLACEMENT
INFORMATION	INFORMATION
INFORMATION	INFORMATION
INFORMATION	INFORMATION
DISPLACEMENT	COMMAND
INFORMATION	INFORMATION
INFORMATION	INFORMATION
INFORMATION	INFORMATION

Relocation directory blocks can contain 14 types of entries. These entries are classified as relocation, or location modification entries. Table 8–5 lists the valid entry types.

Each type of entry is represented by a command byte, which specifies the type of entry and the word or byte modification. This byte is followed by a displacement byte and then by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the preceding text information block, yields the virtual address in the image that is to be modified. The command byte of each entry has the bit assignments shown in Table 8–6. The following sections describe the valid entry types for the RLD data block.

Table 8–5: Valid Entry Types for RLD Blocks

Entry Type	Description
1	Internal Relocation
2	Global Relocation
3	Internal Displaced Relocation
4	Global Displaced Relocation
5	Global Additive Relocation
6	Global Additive Displaced Relocation
7	Location Counter Definition
10	Location Counter Modification
11	Program Limits (.LIMIT)
12	P-sect Relocation
13	Not used
14	P-sect Displaced Relocation
15	P-sect Additive Relocation
16	P-sect Additive Displaced Relocation
17	Complex Relocation

Table 8–6: Bit Assignments for the RLD Command Byte

Bit	Meaning
0–6	Specify the type of entry. Although there is room to specify 128 command types, only 14 decimal are currently implemented in the RT–11 linker.
7	Modification (the B bit in Figures 8–17 through 8–30). This feature is not supported by RT–11, and the bit is ignored if set. The RT–11 linker supports word relocation, not byte relocation. 0 = The command modifies an entire word. 1 = The command modifies only one byte.

8.1.4.1 Internal Relocation (Entry Type 1) — This type of entry (see Figure 8–17) relocates a direct pointer to an address within a module. The linker adds the current p-sect base address to a specified constant and writes the result into the output file at the calculated address — that is, it adds a displacement byte to the value calculated from the load address of the preceding text block.

For example:

```
A:      MOV     #A,R0
```

or

```
        .WORD   A
```

Figure 8–17: Internal Relocation (Entry Type 1)

DISPLACEMENT	B	1
CONSTANT		

8.1.4.2 Global Relocation (Entry Type 2) — This type of entry (see Figure 8–18) relocates a direct pointer to a global symbol. The linker obtains the definition of the global symbol and writes the result into the output file at the calculated address.

For example:

```
MOV     #GLOBAL,R0
```

or

```
.WORD   GLOBAL
```

Figure 8–18: Global Relocation (Entry Type 2)

DISPLACEMENT	B	2
SYMBOL NAME		

8.1.4.3 Internal Displaced Relocation (Entry Type 3) — This type of entry (see Figure 8–19) relocates a relative reference to an absolute address from within a relocatable control section. The linker subtracts from the specified constant the address plus 2 into which the relocated value is to be written. The linker then writes the result into the output file at the calculated address.

For example:

```
CLR     177550
```

or

```
MOV     177550,R0
```

Figure 8–19: Internal Displaced Relocation (Entry Type 3)

DISPLACEMENT	B	3
CONSTANT		

8.1.4.4 Global Displaced Relocation (Entry Type 4) — This type of entry (see Figure 8–20) relocates a relative reference to a global symbol. The linker obtains the definition of the global symbol, and subtracts from the definition value the address plus 2 into which the relocated value is to be written. It then writes the result into the output file at the calculated address.

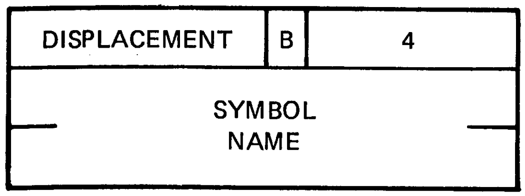
For example:

```
        CLR     GLOBAL
```

or

```
        MOV     GLOBAL,R0
```

Figure 8–20: Global Displaced Relocation (Entry Type 4)



8.1.4.5 Global Additive Relocation (Entry Type 5) — This type of entry (see Figure 8–21) relocates a direct pointer to a global symbol with an additive constant. The linker obtains the definition of the global, adds the specified constant, and then writes the resultant value into the output file at the calculated address.

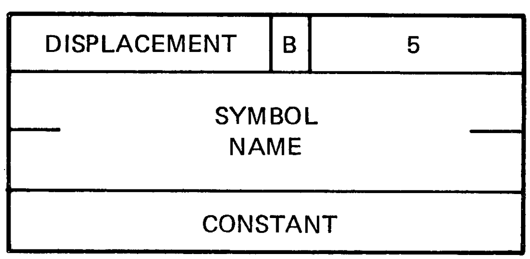
For example:

```
        MOV     #GLOBAL+2,R0
```

or

```
        .WORD   GLOBAL-4
```

Figure 8–21: Global Additive Relocation (Entry Type 5)



8.1.4.6 Global Additive Displaced Relocation (Entry Type 6) – This type of entry (see Figure 8–22) relocates a reference to a global symbol with an additive constant. The linker obtains the definition of the global symbol and adds the specified constant to the definition value. The linker subtracts from the resultant additive value the address plus 2 into which the relocated value is to be written. It then writes the result into the output file at the calculated address.

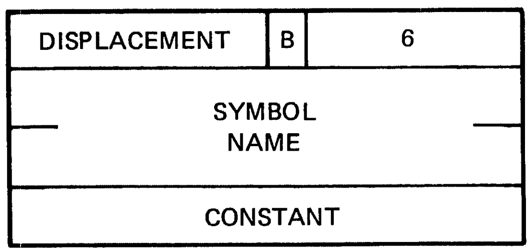
For example:

```
        CLR     GLOBAL+2
```

or

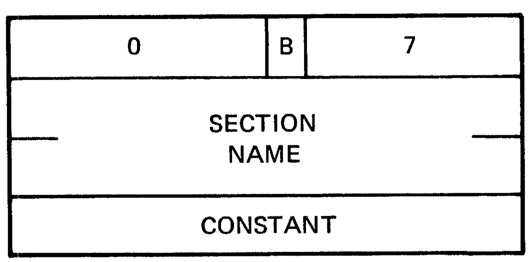
```
        MOV     GLOBAL-5,R0
```

Figure 8–22: Global Additive Displaced Relocation (Entry Type 6)



8.1.4.7 Location Counter Definition (Entry Type 7) – This type of entry (see Figure 8–23) declares a current p-sect and location counter value. The linker stores the control base as the current control section. It adds the current control section base to the specified constant and stores the result as the current location counter value.

Figure 8–23: Location Counter Definition (Entry Type 7)



8.1.4.8 Location Counter Modification (Entry Type 10) – This type of entry (see Figure 8–24) modifies the current location counter. The linker adds the current p-sect base to the specified constant and stores the result as the current location counter.

For example:

```
.=.+N
```

or

```
.BLKB    N
```

Figure 8–24: Location Counter Modification (Entry Type 10)

0	B	10
CONSTANT		

8.1.4.9 Program Limits (Entry Type 11) — The .LIMIT assembler directive generates this type of entry (see Figure 8–25). The linker obtains the first address above the header, which is normally the beginning of the stack, and the highest address allocated to the job. It then writes these addresses into the output file at the calculated address and the following location, respectively.

For example:

```
.LIMIT
```

Figure 8–25: Program Limits (Entry Type 11)

DISPLACEMENT	B	11

8.1.4.10 P-sect Relocation (Entry Type 12) — This type of entry (see Figure 8–26) relocates a direct pointer to the beginning address of another p-sect (other than the p-sect in which the reference is made) within a module. The linker obtains the current base address of the specified p-sect and writes it into the output file at the calculated address.

For example:

```
        .PSECT    A
B:
        .
        .
        .
        .PSECT    C
        MOV       #B,R0
```

or

```
        .WORD     B
```

Figure 8–26: P-sect Relocation (Entry Type 12)

DISPLACEMENT	B	12
SECTION NAME		

8.1.4.11 P-sect Displaced Relocation (Entry Type 14) — This type of entry (see Figure 8–27) relocates a relative reference to the beginning address of another p-sect within a module. The linker obtains the current base address of the specified p-sect. It then subtracts from the base value the address plus 2 into which the relocated value is to be written and writes the result into the output file at the calculated address.

For example:

```
        .PSECT A
B:
        .
        .
        .
        .PSECT C
        MOV     B,R0
```

Figure 8–27: P-sect Displaced Relocation (Entry Type 14)

DISPLACEMENT	B	14
SECTION NAME		

8.1.4.12 P-sect Additive Relocation (Entry Type 15) — This type of entry (see Figure 8–28) relocates a direct pointer to an address in another p-sect within a module. The linker obtains the current base address of the specified p-sect. It adds the base to the specified constant and then writes the result into the output file at the calculated address.

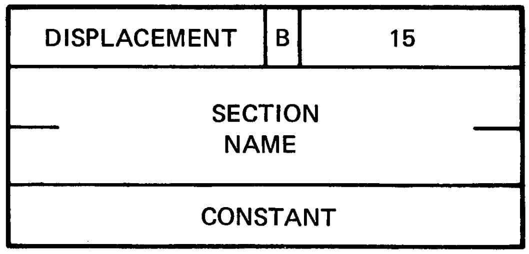
For example:

```
        .PSECT A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT D
        MOV     #B+10,R0
        MOV     #C,R0
```

or

```
        .WORD B+10
        .WORD C
.
```

Figure 8–28: P-sect Additive Relocation (Entry Type 15)

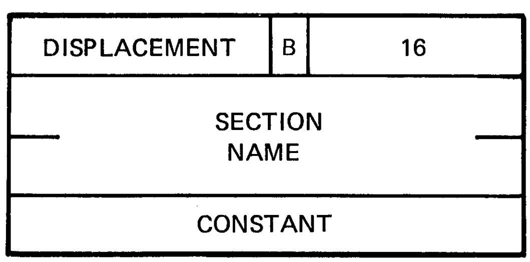


8.1.4.13 P-sect Additive Displaced Relocation (Entry Type 16) — This type of entry (see Figure 8–29) relocates a relative reference to an address in another p-sect within a module. The linker obtains the current base address of the specified p-sect and adds it to the specified constant. Next, it subtracts from the resultant additive value the address plus 2 into which the relocated value is to be written. It writes the final result into the output file at the calculated address.

For example:

```
        .PSECT A
B:
        .
        .
        .
C:
        .
        .
        .
        .PSECT D
        MOV    B+10,R0
        MOV    C,R0
```

Figure 8–29: P-sect Additive Displaced Relocation (Entry Type 16)



8.1.4.14 Complex Relocation (Entry Type 17) — This type of entry (see Figure 8–30) resolves a complex relocation expression. A complex relocation expression is one in which any of the MACRO–11 binary or unary operations are permitted with any type of argument, regardless of whether the argument is unresolved global, relocatable to any p-sect base, absolute, or a complex relocatable subexpression.

The RLD command word is followed by a string of numerically specified operation codes and arguments. The operation codes each occupy one byte and the entire RLD command must fit in a single data block. Table 8–7 shows the list of valid operation codes. Note that complex relocation on foreground links causes a warning message from the linker. The results of complex relocation will be correct if no relocatable symbols are involved, however.

Table 8–7: Operation Codes for Complex Relocation

Code	Description
0	No operation
1	Addition (+)
2	Subtraction (–)
3	Multiplication (*)
4	Division (/)
5	Logical AND (&)
6	Logical inclusive OR (!)
7	Exclusive OR
10	Negation (–)
11	Complement (^C)
12	Store result (command termination)
13	Store result with displaced relocation (command termination)
16	Fetch global symbol. It is followed by four bytes containing the symbol name in Radix–50 representation.
17	Fetch relocatable value. It is followed by one byte containing the sector number, and two bytes containing the offset within the sector.
20	Fetch constant. It is followed by two bytes containing the constant.

The STORE commands (codes 12 and 13) indicate that the value is to be written into the output file at the calculated address.

The linker evaluates all operands as 16-bit signed quantities using two's complement arithmetic. The results are equivalent to expressions that are evaluated internally by the assembler. Note the following rules.

1. An attempt to divide by 0 yields a 0 result. The linker issues a warning message.
2. All results are truncated from the left in order to fit into 16 bits. No diagnostic is issued if the number was too large. If the result modifies a byte, the linker checks for truncation errors. (Byte operations are not allowed.)
3. All operations are perfomed on relocated (additive) or absolute 16-bit quantities. PC displacement is applied to the result only.

For example:

```
        .PSECT  ALPHA
A:
        .
        .
        .
        .PSECT  BETA
B:
        .
        .
        .
        MOV     #A+B-<G1/G2&^C<177120!G3>>,R1
```

Figure 8–30: Complex Relocation (Entry Type 17)

<table>
<tr><td>DISPLACEMENT</td><td>B</td><td>17</td></tr>
<tr><td colspan="3">COMPLEX STRING</td></tr>
<tr><td>12</td><td colspan="2"></td></tr>
</table>

8.1.5 Internal Symbol Directory Block (ISD)

Internal symbol directory blocks (see Figure 8–31) declare definitions of symbols that are local to a module. The linker does not support this feature; therefore, a detailed data block format is not documented here. If the linker encounters this type of data block, it ignores it.

Figure 8–31: Internal Symbol Directory Data Block

<table>
<tr><td>0</td><td>DATA BLOCK CODE
(ISD = 5)</td></tr>
<tr><td colspan="2">NOT SPECIFIED</td></tr>
</table>

8.1.6 End of Module Block (ENDMOD)

The end of module block (see Figure 8–32) declares the end of an object module. Exactly one end of module record must appear in each object module. It is one word long.

Figure 8–32: End of Module Data Block

0	DATA BLOCK CODE (ENDMOD = 6)

8.2 Symbol Table Definition File Format (STB)

The RT–11 linker can produce a symbol table (STB) file as its third output file. The text of the STB file consists of global symbol table definitions. For example, if the source file contains X = = 10, the STB file contains X = 10. Or, if the source file contains A = FOO, the STB file contains the address of FOO.

The STB file can serve as a communication link between a background and a foreground job. This communication comes about when you link the background job and obtain an STB file as output. Then, when you link the foreground job, include the STB file as one of the input files. The foreground job will then be able to reference symbols used by the background job. Similarly, you can use the STB file to create a communication link between a program and a symbolic debugger.

The internal format of the STB file consists entirely of Global Symbol Directory (GSD) data blocks followed by one End of Global Symbol Directory (ENDGSD) data block and one End of Module (ENDMOD) data block. Figure 8–33 illustrates the STB file format.

8.3 Library File Format (OBJ and MAC)

A library file contains concatenated modules and some additional information. RT–11 supports object and macro libraries. Object libraries usually have an .OBJ file type; macro libraries usually have a .MAC file type.

The modules in an object or macro library file are preceded by a Library Header Block and Library Directory, and are followed by the Library End Block, or trailer. Figure 8–34 shows the format of an object or macro library file.

Diagrams of each component in the library file structure are included in the sections that follow. See the *RT–11 System User's Guide* for information on using the librarian.

Figure 8–33: STB File Format

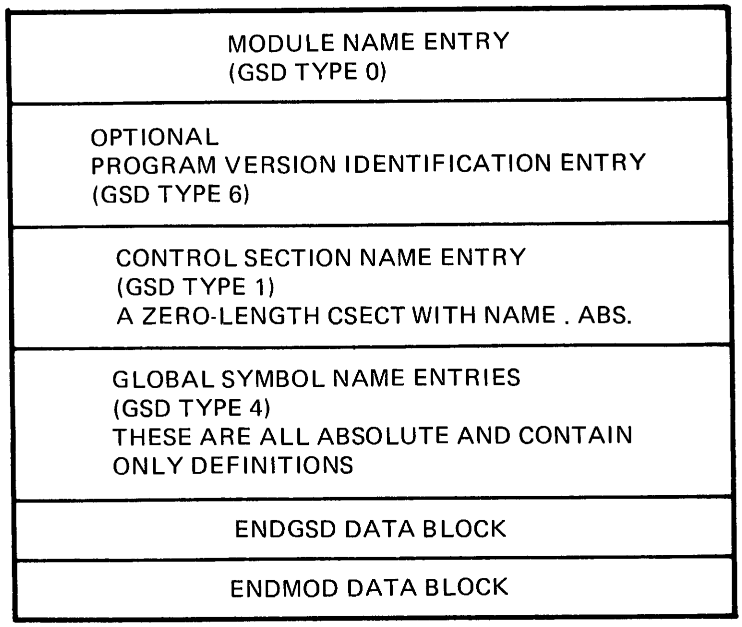
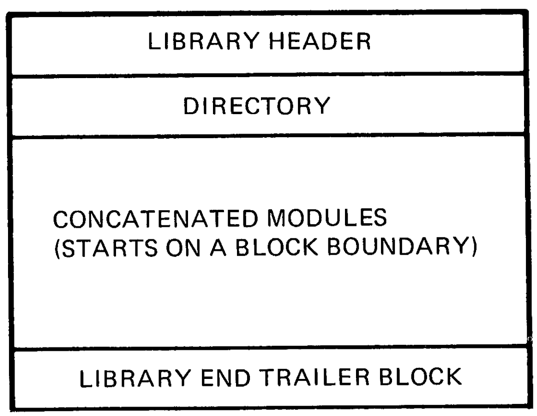


Figure 8–34: Library File Format (OBJ and MAC)



8.3.1 Library Header Format

The library header describes the status of the file. Of the two figures that follow, Figure 8–35 shows the contents of the object library header and Figure 8–36 shows the contents of the macro library header.

All numeric values shown are octal. The date and time, which are in standard RT–11 format, are the date and time the library was created. This information is displayed when the library is listed.

Figure 8–35: Object Library Header Format

OFFSET	CONTENTS	DESCRIPTION
0	1	LIBRARY HEADER BLOCK CODE
2	42	
4	7	LIBRARIAN CODE
6	500	LIBRARY VERSION NUMBER
10	0	RESERVED
12		DATE IN RT-11 FORMAT (0 IF NONE)
14		TIME EXPRESSED IN TWO WORDS
16		
20	0	1 IF LIBRARY CREATED WITH /X OPTION
22	0	RESERVED
24	0	RESERVED
26	10	DIRECTORY RELATIVE START ADDRESS
30		NUMBER OF BYTES IN DIRECTORY
32	0	RESERVED
34		NEXT INSERT RELATIVE BLOCK NUMBER
36		NEXT BYTE WITHIN BLOCK
40		DIRECTORY STARTS HERE

8.3.2 Library Directories

There are two kinds of library directories: for object libraries, the directory is an Entry Point Table (EPT); for macro libraries, the directory is a Macro Name Table (MNT). The EPT directory (see Figure 8–37) consists of four-word entries that contain information related to all modules in the library file.

Note that if you use the librarian /N option for object libraries to include module names, bit 15 of the relative block number word is set to 1. If you invoke the librarian with the monitor LIBRARY command, module names are never included.

In the library directory, the symbol characters represent the entry point or the macro name. The relative byte maximum is 777 octal.

The object library directory starts on the first word after the library header, word 40 octal. The object library directory is only long enough to accommodate the exact number of modules in the library and space for this directory is not pre-allocated. The directory is kept in memory during librarian operations, and the amount of available memory is the only limiting factor on the maximum size of the directory. Reserved locations in the header and at

Figure 8–36: Macro Library Header Format

OFFSET	CONTENTS	DESCRIPTION
0	1001	LIBRARY TYPE AND ID CODE
2	500	LIBRARY VERSION NUMBER
4	0	RESERVED
6		DATE IN RT-11 FORMAT (0 IF NONE)
10		TIME EXPRESSED IN TWO WORDS
12		
14	0	RESERVED
16	0	RESERVED
20	0	RESERVED
22	0	RESERVED
24	0	RESERVED
26	0	RESERVED
30	0	RESERVED
32	10	SIZE OF DIRECTORY ENTRIES
34		DIRECTORY STARTING RELATIVE BLOCK NUMBER
36		NUMBER OF DIRECTORY ENTRIES ALLOCATED (DEFAULT IS 200)
40		NUMBER OF DIRECTORY ENTRIES AVAILABLE

Figure 8–37: Library Directory Format (OBJ)

<table>
<tr><td colspan="3">SYMBOL CHARACTERS 1-3 (RADIX-50)</td></tr>
<tr><td colspan="3">SYMBOL CHARACTERS 4-6 (RADIX-50)</td></tr>
<tr><td></td><td colspan="2">BLOCK NUMBER RELATIVE TO START OF FILE</td></tr>
<tr><td colspan="2">RESERVED
(7 BITS)</td><td>RELATIVE BYTE IN BLOCK
(9 BITS)</td></tr>
</table>

the end of the directory – those not used by the directory – are zero-filled. Modules follow the directory and they are stored beginning in the next block after the directory.

The macro library directory starts on a block boundary, relative block 1 of the library file. Its size is pre-allocated. The default size is two blocks but you can change this by using the librarian /M option. Unused entries in the directory are filled with –1. Macro files are stored starting on the block boundary after the directory. This is relative block 3 of the library file if you use the default directory size.

Modules in libraries are concatenated by byte. (See Figure 8–2 for an example of byte concatenation.) This means that a module can start on an odd address. When this occurs, the linker shifts the module to an even address at link time.

8.3.3 Library End Block Format

Following all modules in an object or macro library is a specially coded Library End Block, or trailer, which signifies the end of the file (see Figure 8–38).

Figure 8–38: Library End Block Format

1		DATA BLOCK HEADER
10		DATA BLOCK LENGTH
10		LIBRARY END BLOCK CODE
0		RESERVED, MUST BE 0
	357	CHECKSUM BYTE

8.4 Absolute Binary File Format (LDA)

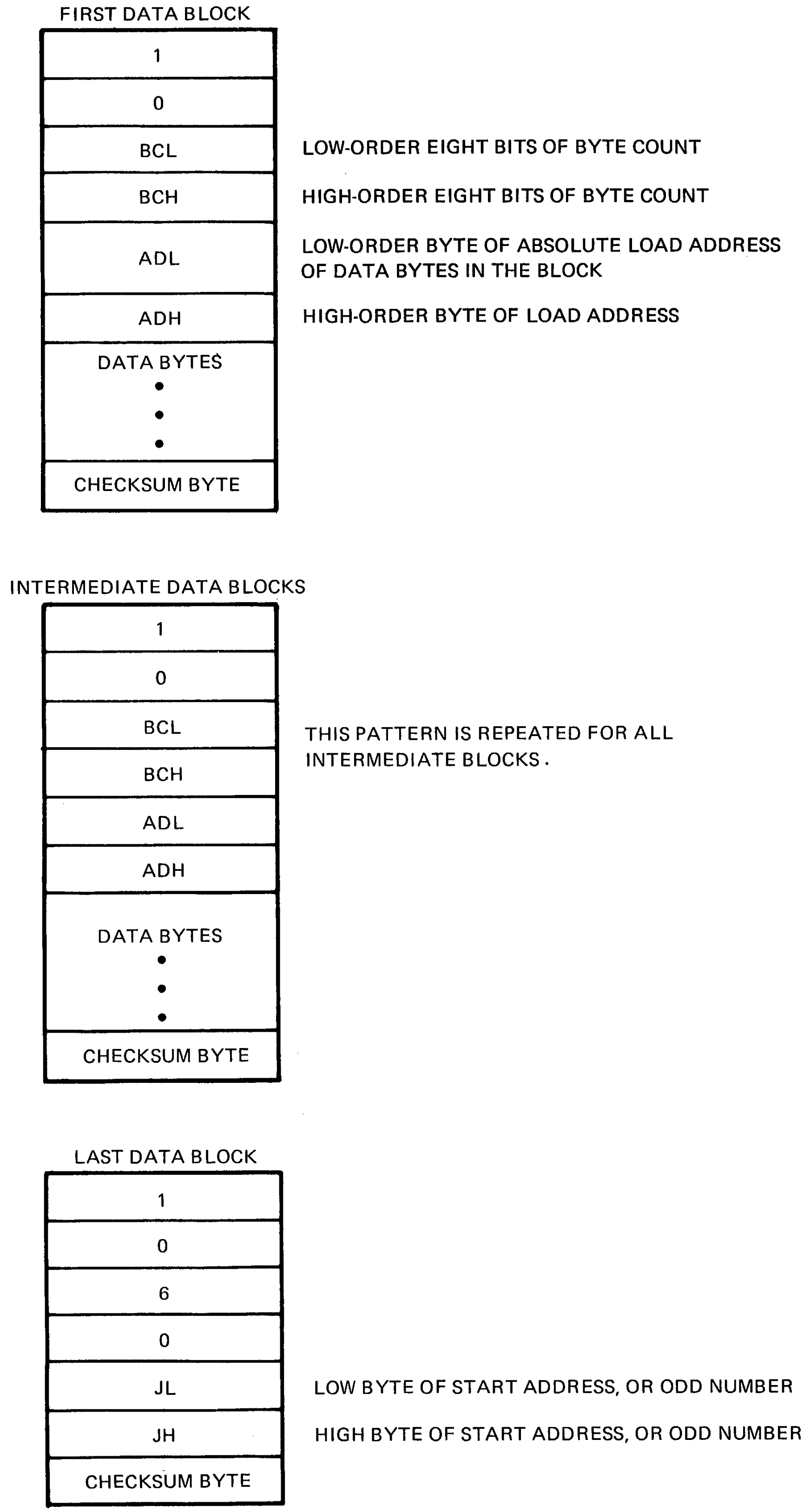
Both the linker /L option and the keyboard monitor LINK command /LDA option produce output files in a paper tape-compatible binary format.

Absolute binary format, shown in Figure 8–39, consists of a sequence of data blocks, where each block represents the data to be loaded into a specific portion of memory. The data portion of each block consists of the absolute load address of the block, followed by the absolute data bytes to be loaded into memory beginning at the load address. There can be as many data blocks as necessary in an LDA file. The last block of the file is special because it contains only the program start address, or transfer address, in its data portion. If this address is even, the Absolute Loader passes control to the loaded program at this address. If it is odd (that is, if the program has no transfer address, or the transfer address was specified as a byte boundary), the loader halts upon completion of loading. The final block of the LDA file is recognized by the fact that its length is six bytes.

You can use LDA format files for down-line loading of programs, for loading stand-alone application programs, and as input to special programs that put code into ROM (Read-Only Memory). The general procedure for loading a program that will execute in a stand-alone environment is as follows:

1. Toggle the Bootstrap Loader into memory.
2. Using the Bootstrap Loader, load the Absolute Loader into memory.

Figure 8–39: Absolute Binary Format (LDA)



3. Using the Absolute Loader, load the LDA file into memory and begin execution.

LSI–11 and LSI–11/2 computer systems have console microcode that makes steps 1 and 2 of this procedure unnecessary. The procedure for loading stand-alone programs is described in the *Microcomputer Processor Handbook*. LSI–11/23 computer systems do not have bootstrap loader microcode and require the use of steps 1 and 2.

You can obtain a listing of the *PDP–11 Bootstrap Loader* from DIGITAL's Software Distribution Center. Its order number is DEC–11-L1PA-LA (November 11, 1970). The Bootstrap Loader is also printed on the PDP–11 Programming Card, which is part of every RT–11 distribution kit. You can obtain a listing of the *PDP–11 Absolute Loader* by ordering product number DEC–11-UABLA-A-LA (June 1975). A paper tape of the Absolute Loader is also available. It is called the *Non-switch Register PDP–11 Absolute Loader VB07.00*, order number DEC–11-UABLB-A-PO (1975).

Complete procedures for using the Bootstrap Loader and the Absolute Loader are provided in the *PDP–11 Paper Tape Software Handbook*, order number DEC–11-XPTSA-B-D (April, 1976). Appendix F contains listings of the Bootstrap and Absolute Loaders.

The load module's data blocks contain only absolute binary load data and absolute load addresses. All global references have been resolved and the linker has performed the appropriate relocation.

8.5 Save Image File Format (SAV)

Save image format is for programs that are to be run in the SJ environment, or in the background in the FB and XM environments. It is also for virtual jobs that are to be FRUN or SRUN in XM environments. Save image files normally have a .SAV file type. This format is an image of the program exactly as it would appear in memory. (Block 0 — the first 256-word unit — of the file corresponds to memory locations 0–776, block 1 to locations 1000–1776, and so on.) See Table 8–8 for the contents of block 0 of a .SAV file. (Note that not all locations are used for each link; for example, whether the job is for an XM environment or whether it is overlaid affect block 0.) See also Chapter 11 of the *RT–11 System User's Guide* for more information on the load modules created by the linker.

Locations 360–377 in block 0 of the file are restricted for use by the system. The linker stores the program memory usage bits in these eight words, which are called a bitmap. Each bit represents one 256-word block of memory and is set if the program occupies any part of that block of memory. Bit 7 of byte 360 corresponds to locations 0 through 777; bit 6 of byte 360 corresponds to locations 1000 through 1777, and so on. The monitor uses this information when it loads the program.

Table 8–8: Information in Block 0

Offset	Contents
0	VIR in Radix–50 if the linker /V option was used
2	Virtual high limit if linker /V option was used
4	Reserved
6	Reserved
10	Reserved
12	Reserved
14	BPT trap (XM only)
16	BPT trap (XM only)
20	IOT trap (XM only)
22	IOT trap (XM only)
24	Reserved
26	Reserved
30	Reserved
32	Reserved
34	Trap vector (TRAP)
36	Trap vector (TRAP)
40	Program's relative start address
42	Initial location of stack pointer (changed by /M option)
44	Job Status Word
46	USR swap address
50	Program's high limit
52	Size of program's root segment, in bytes (used for REL files only)
54	Stack size, in bytes (changed by /R:n option) (used for REL files only)
56	Size of overlay region, in bytes (0 if not overlaid) (used for REL files only)
60	REL file ID (REL in Radix–50) (used for REL files only)
62	Relative block number for start of relocation information (used for REL files only)
64	Address of overlay handler table for overlaid files
66	Address of start of window definition blocks (if /V used)
70–356	Reserved
360–377	Bitmap area

The monitor commands R and RUN load and start a program stored in a SAV file. (The RUN command is actually a combination of the GET and START commands.) First, the Keyboard Monitor reads block 0 of the SAV file into an internal USR buffer. It extracts information from locations 40–64 and 360–377 (the bitmap, described above). Using the protection bitmap (called LOWMAP), which resides in RMON, KMON checks each word in block 0 of the file. It does not load locations that are protected, such as location 54 and the device interrupt vectors. It loads unprotected locations into memory from the USR buffer. Next, KMON sets location 50 to the top of usable memory, or to the top of the user program, whichever is greater.

If the RUN command (or the GET command) was issued, KMON checks the bitmap from locations 360–377 of the SAV file. For each bit that is set, it loads the corresponding block of the SAV file into memory. However, if KMON is in memory space that the program needs to use, KMON puts the block of the SAV file into a USR buffer and then moves it to the file SWAP.SYS.

Finally, when it is time to begin execution of the program, KMON transfers control to RMON. RMON reads the parts of the program, if any, that are stored in SWAP.SYS into memory, where they overlay KMON and possibly the USR. If the R command was issued, KMON does not check the bitmap to see which blocks of the SAV file to load. Instead, it jumps to RMON and attempts to read all locations above 1000 up to the top of the root segment into memory. (The R command does not use SWAP.SYS.) The monitor keeps track of the fact that KMON and the USR are swapped out, and execution of the program begins.

8.6 Relocatable File Format (REL)

To link a foreground job, use the linker /R option or the keyboard monitor LINK command with the /FOREGROUND option. This causes the linker to produce output in a linked, relocatable format, with a .REL file type. Note that system jobs are also stored in relocatable format. The only difference is that system jobs use a file type of .SYS instead of .REL.

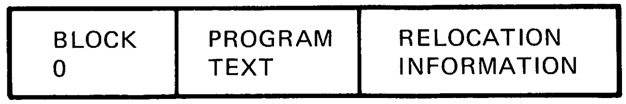
The object modules used to create a REL file are linked as if they were a background SAV image, with a base of 1000. This permits you to use .ASECT directives to store information in locations 0 through 777 in REL files. All global references have been resolved. The linker does not relocate the REL file at link time; it merely includes relocation information to be used at FRUN time. The relocation information in the file is used to determine which words in the program must be relocated when the job is installed in memory.

There are two types of REL files to consider: those programs with overlay segments, and those without them.

8.6.1 REL Files Without Overlays

A REL file for a program without overlays appears as shown in Figure 8–40.

Figure 8–40: REL File Without Overlays



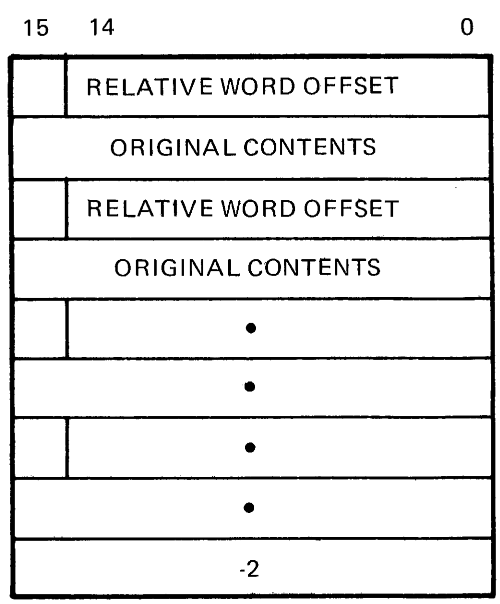
Block 0 (relative to the start of the file) contains the information shown in Table 8–8. Some of this information is used by the FRUN processor.

In the case of a program without overlays, the FRUN processor performs the following general steps to install a foreground job.

1. It reads block 0 of the file into an internal monitor buffer.
2. It obtains the amount of memory required for the job from location 52 of block 0 of the file, and allocates the space in memory by moving KMON and the USR down.
3. It reads the program text into the allocated space.
4. It reads the relocation information into an internal buffer.
5. It relocates the locations indicated in the relocation information area by adding or subtracting the relocation quantity. This quantity is the starting address the job occupies in memory, adjusted by the relocation base of the file. REL files are linked with a base of 1000.

The relocation information consists of a list of addresses relative to the start of the user's program. The monitor scans the list, and for each relative address computes an actual address. That address is then loaded with its original contents plus or minus the relocation constant. The relocation information is shown in Figure 8–41.

Figure 8–41: Root Relocation Information Format



In Figure 8–41 bits 0 through 14 represent the relative address to relocate divided by 2. This implies that relocation is always done on a word boundary, which is indeed the case. Bit 15 indicates the type of relocation to perform – positive or negative. The relocation constant (which is the load address of the program) is added to or subtracted from the indicated location depending on the sense of bit 15; 0 implies addition, while 1 implies subtraction. A full 16-bit word is the original contents. The value 177776, or –2, terminates the list of relocation information for a file without overlays.

8.6.2 REL Files with Overlays

When you include overlays in a program, in addition to relocating the root segment, the FRUN processor must also relocate the overlay segments. Since overlays are not permanently memory resident but are read in from the file as needed, they require an additional operation. FRUN relocates each overlay segment and rewrites it into the file before the program begins execution. Thus, when the overlay is called into memory during program execution, it is correct. This process takes place each time you run an overlaid file with FRUN or SRUN. The relocation information for overlaid files contains both the list of addresses to be modified and the original contents of each location. This allows the file to be executed again after the first usage. It is necessary to preserve the original contents in case some change has occurred in the operating environment. Examples of these changes include using a different monitor version, running on a system with a different amount of memory, and having a different set of device handlers or system jobs resident in memory. Figure 8–42 shows a REL file with overlays.

In the case of a REL file with overlays, location 56 of block 0 of the REL file contains the size in bytes of all the overlay regions. FRUN adds this size to the size of the program base segment (in location 52) to allocate space for the job.

After FRUN relocates the program base (root) code, it reads each existing overlay into the program overlay region in memory, relocates it using the overlay relocation information, and then writes it back into the file.

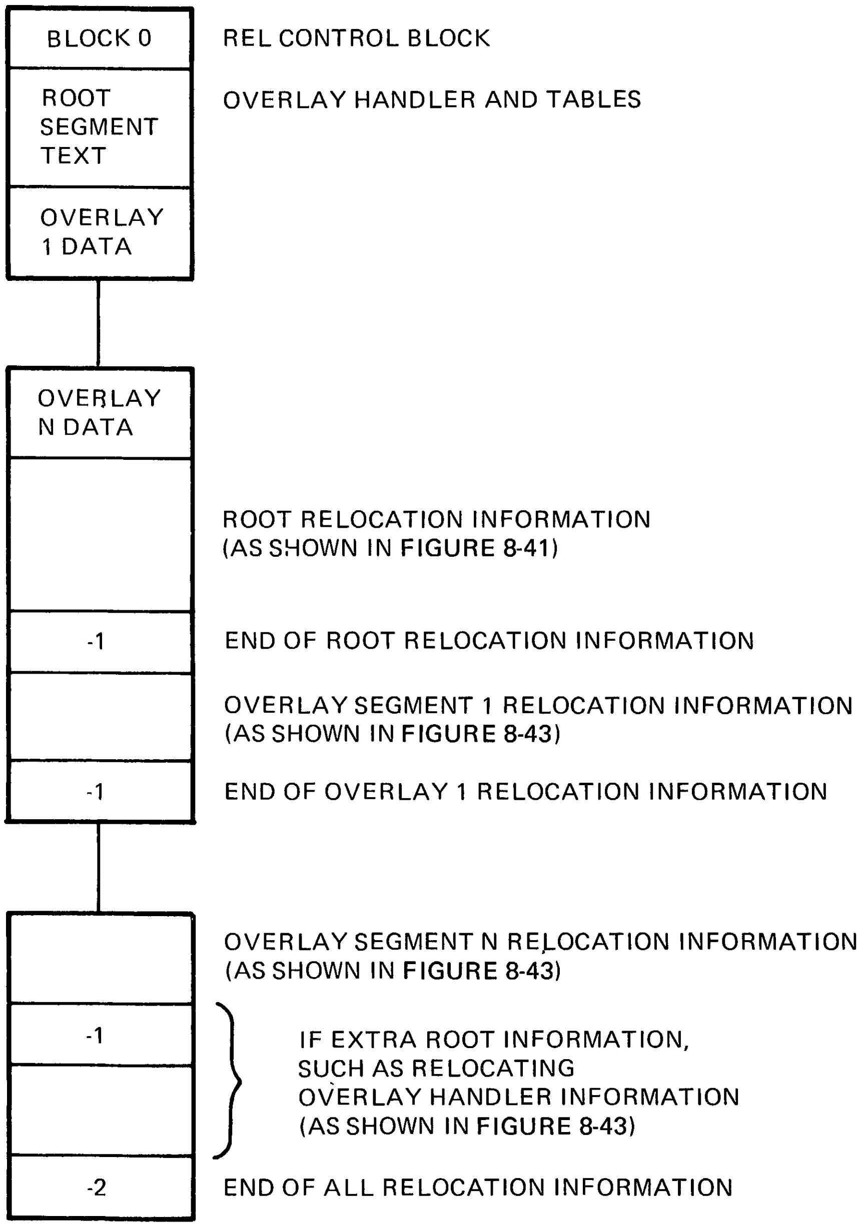
The root relocation information section is terminated with a –1. This –1 is also an indication that an overlay segment relocation block follows.

The relocation is relative to the start of the program and is interpreted as if it were in a file without overlays (that is, bit 15 indicates the type of relocation, and the displacement is the true displacement divided by 2). Encountering –1 indicates that a new overlay region begins here; a –2 indicates the termination of all relocation information.

8.7 Stream ASCII File Format

Source files, such as MACRO–11 and FORTRAN IV programs, and text files that you create with an editor are in stream ASCII format. These files consist of a series of bytes, each byte representing an ASCII character. Stream

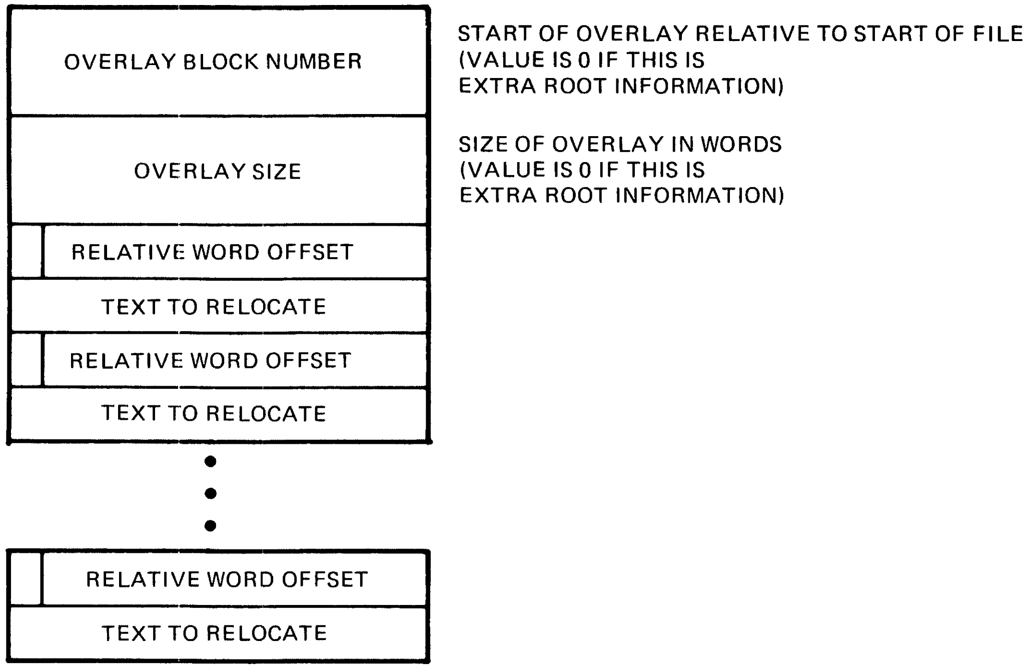
Figure 8–42: REL File with Overlays



ASCII files have no special headers or end blocks, nor do they include any formatted binary blocks.

An ASCII 32, or CTRL/Z character, may terminate a stream ASCII file. When you invoke PIP with the /A option (or when you use the monitor COPY/ASCII command) to copy an ASCII file, PIP expects to find a CTRL/Z at the end of the file. If there is an embedded CTRL/Z character within the file, PIP considers the CTRL/Z to mark the file's end. When you invoke PIP in its default mode (or when you use the monitor COPY command without any option) to copy an ASCII file, PIP does not look for a CTRL/Z character. It simply continues copying until it reaches the end of the file.

Figure 8–43: Overlay Segment Relocation Block



8.8 CREF File Format

The RT–11 CREF program produces a cross-reference listing. You can only run CREF indirectly – that is, by chaining to it from another program, such as your own language processor. CREF appends its cross-reference table to the listing file your calling program creates.

To chain to CREF, you must first store some information in the chain communication area (absolute locations 500 through 776) of the calling program. Table 8–9 lists the information that CREF requires.

Table 8–9: CREF Chain Interface Specification

Location	Contents	Description
500	.RAD50 /SY /	The file specification to invoke CREF:
502	.RAD50 /CRE/	
504	.RAD50 /F /	
506	.RAD50 /SAV/	
510		RT–11 channel number of output file
512		Radix–50 name of output device
514		Highest output block number written, plus 1

(Continued on next page)

Table 8–9: CREF Chain Interface Specification (Cont.)

Location	Contents	Description
516		RT–11 channel number of input file
520		Radix–50 name of input device
522		Highest input block number written, plus 1
524		Listing format: 0 = 80 columns, –1 = 132
526	.RAD50 /dev/	Program to chain back to. (If this value is zero, CREF closes the listing file and exits.)
530	.RAD50 /fil/	
532	.RAD50 /nam/	
534	.RAD50 /typ/	
536–776		ASCIZ string for CREF to use as title line (no page number)

The input file you supply to CREF must consist of 12-byte decimal entries, one entry for each reference to a symbol. Table 8–10 shows the format of the entries.

Table 8–10: Entry Format for CREF Input File

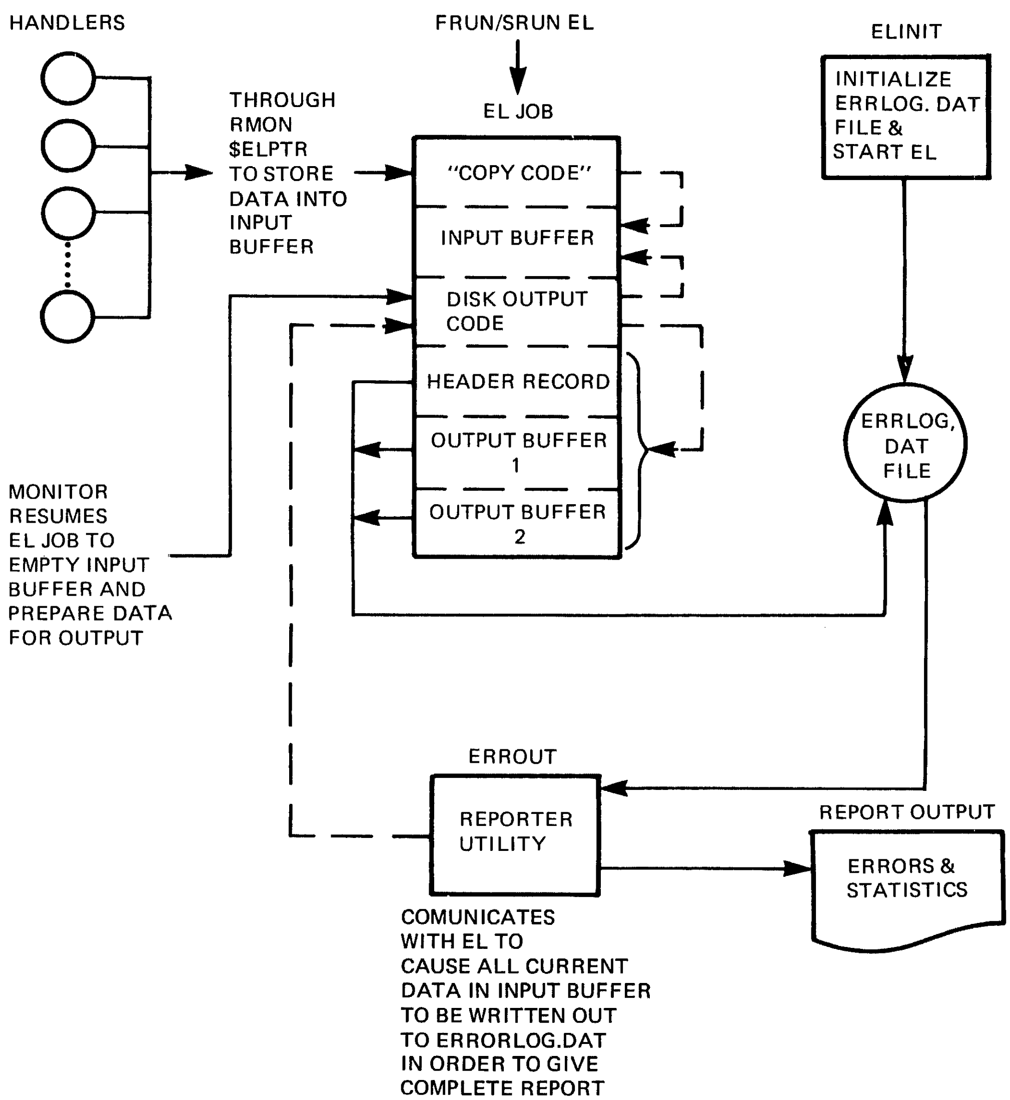
Octal Byte Offset	Value
0	Section descriptor: Bits 0 through 4 contain an alphabetic character for CREF to use as the section name. The ASCII value is stripped to 5 bits. Bits 5 through 7 contain the section number. This number controls the order of the sections.
1–6	The ASCII name of the symbol.
7–10	The page number, in binary. Put –1 here if you are not using page numbers.
11–12	The line number, in binary.
13	A one-character identifier for CREF to print next to this reference. Typically, this character is used to identify a destructive reference or a definitional reference.

8.9 Error Log File Formats

Device handlers that support error logging call the error logger through a monitor pointer on each successful I/O transfer as well as on each error. The **copy code** in the error logger retrieves, or copies, the appropriate information from the handler, storing it in the error log input buffer in the error logger's memory area. The error log job is suspended until the copy routine puts some data into the input buffer, at which point the monitor resumes the error log job so it can process the new data.

The error log job remains suspended until the error log input buffer has filled to 200 or more words of the 256-word decimal total buffer size. The copy code portion of the EL job informs the monitor of this by setting the carry bit on return. Thus the error log job is resumed by the monitor only when the error log input buffer contains a sizeable amount of information to be processed.

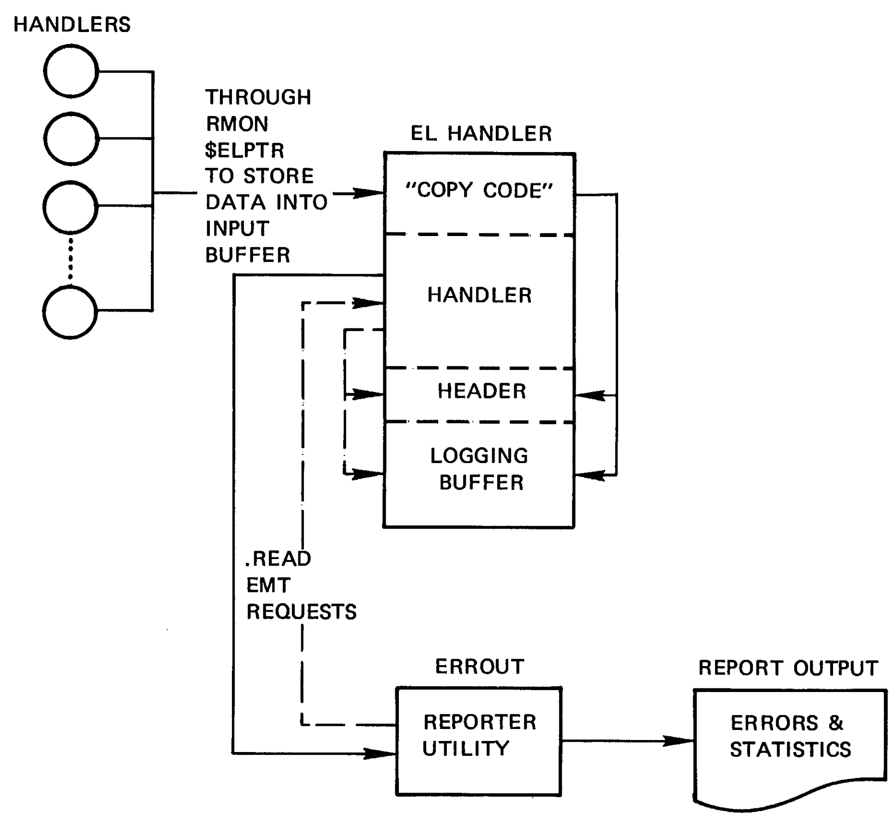
Figure 8–44: Error Logging Subsystem



For device errors, cache errors, and memory parity errors, the error logger first creates or updates the unit statistics information in the copy of the disk file header that is in memory. The EL job (disk output code) stores error records in the disk output buffer (one of two buffers, since double-buffering is used) until a 256-word decimal block is full. That is, it stores records until the buffer cannot contain the next record. Then it writes the updated header record and the accumulated error records to a disk file called ERRLOG.DAT. Figure 8–44 describes the error logging subsystem. Figure 8–45 illustrates the error logging internals for the SJ monitor.

For successful I/O transfers, the error logger first creates or updates the unit statistics information in the copy of the disk file header that is in memory, as it does with device and memory errors. It writes the updated header to disk only after 10 (decimal) good I/O transfers have been logged.

Figure 8–45: Error Logging Internals: SJ Monitor



8.9.1 Error Log Disk File Format

The error log disk file is called ERRLOG.DAT. Figure 8–46 shows its format.

The value of **parity ID** in Figure 8–46 is as follows:

–2 for a memory error

–3 for a cache error

–4 for both memory and cache error

Figure 8–46: ERRLOG.DAT Format

BLOCK 0 OF ERRLOG.DAT: (DATA STARTS IN BLOCK 1)	
POINTER TO THE HEADER SUMMARY SECTION	
DEVICE AND UNIT INFORMATION – (THE LENGTH OF THIS SECTION VARIES DEPENDING ON ERL$U) (A SEVEN-WORD ENTRY FOR EACH DEVICE UNIT LOGGED):	
DEVICE ID	UNIT
NUMBER OF ERROR RECORDS LOGGED	
NUMBER OF ERRORS RECEIVED	
READ SUCESSES (TWO WORDS)	
WRITE SUCCESSES (TWO WORDS)	
HEADER SUMMARY SECTION	
TOTAL NUMBER OF ERRORS RECEIVED (INCLUDING OCCASIONS WHEN THE ERROR LOGGER WAS UNABLE TO RECORD THE ERROR INFORMATION)	
COUNT OF MISSED RECORDS (NOT RECORDED BECAUSE THE ERROR LOGGER INPUT BUFFERS WERE FULL)	
COUNT OF MISSED RECORDS (NOT RECORDED BECAUSE ERRLOG.DAT WAS FULL)	
COUNT OF MISSED RECORDS (NOT RECORDED BECAUSE START UP OR SHUT DOWN WAS IN PROGRESS)	
NUMBER OF MEMORY PARITY ERRORS	
NUMBER OF CACHE PARITY ERRORS	
NEXT PHYSICAL RECORD NUMBER	
BLOCK NUMBER FOR START OF NEXT RECORD	
OFFSET WITHIN THE BLOCK FOR THE NEXT RECORD	
MAXIMUM SIZE OF ERROR FILE, IN BLOCKS	
CONFIGURATION WORD 1 (MONITOR FIXED OFFSET 300)	
CONFIGURATION WORD 2 (MONITOR FIXED OFFSET 370)	

Figure 8–46: ERRLOG.DAT Format (Cont.)

DATE OF INITIALIZATION	
TIME OF INITIALIZATION (TWO WORDS)	
DATA BLOCKS (A SERIES OF 256-WORD BLOCKS CONTAINING ERROR INFORMATION):	
FOR EACH DEVICE ERROR:	
PHYSICAL RECORD NUMBER	SIZE OF THIS ERROR RECORD
DEVICE ID	UNIT
NUMBER OR OCCURRENCES OF THIS ERROR WITH IDENTICAL REGISTERS	RETRY COUNT
DATE OF ENTRY	
TIME OF ENTRY (TWO WORDS)	
PHYSICAL BLOCK NUMBER (Q.BLKN)	
USER BUFFER ADDRESS (Q.BUFF)	
WORD COUNT (Q.WCNT)	
PAR 1 VALUE (Q.PAR) – XM ONLY	
TOTAL NUMBER OF RETRIES	NUMBER OF REGISTERS TO LOG
THE DEVICE REGISTERS: • • •	
FOR EACH PARITY ERROR:	
PHYSICAL RECORD NUMBER	SIZE OF THIS ERROR RECORD
NUMBER OF MEMORY REGISTERS	PARITY ID
NUMBER OF OCCURRENCES OF THIS ERROR WITH THE SAME PC	
DATE OF ENTRY	
TIME OF ENTRY (TWO WORDS)	
PC	
PS	
MPR1	
ADDRESS OF MPR1	
(INFORMATION FOR UP TO 16 MEMORY REGISTERS:) • • •	
MEMORY SYSTEM ERROR REGISTER (IF CACHE IS PRESENT)	
CACHE CONTROL REGISTER (IF CACHE IS PRESENT)	
HIT/MISS REGISTER (IF CACHE IS PRESENT)	

Chapter 9
File Storage

RT–11 stores files under assigned file names on file-structured devices. RT–11 devices that are file-structured include all disks and diskettes, DECtapes, magtape, and cassette.

File-structured devices that have a series of directory segments at the beginning of the device are called directory-structured devices. The directory segments contain entries describing the names, lengths, and creation dates of files on the device. Disks, diskettes, and DECtapes are directory-structured devices. Because the directory is at the beginning of the device, you can access any file on the device, no matter where it is located, without reading any other files. For this reason, directory-structured devices are sometimes called random-access or block-replaceable devices.

Magtape and cassette are file-structured devices that do not have a directory at the beginning. While they do store some directory information at the beginning of each file, you must still read the entire volume to obtain all the information about all the files. Because you must read the files in order, one after the other, magtape and cassette are also called sequential-access devices.

This chapter shows how RT–11 stores files on both random-access and sequential-access devices. It also describes the contents of a device directory and shows how to recover information from a random-access device whose directory is corrupted.

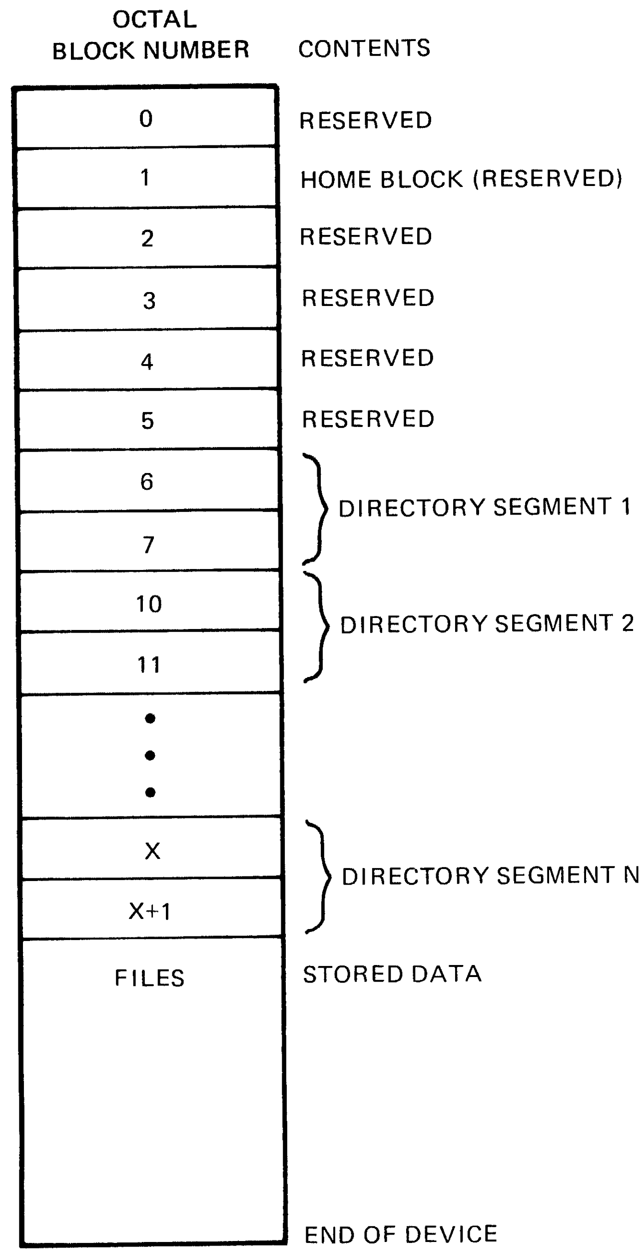
9.1 Random-Access Devices

A random-access device consists of a series of 256-word blocks where blocks 0 through 5 are reserved for system use and cannot be used for data storage. The device directory begins at block 6. Figure 9–1 shows the format of a random-access device.

9.1.1 Home Block

Block 1 of a random-access device, called the **home block**, contains information about the volume and its owner. Figure 9–2 and Table 9–1 show the home block format and contents.

Figure 9–1: Random-Access Device



To compute the checksum, all the bytes are added into a word, which is then negated.

The contents of all other areas in the home block are undefined and reserved for future use by DIGITAL.

Figure 9–2: Home Block Format

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21	22	23	24	25	26	27	30	31	32	33	34	35	36	37
000	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
040	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
100	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
140	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
200	a	a			b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
240	b	b	b	b	b	b	b	b	b	b	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c				
300																																
340																																
400																																
440																																
500																																
540																																
600																																
640																																
700	d	d	e	e															f	f	g	g	h	h	i	i	i	i	i	i	i	i
740	i	i	i	i	j	j	j	j	j	j	j	j	j	j	j	j	k	k	k	k	k	k	k	k	k	k	k	k			l	l

Table 9–1: Home Block Contents

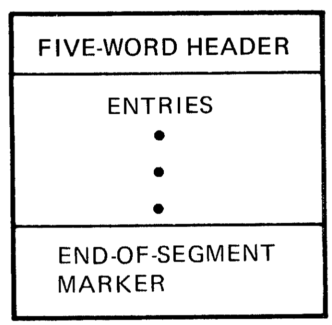
Field	Location	Contents	Default
a	000–201	Bad block replacement table	
b	204–251	INITIALIZE/RESTORE data area	
c	252–273	BUP information area	
d	700–701	(Reserved for DIGITAL)	000000
e	702–703	(Reserved for DIGITAL)	000000
f	722–723	Pack cluster size	000001
g	724–725	Block number of first directory segment	000006
h	726–727	System version	Radix–50 V3A
i	730–743	Volume Identification	RT11A and seven spaces
j	744–757	Owner name	12 spaces
k	760–773	System Identification	DECRT11A and four spaces
l	776–777	Checksum	

9.1.2 Directory Structure

The directory consists of a series of two-block segments. Each segment is 512 words long and contains information about files such as name, length, and creation date. A directory can have from 1 to 31 decimal segments. You establish the size of the directory area by determining at device initialization time the number of segments in the directory. Use the INITIALIZE/SEGMENTS:n command, or DUP with the /Z/N:n options. (See Chapter 4 of the *RT–11 System User's Guide* for more information on the INITIALIZE command and for a table of the default number of segments for all RT–11 devices. See the *RT–11 Installation Guide* for a patch that changes the default number of segments.) In general, you should select many segments if you need to store many small files on a large device. By selecting the minimal number of segments and reducing the size of the directory area, you obtain more space to store large files on a smaller device.

Each directory segment consists of a five-word header plus a number of entries containing file information. Each segment ends with an end-of-segment marker. Figure 9–3 shows the general format of the device directory.

Figure 9–3: Device Directory Format



9.1.2.1 Directory Header Format — Each directory segment contains a five-word header, which leaves the remaining 507 words of the two-block segment for directory entries. Table 9–2 describes the contents of the header words.

9.1.2.2 Directory Entry Format — The remainder of the directory segment consists of a number of directory entries followed by an end-of-segment marker. Figure 9–4 shows the format of a directory entry.

The first word of each directory entry is the status word, which describes the condition of the actual files stored on the device. The high-order byte of the status word contains a code representing the type of file. The low-order byte is reserved and should always be 0. Figure 9–5 illustrates the status word.

Table 9–2: Directory Header Words

Word	Contents
1	The total number of segments in this directory. The valid range is from 1 through 31 decimal. If you do not specify the number of segments you require when you initialize the device, DUP uses the default number of segments for that device.
2	The segment number of the next logical directory segment. The directory is a linked list of segments and this word is the link between the current segment and the next logical segment. If this word is 0, there are no more segments in the list.
3	The number of the highest segment currently in use. RT–11 increments this counter each time it opens a new segment. Note that the system maintains this counter only in word 3 of the header for the first directory segment. It completely ignores the third word of the header of the other segments.
4	The number of extra bytes per directory entry, always an unsigned, even octal number. See Section 9.1.2.2 for more information.
5	The block number on the device where the actual stored data monitored by this segment begins.

Figure 9–4: Directory Entry Format

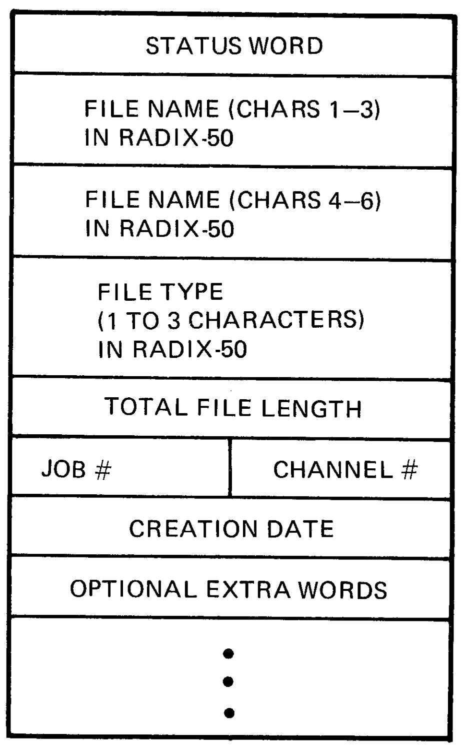


Figure 9–5: Status Word Format

TYPE OF FILE	RESERVED

RT–11 uses three kinds of directory entries:

- tentative entries
- empty entries
- permanent entries

You can think of the three types of entry as describing areas that are categorized as temporary data, available space on the device, or permanent data. The device directory contains at all times sufficient entries to describe the entire device.

A **tentative file** is a file that is in the process of being created. When a program issues the .ENTER programmed request, for example, it creates a tentative file. The program must issue a .CLOSE programmed request to make the tentative file permanent. If you do not eventually close a tentative file, the system deletes it. The DIR utility program lists tentative files that appear in directories as <UNUSED> files.

An **empty entry** defines an area of the device that is available for use. Thus, when you delete a file, you obtain an empty area. DIR lists an empty area as <UNUSED>, followed by its length.

A **permanent file** is a tentative file that has been closed with the .CLOSE programmed request. Permanent files are unique — that is, only one file can exist with a specific name and file type on a device. If another file exists with the same name and type when the program closes the current tentative file, the monitor deletes the first file as part of the .CLOSE routine, thus replacing the old file with the new file. DIR lists permanent files that appear in directories by their file names, file types, sizes, and creation dates.

Table 9–3 lists the five valid status word values and their meanings.

The second, third, and fourth words in a directory entry contain the Radix–50 representation of the file name and file type. For empty area, RT–11 normally ignores these words. However, the DIR /Q option (or the monitor DIRECTORY command with the /DELETED option) lists the names and file types of deleted files.

The fifth word in a directory entry contains the total file length, which consists of the number of blocks the file occupies on the device. Attempts to read or write outside the limits of the file result in an end-of-file error.

The sixth word in a directory entry contains the channel number and sometimes the job number as well. RT–11 uses this information only for tentative files. A tentative file is associated with a job in one of two ways, depending on which RT–11 monitor is running.

Table 9–3: Status Word Values

Status Word	Meaning
400	Tentative file.
1000	Empty area. RT–11 does not use the name, file type, or date fields in the directory entry for an empty area.
2000	Permanent file.
102000	Protected permanent file (see Section 9.1.2.3).
4000	End-of-segment marker. RT–11 uses this marker to determine when it has reached the end of the directory segment during a directory search. Note that an end-of-segment marker can appear as the last word of a segment. It does not have to be followed by a name, file type, or other entry information.

In the SJ environment, the low byte of the sixth word of the entry holds the channel number on which the file is open. This number enables the monitor to locate the correct tentative file for the channel when a program issues the .CLOSE programmed request.

In the FB and XM environments, as with SJ, the low byte of the sixth word of the directory entry contains the channel number. In addition, the high byte of the sixth word contains the number of the job that is opening the file. The job number is required to identify the correct tentative file during the .CLOSE operation. It is also necessary because several jobs can have files open using the same channel number.

NOTE

> RT–11 uses the sixth word (job number and channel number word) only when the file is tentative. Once the file becomes permanent, RT–11 no longer uses the word. The function of the sixth word while the file is permanent is reserved for future use by DIGITAL.

The seventh word of a directory entry contains the file's creation date. When a program creates a tentative file by issuing the .ENTER programmed request, the system moves the system date word into the creation date slot for the entry. The date word is 0 if you did not enter a date with the DATE monitor command. Figure 9–6 shows the format of the date word. Bits 14 and 15 are reserved for future use by DIGITAL.

Figure 9–6: Date Word Format

15 14	13 12 11 10	9 8 7 6 5	4 3 2 1 0
	MONTH, IN DECIMAL (1–12)	DAY, IN DECIMAL (1–31)	YEAR MINUS 110, IN OCTAL

Normally, directory entries are seven words long, but by using DUP with the /Z:n option, you can allocate extra words for each entry when you initialize the device. The fourth word of the directory header contains the number of extra bytes you specify. Although DUP lets you allocate extra words, RT–11 provides no means to manipulate this extra information conveniently. Any program that needs to access these words must perform its own operations on the RT–11 directory. In addition, programs that manipulate the directory should use bit test (BIT) instructions, rather than compare (CMP) instructions.

9.1.2.3 File Protection — RT–11 provides a mechanism to prevent a file from being deleted. A file is protected when the high bit of its status word is set. Note that only permanent files can be protected. You can protect and unprotect files by using the PIP /R option or the monitor RENAME command. For more information, see the *RT–11 System User's Guide*.

9.1.2.4 Sample Directory Segment — The directory listings shown in Figure 9–7 describe a single-density diskette with 11 files.

Figure 9–7: Directory Listings

```
DIRECTORY/FULL DXO:

 29-APR-82
SWAP  .SYS     24       19-FEB-82     RT11SJ.SYS     65 19-FEB-82
< UNUSED >     77                     PIP   .SAV     16 19-FEB-82
DUP   .SAV     21       19-FEB-82     DIR   .SAV     17 19-FEB-82
EDIT  .SAV     19       19-FEB-82     LINK  .SAV     37 19-FEB-82
LIBR  .SAV     20       19-FEB-82     DUMP  .SAV      7 19-FEB-82
MACRO .SAV     45       19-FEB-82     SIPP  .SAV     13 29-APR-82
< UNUSED >     119
 11 FILES, 284 BLOCKS
 196 FREE BLOCKS

DIRECTORY/SUMMARY DXO:

 29-APR-82

    11 FILES IN SEGMENT 1

     4 AVAILABLE SEGMENTS, 1 IN USE

 11 FILES, 284 FREE BLOCKS
 196 FREE BLOCKS
```

Figure 9–8 shows the contents of segment 1 of the diskette directory, obtained by dumping absolute block number 6 of the device.

To find the starting block of a particular file, first find the directory segment containing the entry for that file. Then take the starting block number in the fifth word of that directory segment and add to it the length of each permanent, tentative, and empty entry in the directory before your file. For example, in Figure 9–8 the permanent file RT11SJ.SYS begins at block number 46 octal on the device.

Figure 9–8: RT–11 Directory Segment

HEADER:	4	FOUR SEGMENTS AVAILABLE
	0	NO NEXT SEGMENT
	1	HIGHEST OPEN IS #1
	0	NO EXTRA BYTES PER ENTRY
	16	FILES START AT DEVICE BLOCK 16 OCTAL
ENTRIES:	2000	PERMANENT FILE
	75131	RADIX-50 FOR SWA
	62000	RADIX-50 FOR P
	75273	RADIX-50 FOR SYS
	30	FILE IS 30 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED ON 19-FEB-79
	2000	PERMANENT FILE
	71677	RADIX-50 FOR RT1
	142302	RADIX-50 FOR 1SJ
	75273	RADIX-50 FOR SYS
	101	101 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED ON 19-FEB-79
	1000	EMPTY AREA (THE FILE DXMNFB WAS DELETED)
	16315	RADIX-50 FOR DXM
	54162	RADIX-50 FOR NFB
	75273	RADIX-50 FOR SYS
	115	115 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	62570	RADIX-50 FOR PIP
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	20	20 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	16130	RADIX-50 FOR DUP
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	25	25 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	15172	RADIX-50 FOR DIR
	0	RADIX-50 FOR SPACES
	73376	RADIX-50 FOR SAV
	21	21 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79
	2000	PERMANENT FILE
	17751	RADIX-50 FOR EDI
	76400	RADIX-50 FOR T
	73376	RADIX-50 FOR SAV
	23	23 OCTAL BLOCKS LONG
	–	USED ONLY FOR TENTATIVE FILES
	5147	CREATED 19-FEB-79

Figure 9–8: RT–11 Directory Segment (Cont.)

2000	PERMANENT FILE
46166	RADIX-50 FOR LIN
42300	RADIX-50 FOR K
73376	RADIX-50 FOR SAV
45	45 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
46152	RADIX-50 FOR LIB
70200	RADIX-50 FOR R
73376	RADIX-50 FOR SAV
24	24 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
16125	RADIX-50 FOR DUM
62000	RADIX-50 FOR P
73376	RADIX-50 FOR SAV
7	7 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
50553	RADIX-50 FOR MAC
71330	RADIX-50 FOR RD
73376	RADIX-50 FOR SAV
55	55 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
5147	CREATED 19-FEB-79
2000	PERMANENT FILE
74070	RADIX-50 FOR SIP
62000	RADIX-50 FOR P
73376	RADIX-50 FOR SAV
15	15 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
11647	CREATED 29-APR-79
1000	EMPTY AREA (NEVER USED SINCE INITIALIZATION)
000325	RADIX-50 FOR EM (STORED AT INITIALIZATION)
063471	RADIX-50 FOR PTY (STORED AT INITIALIZATION)
023364	RADIX-50 FOR FIL (STORED AT INITIALIZATION)
167	167 OCTAL BLOCKS LONG
–	USED ONLY FOR TENTATIVE FILES
–	(THE DATE IS NOT SIGNIFICANT)
4000	END-OF-SEGMENT-MARKER

9.1.3 File Storage on Random-Access Devices

RT–11 uses the three types of directory entry mentioned previously to completely describe the contents of a random-access device. All files reside on blocks that are contiguous on a device. There are several advantages and disadvantages to this method of storing data.

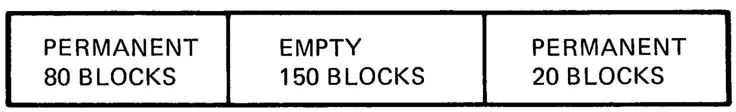
When data is stored in contiguous blocks, I/O is more efficient. Transfers to large buffers are handled directly by the hardware for certain disks; seeks between blocks and program interrupts between blocks are eliminated. File data is processed simply and efficiently since the data is not encumbered by link words in each block. Routines to maintain the directory are relatively small because the directory structure is simple. File operations, such as open, delete, and close, are performed quickly with few disk accesses, because only the directory must be accessed, and not additional bitmaps or retrieval pointers.

One disadvantage of this method of storing data is that a small device can become fragmented, requiring a squeeze operation to consolidate its free space. Another is that once a file is closed, a running program cannot easily increase its size. Only a small number of output files can be opened simultaneously, even on a large device, unless the limits of the file sizes are known in advance. Finally, this scheme precludes the use of multiple and hierarchical directories.

In summary, any method of storing data has its advantages and its disadvantages. The contiguous block method is used in RT–11 because its simple structure and low overhead best suit typical RT–11 applications.

Figure 9–9 shows a simplified diagram of a random-access device that has a total of 250 blocks of space available for files after blocks 0 through 5 and the directory are accounted for. The device in the figure has two permanent files and one empty area stored on it.

Figure 9–9: Random-Access Device with Two Permanent Files



When you create a file, your program must allocate the space for the file in the .ENTER programmed request. If you do not know the actual size, as is often the case, the space you allocate should be large enough to accommodate all the data possible. Two special cases for the .ENTER programmed request permit you to do this easily. In the first case, a length argument of 0 allocates for the file either one-half the largest space available, or the second largest space, whichever is bigger; in the second case, a length argument of –1 allocates the largest space possible on the device.

The monitor creates a tentative file on the device with the length you specified. The tentative file must always be followed by an empty area to enable the system to recover unused space if less data is written to the file than you originally estimated. Figure 9–10 shows an example of a tentative file whose allocated size is 100 blocks. Note that the total amount of space on the device, 250 blocks in this case, remains constant.

Figure 9–10: Random-Access Device with One Tentative File

PERMANENT 80 BLOCKS	TENTATIVE 100 BLOCKS	EMPTY 50 BLOCKS	PERMANENT 20 BLOCKS

Suppose, for example, that while the file is being created by one program, another program enters a new file, allocating 25 blocks for it. The device would appear as shown in Figure 9–11. Remember that every tentative file must be followed by an empty area.

Figure 9–11: Random-Access Device with Two Tentative Files

PERMANENT 80 BLOCKS	TENTATIVE 100 BLOCKS	EMPTY 0 BLOCKS	TENTATIVE 25 BLOCKS	EMPTY 25 BLOCKS	PERMANENT 20 BLOCKS

When a program finishes writing data to the device, it closes the tentative file with the .CLOSE programmed request. RT–11 then makes the tentative file permanent. The length of the file is the actual size of the data that was written. The size of the empty area is its original size plus any unused space from the tentative file.

Figure 9–12 shows the same device after both tentative files are closed. The first file's actual length is 75 blocks, and the second file's length is 10 blocks.

Figure 9–12: Random-Access Device with Four Permanent Files

PERMANENT 80 BLOCKS	PERMANENT 75 BLOCKS	EMPTY 25 BLOCKS	PERMANENT 10 BLOCKS	EMPTY 40 BLOCKS	PERMANENT 20 BLOCKS

Because of this method of storing files, it is impossible in RT–11 to extend the size of an existing file from within a running program. To make an existing file appear to be bigger, you can read the existing file; allocate a new, larger tentative file; and write both the old and the new data to the new file. You can then delete the old file.

The DUP utility program provides the /T option as an easy way to extend the size of an existing file. However, to use this option, you must have an empty file with sufficient space in it immediately following the data file. (You can also access this option through the monitor CREATE/EXTENSION command.)

9.1.4 Size and Number of Files

The number of files you can store on an RT–11 device depends on the number of segments in the device's directory and the number of extra words per entry. If you use no extra words, each segment can contain 72 entries.

The maximum number of directory segments on any RT–11 device is 31 decimal. Use the following formula to calculate the theoretical maximum number of directory entries, and thus, the maximum number of files.

$$31 * \frac{512 - 5}{7 + N} - 2$$

N represents the number of extra information words per directory entry. If **N** is 0, the maximum number of files you can store on the device is 2230 decimal.

Note that all divisions are integer and the remainder should be discarded.

In the formula shown above, the –2 is required for two reasons. First, in order to create a file, the tentative file must be followed by an empty area. Second, an end-of-segment entry must exist. Note that on a disk squeezed by DUP, the end-of-segment entry might not be a full entry, but may contain just the status word.

If you store files sequentially (that is, one immediately after another) without deleting any files, roughly one-half the theoretical maximum number of files will fit on the device before a directory overflow occurs. This situation results from the way RT–11 handles filled directory segments.

When a directory segment becomes full and it is necessary to open a new segment, the monitor moves approximately one-half of the directory entries of the filled segment to the new segment. Thus, when the final segment is full, all previous segments have approximately one-half of their total capacity. See Section 9.1.5 for a detailed explanation of how RT–11 splits a directory segment.

If you add files continually to a device without issuing the SQUEEZE monitor command, you can use the following formula to compute the maximum number of entries, and thus, the maximum number of files.

$$(M - 1) * \frac{S}{2} + S$$

M represents the maximum number of segments.

S can be computed from the following formula:

$$S = \frac{512 - 5}{7 + N} - 2$$

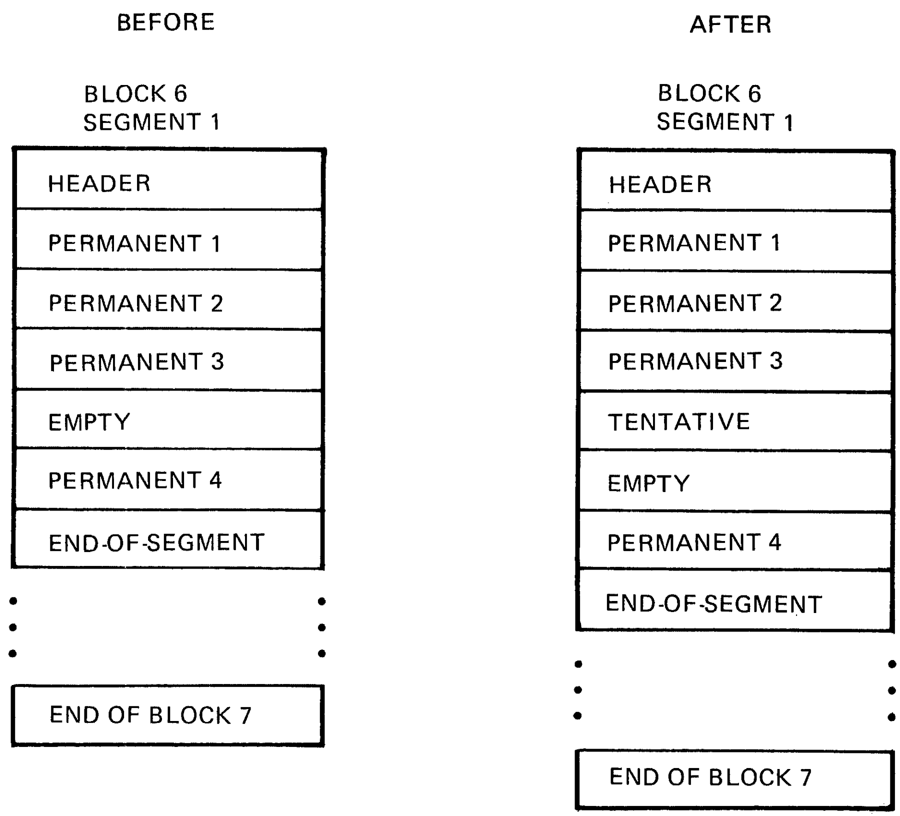
N represents the number of extra information words per entry.

You can realize the theoretical total of directory entries (see the first formula, above) by compressing the device (using the DUP /S option or the monitor SQUEEZE command) when the directory fills up. DUP packs the directory segments as well as the physical device.

9.1.5 Splitting a Directory Segment

Whenever RT–11 stores a new file on a volume, it searches through the directory for an empty area that is large enough to accommodate the new tentative file. When it finds a suitable empty area, it creates the new file as a tentative file followed by an empty area, sliding the rest of the directory entries down to make room for the new entry. Figure 9–13 shows how RT–11 stores a new file as a tentative file followed by an empty area.

Figure 9–13: Storing a New File

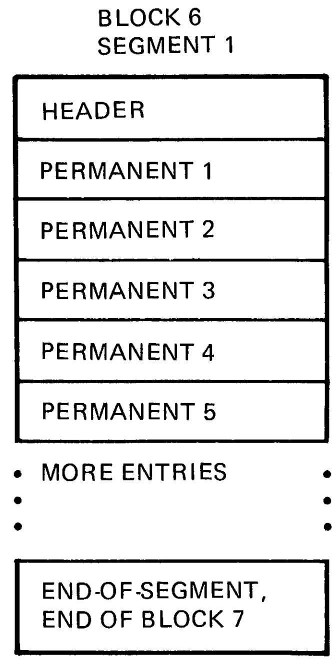


This procedure works properly as long as the empty entry and the entries following it can move downward. However, if the segment is full, the monitor must split the segment, if possible, in order to store the new entry. Figure 9–14 illustrates a directory segment that is full.

First, the monitor checks the header for the number of segments available. If there are none, a directory full error results and the monitor cannot store the new file. You can squeeze the volume at this point to pack the directory segments, and try the operation again.

If there is another directory segment available, the monitor divides the current segment by first finding a permanent or tentative entry near the middle of the segment and saving its first word. In place of the first word, the monitor puts an end-of-segment marker. It then saves the current link information, links the current segment to the next available segment, and writes the current segment back to the volume.

Figure 9–14: Full Directory Segment



Next, the monitor restores the first word of the middle entry to the copy of the segment that is still in memory, and restores the link information. It slides the middle entry and all the entries following it to the top of the segment. Then the monitor writes this segment to the volume as the next available segment. Finally, the monitor reads segment 1 into memory and updates the information in its header, at which point control passes to the top of the .ENTER routine, and the monitor begins its search again for a suitable empty entry to accommodate the new file.

Figures 9–15 and 9–16 summarize the process of splitting a directory segment. In this example, segment 1 was the only segment in use. It contained an empty entry but did not have room for a tentative entry in addition to the empty one. After the split, segments 1 and 2 are both about half full.

After a directory segment splits, the monitor can store the new file in either the new segment or the old one, depending on which segment now contains the empty area. In Figure 9–16, the empty area is in segment 2.

Thus far, the link words seem superfluous since the segments are always in numerical order. However, consider a situation in which four segments are available: segment 1 fills and overflows into segment 2; segment 2 fills and overflows into segment 3; segments 1, 2, and 3 are half full, and they are linked in the order in which they are located on the volume (blocks 6, 10, and 12). The picture changes if you delete a large file from segment 2, leaving a large empty entry, and add a lot of small files to the volume. Segment 2 now fills up and overflows into the next free segment, segment 4, so that the links become visibly significant: segment 1 links to 2, segment 2 links to 4, and segment 4 links to 3 because segment 2 previously linked to 3. Figure 9–17 illustrates this example.

Figure 9–15: Directory Before Splitting

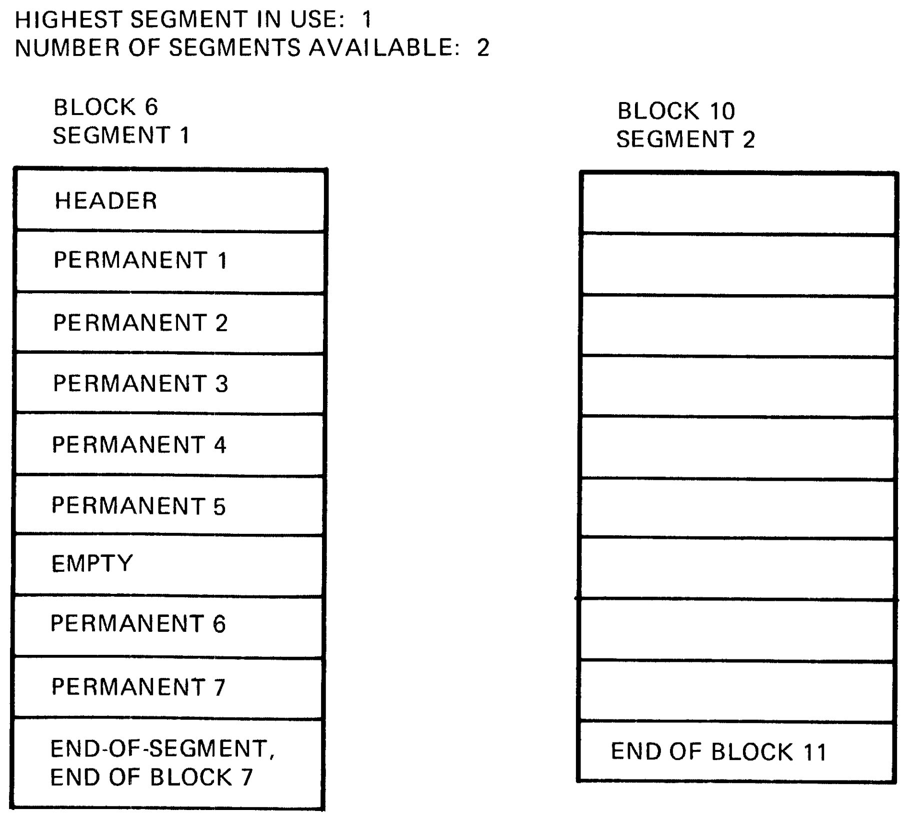


Figure 9–16 Directory After Splitting

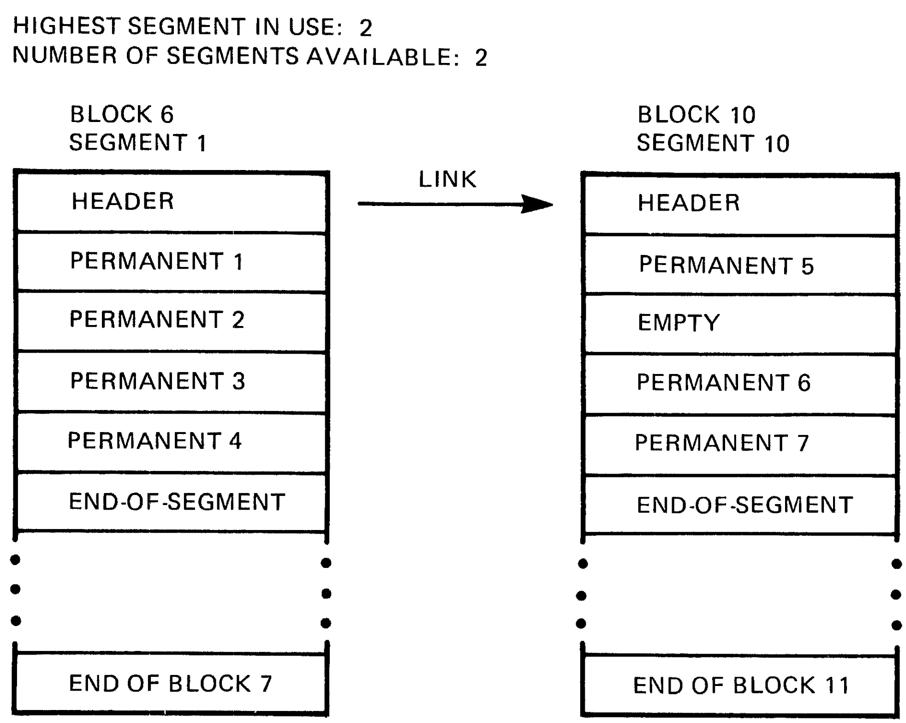
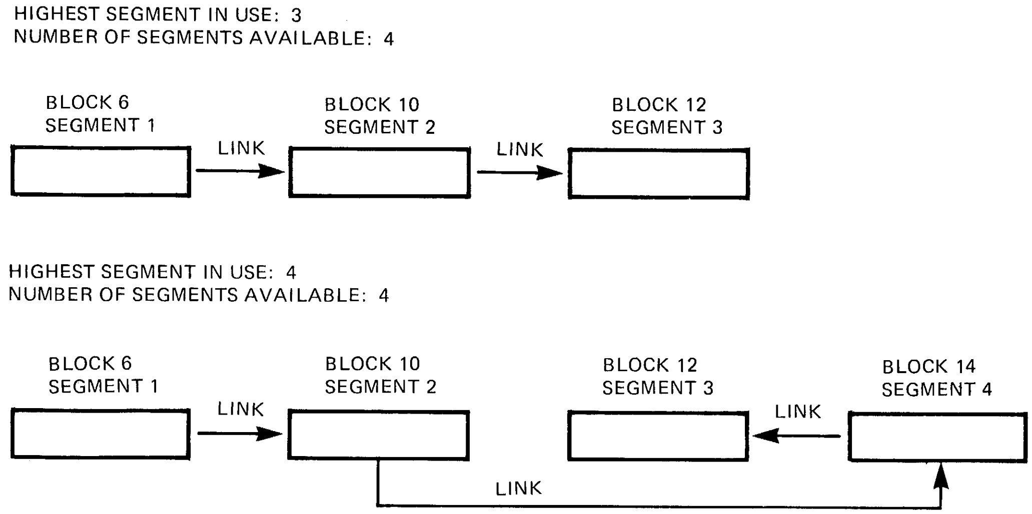


Figure 9–17: Directory Links



The process of splitting a directory segment in half when it is full may seem unnecessarily complicated. In fact, if many files are simply copied to a disk, a *directory full* error will occur when the directory is only half full. A squeeze will be required to consolidate the half-full directory segments before more files can be copied to the disk. However, splitting a directory segment reduces the amount of directory entry shuffling that occurs as files are added and deleted on a volume, thereby improving overall efficiency.

Refer back to Figure 9–15. Assume the empty entry between PERMANENT–5 and PERMANENT–6 represents 100 blocks of free disk space and that directory splitting is not done. If a program makes a directory entry for a new 25-block file, the directory entry for the file PERMANENT–7 must be moved to directory segment 2 to make room in segment 1 for the entry for PERMANENT–5A and an empty entry of 75 blocks. If the program makes another directory entry for another 25-block file, the file PERMANENT–6 must be moved to segment 2 to make room for the entry PERMANENT–5B and an empty entry of 50 blocks. In other words, each time a new directory entry occurs in segment 1, segment 2 must be updated as well, requiring an extra disk read and write.

If a directory segment is split when full, however, this problem does not occur as readily. Consider Figure 9–16 and observe what has happened. When a program creates the new 25-block file PERMANENT–5A, only directory segment 2 must be updated. No directory shuffling is required. If the file PERMANENT–4 is deleted and replaced with two or more files, directory segment 1 does not have room to accommodate the new file entries. Because directory splitting moves several directory entries from one segment to another at one time, any given directory operation is far less likely to require access to more than one directory segment. Directory splitting reduces dramatically the number of disk accesses required, on average, and improves overall directory efficiency.

9.1.6 How to Recover Data When the Directory Is Corrupted

One of the most frustrating experiences you can have as a programmer is to lose data on a volume because a block in the device directory went bad or because another user wrote over the directory. Usually, in a situation like this the files on the volume are intact, but the directory entries for some of the files have been destroyed. This section presents some guidelines you can follow to recover as much data as possible from a volume with a corrupted directory.

9.1.6.1 Examine Segment 1 — Your first step in recovering data is to determine whether or not segment 1 of the directory is bad. Remember, segment 1 occupies physical blocks 6 and 7 of the device. To examine segment 1, mount the volume and try to get an ordinary listing of the files. Use the DIRECTORY monitor command without any options.

If you get an immediate *?MON-F-Directory I/O error* or *?MON-F-Dir I/O err* message, you know that segment 1 is bad. This leaves you with two alternatives: you can reformat and reinitialize the volume (the volume is reusable if a bad block scan shows no bad block in the directory area); or, if you are desperate to recover the data on the volume, you can open the volume in non-file-structured mode with TECO and search for data that resembles source code or other ASCII information that looks familiar. This kind of search is a tedious process; you probably shouldn't even consider it unless you have a video terminal to use with TECO. See Section 9.1.6.4 for information on removing a file from the volume.

If, on the other hand, the DIRECTORY monitor command gives you at least a partial directory listing, you will be able to recover some of the information from the volume by issuing the DIRECTORY/SUMMARY monitor command. The /SUMMARY option lists information up to but not including the bad segment. To recover as many files as possible, you must repair the directory by linking around the bad segment.

9.1.6.2 Follow the Chain of Segments — Use SIPP to open the volume in non-file-structured mode. Look first at location 6000, which is the start of the header for directory segment 1. It contains the total number of segments available. Location 6002 contains the number of the next segment, and location 6004 shows the highest segment in use. (To review the directory header words and their meanings, see Table 9–2.)

To find the absolute location of the next segment, multiply the link word by 2000 and add 4000. For example, if the link word is 2, the next segment starts at location 10000 on the volume. Chain your way through the segments by opening the next segment and following its link word. As you go, make a worksheet for the link information, according to the format shown in Figure 9–18. Continue chaining until you have accounted for all the segments. Remember that segment 1 is always the first segment — that is, nothing links to it. The last segment always links to 0.

Figure 9–18: Worksheet for a Directory Chain with Four Segments

HIGHEST SEGMENT IN USE: 4
NUMBER OF SEGMENTS AVAILABLE: 4

SEGMENT:	LINKED TO:
1	2
2	4
4	3 (bad)
3 (bad)	?

In Figure 9–18, segment 3 must link to 0 – that is, it is the last segment since all the others have been accounted for already. To repair this directory, modify the header of segment 4 so that the link word contains 0 instead of 3. This eliminates segment 3 from the chain. Section 9.1.6.3 describes how to remove the files from the volume.

Figure 9–19 shows a more complicated example. In this case the bad segment is not the last one in the directory.

Figure 9–19: Worksheet for a Directory Chain with Nine Segments

HIGHEST SEGMENT IN USE: 9
NUMBER OF SEGMENTS AVAILABLE: 9

SEGMENT:	LINKED TO:
1	2
2	5
5	4
4	8
8	7 (bad)
7 (bad)	?
3	0
6	9
9	3

In a situation of the kind shown in Figure 9–19, you can follow the chain from segment 1 through segment 8, which points to the bad segment. To continue from that point, enter in the left column the lowest segment number not yet accounted for and follow its link. Remember, if a segment links to 0, it is the last segment in the directory. Continue until all the segments are accounted for in the left column. When you finish, the number that is missing from the right column is the segment to which the bad segment links. In Figure 9–19, this is segment 6. As in the previous example, use SIPP to link around the bad segment. In this case, change the link word in segment 8 to point to segment 6, thus removing segment 7 from the chain.

9.1.6.3 Remove the Data from the Good Segments – Once you have eliminated the bad segment by linking around it, you are ready to save the files whose entries appear in the good segments. Use the monitor COPY command to copy the files to a good volume. The following command, for example, copies all the files from one diskette to another:

```
COPY DX0:*.* DX1:(RET)
```

This procedure removes all the files from the volume except those whose entries appear in the bad segment.

9.1.6.4 Remove the Data from the Bad Segment – You can sometimes save files whose entries appear in the bad segment by using SIPP and DUP. If, when you open the segment with SIPP, block 1 of the segment is unreadable, you should probably give up because chances are that even if block 2 of the segment is readable, it contains old data that is not valid.

If you can read block 1, decode the header and the entries according to the diagram in Figure 9–4. Continuing with SIPP, try to locate the files on the volume. (Section 9.1.2.4 explains how to locate a file on the volume.) Once you establish the starting and ending blocks of a specific file, run DUP and use the following command sequence to transfer the file to a new device:

```
output-filespec=input-device:/G:startblock/E:endblock/I/F
```

You can use the following keyboard monitor command to achieve the same results:

```
COPY/DEVICE/FILES input-device/START:startblock/END:endblock output-filespec
```

When you have finished removing files from the volume, you can return it to another user (if someone wrote over the directory); or, you can reformat and initialize it (if there was a bad block in the directory area). If reformatting does not remove the bad block, label the volume clearly so you don't accidentally use it again.

9.1.7 Interchange Diskette Format

You can use the FILEX /U option (or the monitor COPY/INTERCHANGE command) to transfer data between RT–11 devices and interchange diskettes. An interchange diskette, also known as an IBM floppy disk, consists of 77 tracks. Each track contains 26 decimal sectors, and you can store one record of 128 or fewer characters per sector, using EBCDIC format.

Track 0 of the diskette is reserved for dataset labels, which are a form of directory. The functions of the sectors of track 0 are as follows.

Sectors 1 through 4 are reserved by IBM for system use. There are 80 blanks per sector.

Positions 1 through 13 of sector 5 are used to record the identity of an error track. Positions 1 through 5 contain **ERMAP** to identify the sector as an error map. Sector 5 is not supported by RT–11.

Sector 6 is reserved by IBM for system use. It contains 80 blanks.

Sector 7 is the volume label. Positions 1 through 4 (bytes 0 through 3) contain **VOL1** in EBCDIC. This identifies the diskette as an IBM floppy. Within DIGITAL, these four bytes identify the system that wrote the diskette. RT–11 stores **RT11** here. Other fields in sector 7 identify the diskette, its format, and its owner, and indicate whether or not the diskette uses standard labels. Table 9–4 describes the contents of sector 7.

Table 9–4: Interchange Diskette Sector 7

Offset	Byte	Contents
0	1	V
1	2	O
2	3	L
3	4	1
4	5	Bytes 5 through 10 contain the volume ID field. The ID consists of one to six digits or letters (left-justified); unused positions must contain blanks.
12	11	Access code. Must be blank to permit access to diskette.
13	12	Bytes 12 through 37 are reserved by IBM.
45	38	Bytes 38 through 51 contain the owner ID field. Not all systems use this field.
63	52	Bytes 52 through 76 are reserved by IBM.
114	77	Bytes 77 and 78 are the record sequence field, or interleave factor.
115	78	Two blanks represents 1:1 interleave.
116	79	Reserved by IBM.
117	80	Label version field. **W** indicates standard labels.

Sectors 8 through 26 are the dataset labels. They are 40 words long, and contain directory information. Table 9–5 describes the contents of sectors 8 through 26.

Table 9–5: Interchange Diskette Sectors 8 Through 26

Offset	Byte	Contents
0	1	H
1	2	D
2	3	R
3	4	1
4	5	Reserved.
5	6	Bytes 6 through 13 contain the user name for the dataset (the file label).
15	14	Bytes 14 through 22 are reserved.
26	23	Bytes 23 through 27 contain the block/record length. The default is 80, but 128 is possible.
33	28	Reserved.
34	29	Bytes 29 and 30 contain two EBCDIC characters representing the track number of the beginning of data.
36	31	EBCDIC 0 (octal 360).
37	32	Bytes 32 and 33 contain two EBCDIC characters representing the sector number of the beginning of data.
41	34	Reserved.
42	35	Bytes 35 and 36 contain two EBCDIC characters representing the number of the last track reserved for this dataset.
44	37	EBCDIC 0 (octal 360).
45	38	Bytes 38 and 39 contain two EBCDIC characters representing the number of the last sector reserved for this dataset.
47	40	Reserved.
50	41	Bypass indicator.
51	42	Dataset security.
52	43	Write protect.
53	44	Blank for data interchange (octal 100).
54	45	Multi-volume indicator: C = Continued, L = Last, blank = Not continued.
55	46	Bytes 46 and 47 contain the volume sequence number.
57	48	Bytes 48 and 49 contain the creation year (such as **80**).
61	50	Bytes 50 and 51 contain the creation month.
63	52	Bytes 52 and 53 contain the creation day.
65	54	Bytes 54 through 66 are reserved.
102	67	Bytes 67 through 72 contain the expiration date (in the same format as the creation date).
110	73	Verify mark: V = Dataset verified, blank = Not verified.
111	74	Reserved.
112	75	Bytes 75 through 79 contain the number of the next unused track and sector within this dataset.
113	76	Bytes 75 and 76 contain the track number.
114	77	EBCDIC 0.
115	78	Bytes 78 and 79 contain the number of the next unused sector.
117	80	Reserved.

9.2 Sequential-Access Devices

The two RT–11 devices that are file-structured but not random-access are magtape and cassette. This section describes the formats of those two sequential-access devices and shows how RT–11 stores files on them.

9.2.1 Magtape Structure

With RT–11 V05 you can read magtapes created with versions V2C, V03, V03B, and V04. RT–11 automatically writes magtapes using a subset of the VOL1, HDR1, and EOF1 labels (ANSI standard X3.27 level 1).

RT–11 magtape implementation includes the following restrictions:

1. There is no EOV (end-of-volume) support. This means that no file can continue from the end of one tape volume to another volume.
2. RT–11 does not ignore noise blocks on input.
3. RT–11 assumes that data is written in records of 512 characters per block. The logical record size equals the physical record size.

NOTE

> The hardware magtape handler (as opposed to the file structure magtape handler) can read data in any format at all. You can also make use of .SPFUN programmed requests and the file structure magtape handler to read tapes with data in a nonstandard format (see Chapter 10 for details). The RT–11 utility programs, such as PIP, DUP, and DIR, can only read and write tapes in the standard RT–11 format of 512-character blocks.

4. RT–11 does not check access fields and therefore provides no volume protection.

In the following examples, an asterisk (*) represents a tape mark. The structure of the actual tape mark itself depends on the encoding scheme that the hardware uses. A typical nine-channel NRZ tape mark consists of one tape character (octal 23) followed by seven blank spaces and an LRCC (octal 23). Consult the hardware manual for your own tape device if the format of the tape mark is important to you.

A file stored on magtape has the following format:

HDR1 * data * EOF1 *

A volume containing a single file has the following format:

VOL1 HDR1 * data * EOF1 * * *

A volume containing two files has the following format:

VOL1 HDR1 * data * EOF1 * HDR1 * data * EOF1 * * *

A double tape mark following an EOF1 * label indicates logical end of tape. (Note that the EOF1 label is considered to consist of the actual EOF1 information plus a single tape mark.)

A magtape that has been initialized has the following format:

VOL1 HDR1 * * EOF1 * * *

A bootable magtape is a multi-file volume that has the following format:

VOL1 BOOT HDR1 * data * EOF1 * * *

To create an RT–11 bootable magtape, you must copy the primary bootstrap by using the INITIALIZE/FILE:MBOOT command. The primary bootstrap is represented by **BOOT** in the format given for a bootable magtape. It occupies a 256-word physical block. The first real file on the tape must be the secondary bootstrap, the file MSBOOT.BOT. If the tape is designed to allow another user to create another bootable magtape, you should copy the file MBOOT.BOT to the tape, as a file. (This is in addition to copying it into the boot block at the beginning of the tape.) More detailed instructions for building bootable magtapes are in the *RT–11 System Generation Guide*.

MBOOT and MSBOOT inspect the I/O page for the presence or absence of standard magtape device registers to determine which type of magtape controller is available. Make sure that other peripheral devices on your system do not use addresses in the I/O page normally used by another magtape controller. MBOOT and MSBOOT might attempt to use the magtape controller assigned to those addresses rather than the magtape controller actually installed on your system.

Each label on the tape, as shown in the formats of the various magtape structures, occupies the first 80 bytes of a 256-word physical block, and each byte in the label contains an ASCII character. (That is, if the content of a byte is listed as '1', the byte contains the ASCII code and not the octal code for '1'.) Table 9–6 shows the contents of the first 80 bytes in the three labels. Note that the VOL1, HDR1, and EOF1 occupy a full 256-word block each, of which only the first 80 bytes are meaningful.

The meanings of the table headings for Table 9–6 are as follows:

CP:	Character position in label
Field Name:	Reference name of field
L:	Length of field in bytes
Content:	Content of field
(space):	ASCII space character

9.2.2 Cassette Structure

A blank, newly initialized TU60 cassette appears in the format shown in Figure 9–20.

Figure 9–20: Initialized Cassette Format

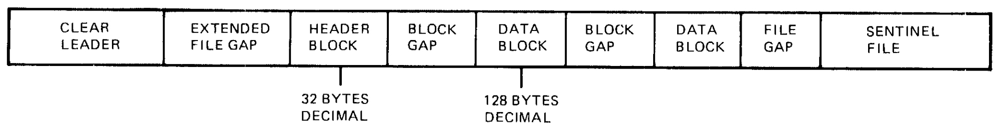
CLEAR LEADER	EXTENDED FILE GAP	SENTINEL FILE 32 BYTES DECIMAL	UNPREDICTABLE INFORMATION

Table 9–6: ANSI Magtape Labels in RT–11

CP	Field Name	L	Content
Volume Header Label (VOL1)			
1–3	Label identifier	3	VOL
4	Label number	1	1
5–10	Volume identifier	6	Volume Label. If you do not specify a volume ID at initialization time, the default is RT11A(space).
11	Accessibility	1	(Space)
12–37	Reserved	26	(Spaces)
38–50	Owner identifier	13	CP38 = D This means tape CP39 = % was written by CP40 = B DEC PDP–11. CP41–50 = Owner Name. Maximum is 10 characters; default is (spaces).
51	DEC standard version	1	1
52–79	Reserved	28	(Spaces)
80	Label standard version	1	3
File Header Label (HDR1)			
1–3	Label identifier	3	HDR
4	Label number	1	1
5–21	File identifier	17	The six-character ASCII file name, dot, three-character file type. (You can use spaces to pad the file name to six characters; you can write the dot without the padding.) This field is left-justified and followed by spaces.
22–27	File set identifier	6	RT11A(space)
28–31	File section number	4	0001
32–35	File sequence number	4	First file on tape has 0001. This value is incremented by 1 for each succeeding file. On a newly initialized tape, this value is 0000.
36–39	Generation number	4	0001
40–41	Generation version	2	00
42–47	Creation date	6	(Space) followed by (year*1000) + day in ASCII; (space) followed by 00000 if no date. For example, 2/1/75 is stored as (space)75032.
48–53	Expiration date	6	(Space) followed by 00000 indicates an expired file.
54	Accessibility	1	(Space)
55–60	Block count	6	000000
61–73	System code	13	DECRT11A(space) followed by spaces.
74–80	Reserved	7	(Spaces)
First End-of-File Label (EOF1)			
This label is the same as the HDR1 label, with the following exceptions:			
1–3	Label identifier	3	EOF
55–60	Block count	6	Number of data blocks since the preceding HDR1 label, unless you issue an .SPFUN programmed request. If you issue .SPFUNs, the block count is 0. However, if the only special function operations you do are 256-word .SPFUN writes, the block count is accurate.

A cassette with a file on it appears as shown in Figure 9–21. The header block contains file names.

Figure 9–21: Cassette with Data

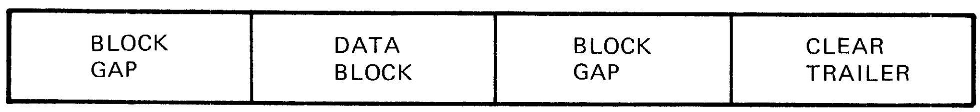


Files normally have data written in 128-byte decimal blocks. You can alter this by writing cassettes in hardware mode. In hardware mode, your program must handle the processing of any headers and sentinel files. In software mode, the handler automatically does this.

Figure 9–21 illustrates a file terminated in the usual manner by a sentinel file. However, the physical end-of-cassette can occur before the actual end of the file. This format appears as shown in Figure 9–22.

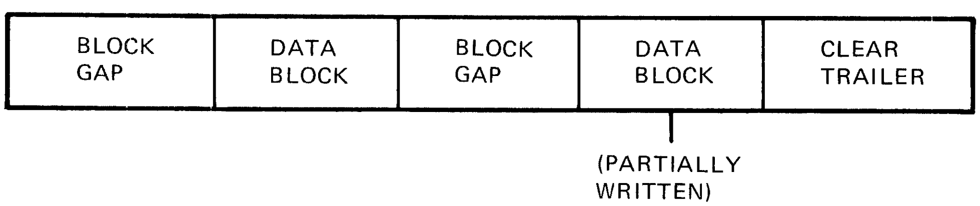
Figure 9–22: Physical End of Cassette

(1)



OR:

(2)



In case 2, for multi-volume processing the partially written block must be rewritten as the first data block of the next volume.

The file header is a 32-byte decimal block that is the first block of any data file on a cassette. If the first byte of the header is null (000), the header is interpreted as a sentinel file, which is an indication of logical end-of-cassette. The format of the header is illustrated in Table 9–7. The data in Table 9–7 is binary (that is, 0 equals a byte of 0) unless it is specified to be ASCII.

Table 9–7: Cassette File Header Format

Byte Number	Contents
0–5	File name in ASCII characters (ASCII implies a seven-bit code). The first character of a deleted file's name contains either a binary 0 or a binary 177.
6–8	File type in ASCII characters.
9	Data type (0 for RT–11).
10–11	Block length of 128 decimal, 200 octal (byte 10 = 0, high order; byte 11 = 200, low order).
12	File sequence number (0 for single volume file or the first volume of a multivolume file; successive numbers are used for continuations).
13	Level 1 (This byte is a 1. You must change this byte to 0 if you are using CAPS–11 to load files.
14–19	Date of file creation (six ASCII digits representing day (0–31); month (0–12); and last two digits of the year; 0 or 40 octal in first byte means no date present).
20–21	0
22	Record attributes (0 is RT–11 cassette).
23–28	Reserved for DIGITAL.
29–31	Reserved for your special applications.

Chapter 10
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:

1. Magtape handlers: MM, MS, and MT
2. Cassette handler: CT
3. Diskette handlers: DX and DY
4. Card reader: CR
5. High-speed paper tape reader and punch: PC
6. Console terminal handler: TT
7. Disk handlers: DL and DM
8. Null handler: NL
9. DECtape II handler: DD
10. MSCP class disk handler: DU
11. Virtual Memory handler: VM
12. Logical disk handler: LD

10.1 Magtape Handlers (MM, MS, MT)

Magnetic tape is file-structured, but not random-access. This means that it stores files sequentially but has 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, which gives you full access to the tape controller without concern for the specifics of the device. See Chapter 9 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.

RT–11 magtape handlers exist in two versions: a hardware handler and a file structure handler. All the handlers are included in the distribution kit. Hardware handlers are named MMHD.SYS, MSHD.SYS, and MTHD.SYS. File structure handlers are named MM.SYS, MS.SYS, and MT.SYS.

The handlers for MM and MT accept SET commands to set the number of tracks, the density, and the parity of the tape drive; these commands apply to all units of a particular controller. These commands are described in Chapter 4 of the *RT–11 System User's Guide*. The MS handler does not accept SET commands.

The MM and MT handlers support up to eight tape drives with one controller. The MS handler supports up to eight drives and eight controllers. DIGITAL recommends that you use a file structure handler (unless special circumstances indicate that a hardware handler is appropriate), since only the file structure handlers can communicate with the RT–11 system utility programs. The following sections describe these handlers.

This chapter uses some magtape-specific abbreviations. They are: **BOT**, for beginning-of-tape; **EOT**, for physical end-of-tape; **LEOT**, for logical end-of-tape, and **EOF**, for end-of-file. LEOT consists of an EOF1 label (which includes one tape mark) followed by two tape marks.

10.1.1 File Structure Magtape Handler

The file structure magtape handlers combine the hardware handler, described in Section 10.1.2, with a file structure module. The file structure module, which is designed to operate with any magtape handler, permits the handler to accept file structure requests. The file structure magtape handler is a superset of the hardware handler. You can issue hardware commands to the file structure handler. The file structure magtape handlers are named MM.SYS, MS.SYS, and MT.SYS. The distributed versions of these handlers support tape drives 0 and 1. You can add support for more drives at system generation time.

A tape containing two files has the following format:

VOL1 HDR1 * data * EOF1 * HDR1 * data * EOF1 * * *

VOL1, **HDR1**, and **EOF1** are ANSI tape labels. The asterisk (*) represents a tape mark. See Chapter 9 for more information on magtape formats.

10.1.1.1 Searching by Sequence Number – The file structure handler 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.

10.1.1.2 Searching by File Name — The file structure handler 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 handler 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 and padded with spaces, if necessary.

typ is a file type left-justified in a three-character field.

The algorithm the handler uses allows RT–11 V03 and later versions to read and match tapes written under V2C and earlier versions. 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 first 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 < 9 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.

10.1.1.3 Programmed Requests — The following sections describe how programmed requests for magtape function.

.ENTER Programmed Request

The .ENTER programmed request writes a HDR1 label and tape mark on the tape, and leaves the tape positioned after the tape mark. The request initializes some internal tables, including entries for the last block written and current block number. (The last block or file on tape is always the most recent one written.) The information for the internal tables and entries for the last written block is correct unless an .SPFUN request is performed on that channel. Normally, files opened with an .ENTER request do not have special functions performed on them, except when a nonstandard block size is to be written (one that is not 256 words long). To write a nonstandard block, open the file with an .ENTER request; then issue an .SPFUN write request. Close the file with a .CLOSE request after the operation is complete. If a file search is to be performed, open the tape non-file-structured with a .LOOKUP request. Table 10–1 shows the sequence number values for .ENTER requests.

The .ENTER programmed request has the following format:

```
.ENTER  area,chan,dblk,,seqnum
```

The .ENTER request issues a directory hard error if errors occur while entering the file.

.LOOKUP Programmed Request

The .LOOKUP request causes a specific HDR1 label to be searched and read. After this request, the tape is left positioned before the first data block of the file. Table 10–3 shows the sequence number values for the .LOOKUP request, which has the following format:

```
.LOOKUP    area,chan,dblk,seqnum
```

Table 10–1: Sequence Number Values for .ENTER Requests

Seqnum Argument	File Name	Action Taken	Tape Position
>0	not null	Position at file sequence number and perform an .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 10–2.

Table 10–2: .ENTER Errors

Byte 52 Code	Meaning
0	Channel in use.
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.
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	Illegal argument error. A seqnum argument in the range –3 through –32767 was detected. A null file name was passed to .ENTER.

Table 10–3: Sequence Number Values for .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.
>0	null	Perform a filestructured .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. 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 LEOT.

NOTE

If a channel is opened with a non-file-structured .LOOKUP (file name null and file sequence number 0 or –1), .READ, .READC, and .READW requests use an implied word count equal to the physical block size on the tape; .WRITE, .WRITC, and .WRITW requests use the word count to determine the block size on the tape. This convention is used instead of using 512 bytes as a default block size and performing blocking and unblocking. This request is almost identical to an .SPFUN read or write that does not report any errors (**blk** = 0). Also note that the error and status block must not be overlaid by the USR.

The .LOOKUP returns the errors shown in Table 10–4.

Table 10–4: .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	Illegal argument error. A **seqnum** argument in the range –2 through –32767 was detected. A .LOOKUP request for the hardware handler must have a positive sequence number.

The .LOOKUP request issues the directory hard error in the same manner as the .ENTER request.

.READx Programmed Requests

NOTE

> The term .READx/.WRITx refers to the following group of programmed requests: .READ, .READC, .READW, .WRITE, .WRITC, and .WRITW.

The .READx requests read data from magtape in blocks of 512 bytes each. This group of requests is described here for files opened with the .ENTER and file-structured .LOOKUP requests. In addition to this description, there are .READx and .WRITx descriptions appropriate to non-file-structured .LOOKUP requests (see Sections 10.1.2.11 and 10.1.2.12).

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 requests (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 (0 code returned). 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 is not positioned after the last block written. This causes loss of information when a program issues a read for a block that was written before the last block and fails to reread the last block, thereby positioning the tape at the end of the data.

The guidelines for block numbers are as follows:

1. .READx: 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. .WRITx and .READx: On an entered file, a check is made 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 format of the .READx request is as follows:

.READx   area,chan,buf,wcnt,blk[,crtn]

Table 10–5 shows the errors the .READx requests return.

Table 10–5: .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.

.WRITx Programmed 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 or with a non-file-structured .LOOKUP. The .WRITx requests have the following format:

.WRITx   area,chan,buf,wcnt,blk[,crtn]

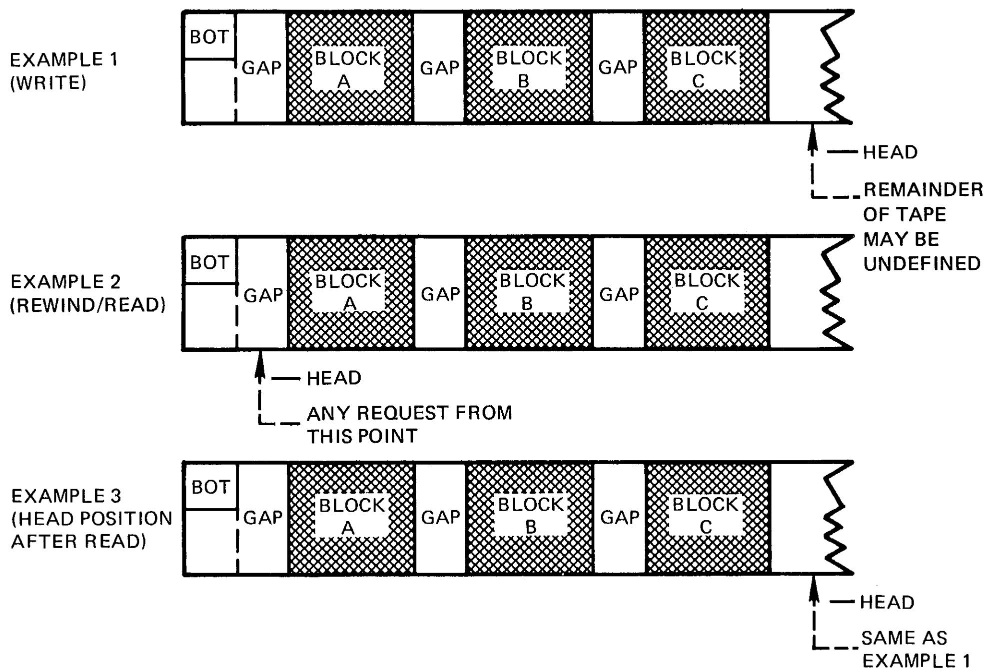
Table 10–6 shows the errors the .WRITx requests return.

Table 10–6: .WRITx Errors

Byte 52 Code	Meaning
0	End-of-tape. This means that the data 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 may be undefined (see Figure 10–1).

Figure 10–1: Operations Performed After the Last Block Written on Magtape



In example 1 in Figure 10–1, 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 10–1, 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.

.DELETE and .RENAME Programmed Requests

The .DELETE and .RENAME requests are invalid operations on magtape, and any attempt to execute them results in an illegal operation code (code 2) being returned in byte 52.

.CLOSE Programmed Request

The .CLOSE request operates in three different ways, depending on how the file was opened:

1. 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.

2. 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.
3. When a file is opened with a non-file-structured .LOOKUP, no action is taken and the channel becomes free.

The .CLOSE request has the following format:

```
.CLOSE    chan
```

This request issues a directory hard error if a malfunction is detected. The error can be recovered with the .SERR request.

.SPFUN Programmed Request

The .SPFUN programmed request can perform asynchronous directory operations without the USR, which makes it useful for long tape searches. It is particularly useful for programmers in multi-job systems who do not want to wait for the long tape searches that can occur during .ENTER and .LOOKUP requests. It is also useful and desirable for FB and XM users who do not want to lock the USR. This request allows the .ENTER and .LOOKUP requests to be issued after a non-file-structured .LOOKUP assigns a channel to the magtape handler. Unpredictable results occur if this request is issued for a channel that was not opened with a non-file-structured .LOOKUP. The .SPFUN request has the following format:

```
.SPFUN    area,chan,#–20.,buf,,blk
```

–20. is the code for the asynchronous directory request.

buf is the address of a seven-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 four-word error and status block used for returning .LOOKUP and .ENTER errors that are normally reported in byte 52. Only the first word of *blk* is used by this request. The other three words are reserved for future use and must be zero. When the first word of *blk* is 0, no error information is returned. This block must always be mapped when the program is running in the extended memory environment. Figure 10–2 shows a programming example.

Figure 10–2: 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

        ASYREQ= -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.

START:  .LOOKUP #AREA,#CHAN,#NFSBLK,#0  ;Open a channel

                                ;for the next request
        BCS     LOOKER          ;Branch if error occurred
        .SPFUN  #AREA,#CHAN,#ASYREQ,#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 nonfile-
        .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
```

Figure 10–2: Asynchronous Directory Operation Example (Cont.)

```
;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

SYMBOL TABLE

AREA    000130R         ERRBLK  000170R         LOOKUP= 000003
ASYERR  000317R         FILFND  000104R         NFSBLK  000142R
ASYREQ= 177754          FNF     000001          NOTFND  000112R
CHAN  = 000000          FNFERR  000300R         OK      000242R
CLOSE   000116R         FSN   = 000000          START   000000R
COMBLK  000152R         LOOERR  000200R         ...V1 = 000003
ENTER = 000004          LOOKER= 000076R         ...V2 = 000027

. ABS.  000000     000
        000354     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  9472 WORDS  ( 37 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  72 PAGES
,SSM012/L:TTM=SSM102
```

10.1.1.4 Issuing Hardware Handler Calls with the File Structure Module — The magtape handler is designed to perform two distinct types of access. One type of access is file-oriented; it makes the magtape appear to be a disk. In other words, it makes the magtape as device-independent as possible. The other type of access allows access to the hardware commands such as read, write, space, and so on, but the programmer need not know whether the device is a TM11 or TJU16, for example (see Section 10.1.2).

When the handler accesses magtape using file-oriented commands, it keeps track of the file sequence number where the tape is positioned. Thus, it can optimize tape movement during file searches. When the handler accesses data in a magtape file using the .READx/.WRITx requests, it keeps track of the current block number as well as the last block number accessible. The block number argument can be used to simulate a random-access device even on files opened with .ENTER.

The two access methods just described can be combined; that is, it is possible to use hardware handler tape movement commands on a magtape file. However, doing so has the following implications:

1. When the first hardware handler command is received, the stored file sequence number and block number information described above 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. Also note that

the tape will no longer be an ANSI-compatible magtape. 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.

2. The only exception to the rule explained above occurs when you need to open the tape as file-structured and write data blocks that are not the standard 512-byte size that magtape .WRITx requests use. The magtape handler keeps track of the number of blocks written and the EOF1 labels are correct as long as no commands other than the .SPFUN write command are used. If other commands are used, the file size is lost.

NOTE

DIGITAL recommends that programmers issue .SPFUN commands to a magtape file only for the case described in 2 above.

10.1.2 Hardware Magtape Handler

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.

If you do not need the extra file structure support, use the hardware handlers. You must perform a SYSGEN 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 MT handler with the hardware MT handler.

Command	Action
`REMOVE MT`	Removes the file-structure handler.
`RENAME/SYS MT.SYS MTFS.SYS`	Saves the file-structure handler.
`RENAME/SYS MTHD.SYS MT.SYS`	Creates the new hardware handler.
`INSTALL MT`	Installs the new handler.

You access the hardware handler with non-file-structured .LOOKUP programmed requests, with .SPFUN special function requests, and with .READ, .READC, .READW, .WRITE, .WRITC, .WRITW, and .CLOSE requests. The hardware handler can perform I/O operations on physical blocks, position the tape, and recover from errors.

10.1.2.1 Exception Reporting — Those .SPFUN requests that are accepted by the hardware handler report end-of-file and hard error conditions through byte 52 in the system communication area. In addition, they use the argument normally used for a block number as a pointer to a four-word error and status block in order to return qualifying information about exception conditions. When the block number argument is 0, no qualifying information is returned. Note that the contents of these words are undefined when no exception conditions have occurred and the carry bit is not set. The block is defined as follows: words 1 and 2 are qualifying information; words 3 and 4 are reserved for DIGITAL and must be 0. You need initialize words 3 and 4 only once in your program. The system modifies words 1 and 2 only when it reports exception information.

Qualifying information returned in the first word for the end-of-file condition is shown in Table 10–7. Note that the carry bit is set, and byte 52 is zero.

Table 10–7: End-of-file Qualifying Information

First Word Octal Code	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).

When a tape mark is detected during a spacing operation, the number of blocks not spaced is returned in the second word.

EOT, tape mark, and BOT are returned as an EOF by the hardware handler.

Qualifying information returned in the first word for the hard error condition is shown in Table 10–8. Note that the carry bit is set, and byte 52 is 1.

Table 10–8: Hard Error Qualifying Information

First Word Octal Code	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.
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.

The hardware handler issues a hard error if it receives any request other than non-file-structured .LOOKUP, .CLOSE, or any .SPFUN request not defined for the hardware handler.

When a program runs in the XM environment, the status block for error reporting must be mapped at all times.

10.1.2.2 Reading and Writing Physical Blocks — The hardware handler reads and writes blocks of any size. Requests for reading and writing a variable number of words are implemented through two .SPFUN codes.

The .SPFUN request to read a variable number of words in a block has the following format:

.SPFUN area,chan,#370,buf,wcnt,blk[,crtn]

370 is the function code for a read operation.

blk is the address of a four-word error and status block used for returning the exception conditions.

crtn is an optional argument that specifies a completion routine to be entered after the request executes.

This request returns the errors shown in Table 10–9. Additional qualifying information for these errors is returned in the first two words of the *blk* argument status block.

Table 10–9: SPFUN 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 = 2	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.

The .SPFUN request to write a variable number of words to a block has the following format:

.SPFUN area,chan,#371,buf,wcnt,blk[,crtn]

371 is the function code for a write operation.

This request returns the errors shown in Table 10–10. Additional qualifying information for these errors is returned in the first two words of the status block.

Table 10–10: SPFUN 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.

10.1.2.3 Spacing Forward and Backward — The hardware handler accepts a command that spaces forward or backward 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 command to space forward by block has the following format:

.SPFUN area,chan,#376,,wcnt,blk[,crtn]

376 is the function code for a forward space operation.

wcnt is the number of blocks to space past (must not exceed 65534).

crtn is an optional argument that specifies a completion routine to be entered after the request executes.

This request returns the errors shown in Table 10–11. Additional qualifying information for these errors is returned in the first two words of the status block.

Table 10–11: .SPFUN 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).
		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.

NOTE

> Due to hardware restrictions, DIGITAL recommends that no forward space commands be issued if the reel is positioned past the EOT marker.

The format of the command to space backward by block is as follows:

.SPFUN area,chan,#375,,wcnt,blk[,crtn]

375 is the function code for a backspace operation.

This request returns the errors shown in Table 10–12. Additional qualifying information for these errors is returned in the first two words of the status block.

10.1.2.4 Rewinding — The handler accepts a rewind command and 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 rewind request has the following format:

.SPFUN area,chan,#373,,,blk[,crtn]

373 is the function code for the rewind operation.

Table 10–12: .SPFUN 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.

crtn is an optional argument that specifies a completion routine to be entered after the request executes.

This request returns the error shown in Table 10–13. Additional qualifying information is returned in the status block.

Table 10–13: .SPFUN 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.

10.1.2.5 Rewinding and Going Off Line — 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 rewind and go off line request has the following format:

```
.SPFUN    area,chan,#372,,,blk[,crtn]
```

372 is the function code for the rewind and go off line operation.

crtn is an optional argument that specifies a completion routine to be entered after the request executes.

This request returns the same error code and qualifying information as the rewind request.

10.1.2.6 Writing with Extended Gap — This request permits you to write on tapes that have bad spots. It is identical to the write request except for its function code, which is 374. The errors are identical to those for the write request.

10.1.2.7 Writing a Tape Mark — The hardware handler accepts a request to write a tape mark. This request has the following format:

.SPFUN area,chan,#377,,,blk[,crtn]

377 is the function code for the write tape mark operation.

This request returns the errors shown in Table 10–14. Additional qualifying information for these errors is returned in the first two words of the *blk* argument status block.

Table 10–14: .SPFUN 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.

10.1.2.8 Error Recovery — Any errors detected during spacing operations cause the recovery attempt to be aborted, and a hard (position) error is reported.

Both the file structure handler and the hardware handler 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 three 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.

10.1.2.9 Non-File-Structured .LOOKUP Programmed Request — The hardware handler accepts a non-file-structured .LOOKUP request, a function that is necessary to open a channel to the device before any I/O operations can be executed. It causes the hardware handler to mark the drive busy so that no other channel can be opened to that drive until a .CLOSE is issued. This request has the following format:

```
.LOOKUP    area,chan,dblk,seqnum
```

seqnum is an argument that specifies whether the tape is to be rewound. When this argument is 0, the tape is rewound. When this argument is –1, the tape is not rewound. Table 10–15 shows the errors this request returns.

Table 10–15: .LOOKUP Errors

Byte 52 Code	Meaning
0 or 1	Not meaningful for this request.
2	Device in use. The drive being accessed is already attached to another channel.
3	Tape drive not available.
4	Invalid argument detected. The file name was not 0, or the *seqnum* argument was not 0 or –1.

10.1.2.10 .CLOSE Programmed Request — The hardware handler accepts the .CLOSE request and causes the handler to mark the drive as available. This request has the following format:

```
.CLOSE    chan
```

10.1.2.11 Non-File-Structured .WRITx Programmed Requests — The hardware handler accepts .WRITx requests that write a variable number of words to a block on tape. The block number field is ignored. These requests have the following format:

```
.WRITx    area,chan,buf,wcnt[,,crtn]
```

These requests return the errors shown in Table 10–16. No additional qualifying information is available.

Table 10–16: .WRITx Errors

Byte 52 error	Meaning
0	The EOT marker has been detected.
1	Hard error occurred on channel.
2	Channel not open.

10.1.2.12 Non-File-Structured .READx Programmed Requests – These requests read a variable number of words from a block on tape. They ignore the EOT marker and only report end-of-file when a tape mark is read. The block number field is ignored. The requests have the following format:

.READx area,chan,buf,wcnt[,,crtn]

These requests return the errors shown in Table 10–17. No additional qualifying information is available.

Table 10–17: .READx Errors

Byte 52 Code	Meaning
0	Attempt to read past a tape mark. Also generated by a block that is too large.
1	Hard error occurred on channel.
2	Channel not open.

10.1.2.13 Enabling 100ips Streaming on a TS05 – The TS05/TSV05 Q-bus magtape hardware has a 100ips streaming mode. To enable this feature you must issue a .SPFUN request with a function code of –9. The form of the .SPFUN is

.SPFUN area, chan, #–9., buffer,, blk

where *area* and *chan* are as defined in the *Programmer's Reference Manual*, *blk* is a pointer to a 4-word error block, and *buffer* is a pointer to a word which enables or disables streaming. If *buffer* contains a 1, streaming is enabled, if *buffer* contains a 0, streaming is disabled.

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 .SPFUN function is valid only for a TS05/TSV05 running on a Q-bus machine using the MS handler. If a .SPFUN function code of –9 is used with any other magtape handler or if it is used with the MS handler running a TS11 magtape, the .SPFUN is ignored.

If you want to run a TS05/TSV05 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.

10.1.3 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 System User's Guide* for information on transporting tapes from RT–11 to other systems.)

10.1.3.1 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. In order to ensure that an ANSI file structure is written, issue the following commands:

Commands	Action
`ASSIGN ddn:,ANSI`	Allocates the device to the job and ensures that an ANSI file structure is used.
`RUN $PIP`	
`ddn:xxxxxx/ZE`	PIP initializes the tape; *xxxxxx* is the volume ID.
`Really zero ddn:?+YES`	PIP prompts before initializing the tape.
`PIP ddn:=TEST1.MAC,TEST2.MAC`	PIP copies files to the tape.
`DEASSIGN ddn:`	Deallocates the device.

10.1.3.2 From RSX–11M – RSX–11M needs the following commands to access a magtape:

Commands	Actions
`ALL ddn:`	Allocates a drive.
`INIT ddn:RT11`	Initializes the tape and gives the name **RT11** as the volume identification.
`MOU ddn:RT11`	Mounts the tape volume.

```
PIP+ddn:=[13,14]TEST1.MAC,TEST2.MAC
```
Copies files to the tape.

`DMO ddn:RT11`	Dismounts the tape volume.
`DEA ddn:`	Deassigns the drive.

10.1.3.3 From RSX–11D and IAS – Use the following commands to write an ANSI tape on RSX–11D or IAS:

Commands	Actions
`INIT ddn:RT11`	Initializes the tape and gives the name **RT11** as the volume identification.
`MOU ddn:RT11`	Mounts the tape volume.

For RSX–11D, use PIP to write files to the tape; for IAS, use the COPY command.

`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.

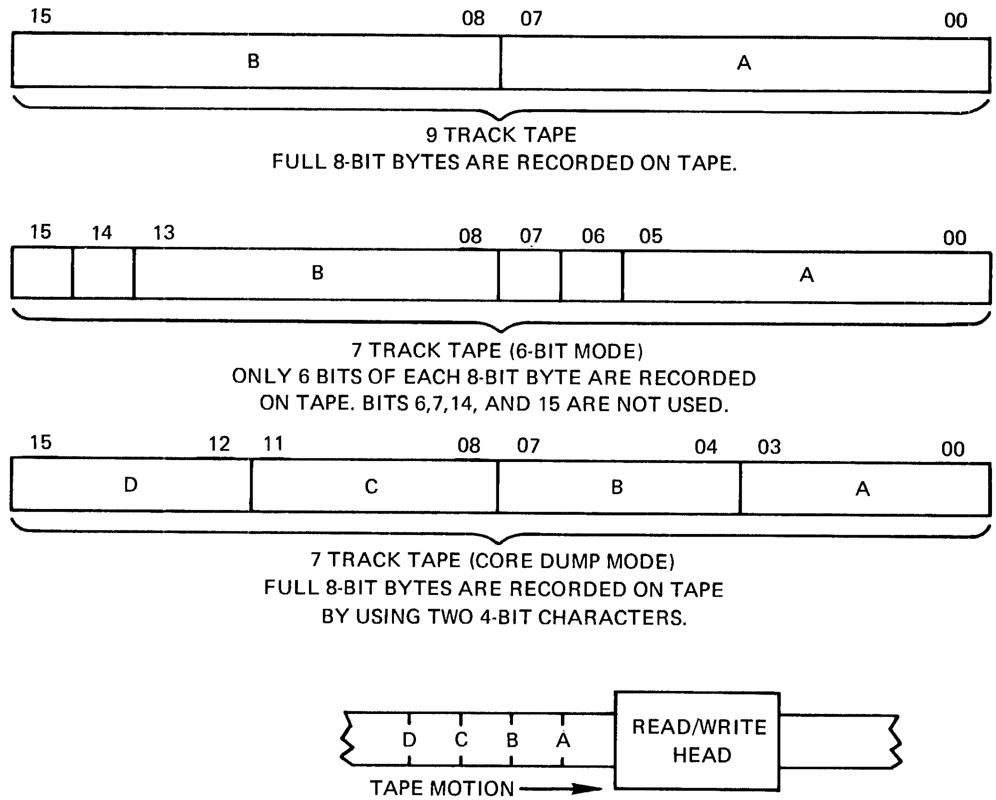
When you write files to be read under RT–11, the only valid block size the utility programs use is 512 characters per block. However, the DIR program will list the directory of any ANSI compatible tape.

10.1.4 Seven-Track 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. 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 10–3 illustrates the differences between six-bit mode and core dump 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.

Figure 10–3: Seven-Track Tape



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; the high-order four bits of the PDP–11 byte are then written 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.

10.2 Cassette Handler: CT

The cassette handler can operate in two different modes: hardware mode and software mode. These names refer to the type of operation that can be performed on the device at a given time. Software mode is the normal mode of operation you use when you access the device through any of the RT–11 utility programs. In software mode, the handler automatically attends to file headers and uses a fixed record length of 64 words to transfer data. (For more information on cassette structure, see Chapter 9).

Hardware mode allows you to read or write any format, using any record size. In this mode, the handler interprets the word count as the physical record size.

When the handlers are initially loaded by either the .FETCH programmed request or the monitor LOAD command, only software functions are permitted. To switch from software to hardware mode, issue either a rewind or a non-file-structured .LOOKUP. (A non-file-structured .LOOKUP is a .LOOKUP in which the first word of the file name is null.)

In software mode, the following functions are permitted:

Request	Action
.ENTER	Opens new file for output.
.LOOKUP	Opens existing file for I/O.
.DELETE	Deletes an existing file.
.CLOSE	Closes a file opened with .LOOKUP or .ENTER.
.READx/.WRITEx	Performs data transfer requests.

In .ENTER, .LOOKUP, and .DELETE, you can specify an optional file count argument, which results in the following actions:

Argument	Action
0	A rewind is done before the operation.
>0	No rewind is done. The value of the count is taken as a limit of how many files to look at before performing the operation. (For example, a count of 2 looks at two files at most. A count of 1 looks at only the next file.)
<0	A rewind is done. The absolute value of the count is then used as the limit.

If the file indicated in the request is located before the limit is exhausted, the search succeeds at that point.

Consider the following example:

```
        .LOOKUP #AREA,#0,#PTR,#5
        BCS     A1
        .
        .
        .
AREA:   .BLKW   10.
PTR:    .RAD50  /CT0/
        .RAD50  /EXAMPLMAC/
```

In this case, the file count argument is +5, indicating that no rewind is to be done and that CT0 is to be searched for the indicated file EXAMPL.MAC. If the file is not found after four files have been skipped, or if an EOT occurs in that space, the search is stopped, and the tape is positioned either at EOT or at the start of the fifth file. If the named file is found within the five files, the tape is positioned at its start. If EOT is encountered first, an error is generated.

The following example performs a rewind, then uses a file count of five as in the previous example.

```
.LOOKUP #AREA,#0,#PTR,#-5
```

10.2.1 Handler Functions

The following sections describe the functions performed by the cassette handler.

10.2.1.1 .LOOKUP Request — If the file name (or the first word of the file name) is zero, the operation is considered to be a non-file-structured .LOOKUP. This operation puts the handler into hardware mode. A rewind is automatically done in this case.

If the file name is not null, the handler tries to find the indicated file. The .LOOKUP request uses the optional file count argument described in Section 10.2. Only software functions are allowed.

10.2.1.2 .DELETE Request — The .DELETE request eliminates a file of the designated name from the device. The .DELETE request also uses the file count argument and can thus delete a numbered file as well as a named file. When a file is deleted, an unused space is created. However, it is not possible to reclaim that space, as it is when the device is random-access. The space remains unused until the volume is reinitialized and rewritten. If a file name is not present, a non-file-structured .DELETE is performed and the tape is initialized.

10.2.1.3 .ENTER Request — The .ENTER request creates a new file of the designated name on the device. This request uses the optional file count and can thus create a file by name or by number. If the request creates a file by name, the handler deletes any files of the same name. If the request creates a file by number, the indicated number of files is skipped and the tape is positioned at the start of the next file.

NOTE

> Care must be exercised in performing numbered .ENTER requests, as it is possible to create a file in the middle of existing files and thus destroy the files from the next file to the end of the tape.
>
> It is also possible to create more than one file with the same name, since .ENTER only deletes files of the same name it encounters while passing down the tape. If an .ENTER is

issued with a count greater than 0, no rewind is performed before the file is created. If a file of the same name is present at an earlier spot on the tape, the handler cannot delete it. A non-file-structured .ENTER performs the same function as a non-file-structured .LOOKUP but does not rewind the tape. Since both functions allow writing to the tape without regard for the tape's file structure, they should be used with care on a file-structured tape.

10.2.1.4 .CLOSE Request — The .CLOSE request terminates operations to a file on cassette and resets the handler to allow more .LOOKUP, .ENTER, or .DELETE requests. If a .CLOSE request is not performed on a file created with .ENTER, the EOT label will be missing and no new files can be created on that volume. In this case, the last file on the tape must be rewritten and closed to create a valid volume.

10.2.1.5 .READ/.WRITE Requests — The .READ and .WRITE requests can be issued either in hardware or software mode. In software mode (file opened with .LOOKUP or .ENTER), records are written with a fixed size of 64 words. The word count specified in the operation is translated to the correct number of records. On a .READ request, the user buffer is filled with zeroes if the word count exceeds the amount of data available.

Following is a description of how the various arguments for .READ and .WRITE are used.

1. Block number (**blk**)

   Only sequential operations are performed. If the block number is 0, the cassette is rewound to the start of the file. Any other block number is disregarded.

2. Word count (**wcnt**)

   If the word count is 0, the following conditions are possible:

   If the block number is nonzero, the operation is a file name seek. The block number is interpreted as the file count argument, as discussed in the example of .LOOKUP. The buffer address should point to the Radix–50 equivalents of the device and file to be located. This feature essentially allows an asynchronous .LOOKUP to be performed. The standard .LOOKUP request does not return control to the user program until the tape is positioned properly, whereas this asynchronous version returns control immediately and interrupts when the file is positioned.

   You can then issue a synchronous, positively numbered .LOOKUP to the file just positioned, thus avoiding a long synchronous search of the tape.

If the block number is 0, a cyclical redundancy check error occurs.

10.2.2 Cassette Special Functions

The following sections describe the valid hardware mode functions for the handler and include examples of how to call them. In general, special functions are issued by using the .SPFUN programmed request. The special functions require a channel number as an argument. The channel must initially be opened with a non-file-structured .LOOKUP request, which places the handler in hardware mode.

The general form of the .SPFUN request is:

```
.SPFUN    area,chan,func,buf,wcnt,blk,crtn
```

func is the code for the function to be performed.

10.2.2.1 Rewind — The rewind request rewinds the tape to its load point. This puts the unit in hardware mode in the same manner as a non-file-structured .LOOKUP, where any of the other functions can be performed. Unless a completion routine is specified, control does not return to the calling program until the rewind completes. This request has the following format:

```
.SPFUN    area,#0,#373,#0,#0,#0,crtn
```

crtn is a completion routine to be entered when the operation is complete.

10.2.2.2 Last File — The last file request rewinds the cassette and positions it immediately before the sentinel file (LEOT). This request has the following format:

```
.SPFUN    area,#0,#377,#0,#0,#0[,crtn]
```

10.2.2.3 Last Block — The last block request rewinds one record. This request has the following format:

```
.SPFUN    area,#0,#376,#0,#0,#0,[,crtn]
```

10.2.2.4 Next File — The next file request spaces the cassette forward to the next file. This request has the following format:

```
.SPFUN    area,#0,#375,#0,#0,#0[,crtn]
```

10.2.2.5 Next Block — The next block request spaces the cassette forward by one record. This request has the following format:

```
.SPFUN    area,#0,#374,#0,#0,#0[,crtn]
```

10.2.2.6 Write File Gap — The write file gap request terminates a file written by the calling program in hardware mode. The following example writes a file gap synchronously:

```
.SPFUN    area,#0,#372,#0,#0,#0
```

The next two examples write file gaps asynchronously:

.SPFUN area,#0,#372,#0,#0,#0,#1

.SPFUN area,#0,#372,#0,#0,#0,crtn

10.2.3 EOF Detection

Since the cassette is a sequential device, the handler for this device cannot anticipate the number of blocks in a particular file, and thus cannot determine if a particular read request is attempting to read past the end of the file. Programs can use the following procedures to determine if the handler has encountered EOF in either software or hardware mode.

In software mode, if EOF is encountered during a read and some data is read, the cassette handler zero-fills the rest of the buffer and returns to the program. The next read attempted on that channel returns with the carry bit set and with the error byte (byte 52) set to indicate an attempt to read past EOF.

In hardware mode, the cassette handler does not report EOF as it does in software mode. The best way that programs can determine if a cassette read has encountered a file gap is to check the device status registers after each hardware-mode read is complete. The following example shows how to check EOF and EOT bits.

```
        TACS=177500             ;TA11 CSR
        TAEOF=4000              ;EOF BIT IN TACS
        TAEOT=20000             ;EOT BIT IN TACS
        .
        .
        .
        .READW  #AREA,#CHNL,#BUFF,#400,BLKNUM
                                ;READ FROM CT
        BCS     EMTERR          ;TEST ERRORS
        TST     @#TACS          ;ERROR BIT SET IN TACS?
        BPL     NOERR           ;IF PLUS, NO
        BIT     #TAEOF,@#TACS   ;YES; WAS IT EOF?
        BNE     EOF             ;IF NE, YES
        .
        .
        .
EOF:    .                       ;EOF ENCOUNTERED
        .
        .
; BOTH THE EOF AND EOT BITS CAN BE CHECKED:

        BIT     #MTSEOF+MTSEOT,@#MTS    ;MT EOF OR EOT?
        BIT     #TAEOF+TAEOT,@#TAEOT    ;CT EOF OR EOT?
```

10.3 Diskette Handlers: DX and DY

The .SPFUN request permits reading and writing of absolute sectors on the diskettes. The DY handler accepts an additional .SPFUN request to determine the size, in 256-word blocks, of the volume mounted in a particular unit. On double-density diskettes, sectors are 128 words long. RT–11 normally reads and writes them in groups of two sectors. On single-density diskettes, sectors are 64 words long. RT–11 reads and writes them in groups of four sectors. Sectors can be accessed individually through the .SPFUN request. The format of the request is as follows:

.SPFUN area,chan,func,buf,wcnt,blk,crtn

func is the code for the function to be performed. The codes and functions are as follows:

Code	Function
377	Read physical sector
376	Write physical sector
375	Write physical sector with deleted data mark
374	Reserved
373	Determine device size, in 256-word blocks, of volume (DY only)

buf for function codes 377, 376, and 375 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.

If the first word is set to 1, a read on a physical sector containing a deleted data mark is indicated. The actual data area of the buffer extends from the second word to the end of the buffer.

buf for function code 373 is the location of a one-word buffer in which the size of the volume in the specified unit is returned. (For single-density diskettes, 494 is returned. For double-density diskettes, 988 is returned.)

wcnt for functions 377, 376, and 375 is the absolute track number, 0 through 76, to be read or written.

wcnt for function 373 is reserved and should be set to 1.

blk for functions 377, 376, and 375 is the absolute sector number, 1 through 26, to be read or written.

blk for function 373 is reserved and should be set to 0.

The diskette should be opened with a non-file-structured .LOOKUP. Note also that the *buf*, *wcnt*, and *blk* arguments have different meanings when used with diskettes. The following example performs a synchronous sector read from track 0, sector 7, into a 65-word area called BUFF.

```
.SPFUN   #RDLIST, #377, #BUFF, #0, #7, #0
```

Each DX and DY handler can support two controllers, and each controller supports two 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 vector 264 and CSR 177170. DX2 and DX3 are drives 0 and 1 of the other controller. 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.

10.4 Card Reader Handler: CR

The card reader handler can transfer data either as ASCII characters in DEC 026 or DEC 029 card codes (see Table 10–18) or as column images controlled by the SET command (see the *RT–11 System User's Guide*). In ASCII mode (SET CR: NOIMAGE), invalid punch combinations are decoded as the error character 134 octal, which is a backslash. In IMAGE mode, no punch combination is invalid; each column is read as 12 bits of data right-justified in one word of the input buffer. The handler continues reading until the transfer word count is satisfied or until a standard EOF card is encountered (12–11–0–1–6–7–8–9 punch in column 1; the rest of the card is arbitrary). On EOF, the buffer is filled with zeroes and the request terminates successfully; the next input request from the card reader gives an EOF error. Note that if the transfer count is satisfied at a point that is not a card boundary, the next request continues from the middle of the card with no loss of information. If the input hopper is emptied before the transfer request is complete, the handler hangs until the hopper is reloaded and the START button on the reader is pressed again. The transfer then continues until completion or until another hopper-empty condition exists. EOF is not reported on the hopper-empty condition. The handler hangs if the hopper empties during the transfer, regardless of the status of the SET CR: HANG/NOHANG option. No special action is required to use the card reader handler with the CM11 mark sense card reader. The program should take into account the fact that mark sense cards can contain fewer than 80 characters. Note also that when the CR handler is set to CRLF or TRIM and is reading in IMAGE mode, unpredictable results can occur.

The card reader handler uses the *blk* argument of the .READx programmed request to determine if a new card should be read. The format of the request is as follows:

```
.READx    area,chan,buf,wcnt,blk
```

If *blk* is 0, a new card is read and the word count argument is filled by characters on that card. Subsequent cards are read, if necessary, to complete the word count. If *blk* is nonzero, the word count argument is first satisfied by characters remaining from a previous card read request, and more cards are read, if necessary, to satisfy the count.

Table 10–18: DEC 026/DEC 029 Card Code Conversions

Zone	Digit	Octal	Character	Name
none				
	none	040		SPACE
	1	061	1	digit 1
	2	062	2	digit 2
	3	063	3	digit 3
	4	064	4	digit 4
	5	065	5	digit 5
	6	066	6	digit 6
	7	067	7	digit 7
12				
(DEC 029)	none	046	&	ampersand
(DEC 026)		053	+	plus sign
	1	101	A	upper-case A
	2	102	B	upper-case B
	3	103	C	upper-case C
	4	104	D	upper-case D
	5	105	E	upper-case E
	6	106	F	upper-case F
	7	107	G	upper-case G
11				
	none	055	–	minus sign
	1	112	J	upper-case J
	2	113	K	upper-case K
	3	114	L	upper-case L
	4	115	M	upper-case M
	5	116	N	upper-case N
	6	117	O	upper-case O
	7	120	P	upper-case P
0				
	none	060	0	digit 0
	1	057	/	slash
	2	123	S	upper-case S
	3	124	T	upper-case T
	4	125	U	upper-case U
	5	126	V	upper-case V
	6	127	W	upper-case W
	7	130	X	upper-case X
8				
	none	70	8	digit 8
	1	140	\`	accent grave
(DEC 029)	2	072	:	colon
(DEC 026)		137	_	backarrow (underscore)
(DEC 029)	3	043	#	number sign
(DEC 026)		075	=	equal sign
	4	100	@	commercial "at"
(DEC 029)	5	047	'	single quote
(DEC 026)		136	^	uparrow (circumflex)
(DEC 029)	6	075	=	equal sign
(DEC 026)		047	'	single quote
(DEC 029)	7	042	"	double quote
(DEC 026)		134	\	backslash
9				

(Continued on next page)

Table 10–18: DEC 026/DEC 029 Card Code Conversions (Cont.)

Zone	Digit	Octal	Character	Name
	none	071	9	digit 9
	2	026	CTRL/V	SYN
	7	004	CTRL/D	EOT
12–11				
	none	174	|	vertical bar
	1	152	j	lower-case J
	2	153	k	lower-case K
	3	154	l	lower-case L
	4	155	m	lower-case M
	5	156	n	lower-case N
	6	157	o	lower-case O
	7	160	p	lower-case P
12–0				
	none	173	{	open brace
	1	141	a	lower-case A
	2	142	b	lower-case B
	3	143	c	lower-case C
	4	144	d	lower-case D
	5	145	e	lower-case E
	6	146	f	lower-case F
	7	147	g	lower-case G
12–8				
	none	110	H	upper-case H
(DEC 029)	2	133	[	open square bracket
(DEC 026)		077	?	question mark
	3	056	.	period
(DEC 029)	4	074	<	open angle bracket
(DEC 026)		051	)	close parenthesis
(DEC 029)	5	050	(	open parenthesis
(DEC 026)		135	]	close square bracket
(DEC 029)	6	053	+	plus sign
(DEC 026)		074	<	open angle bracket
	7	041	!	exclamation mark
12–9				
	none	111	I	upper-case I
	1	001	CTRL/A	SOH
	2	002	CTRL/B	STX
	3	003	CTRL/C	ETX
	5	011	CTRL/I	HT
	7	177		DEL
11–0				
	none	175	}	close brace
	1	176	~	tilde
	2	163	s	lower-case S
	3	164	t	lower-case T
	4	165	u	lower-case U
	5	166	v	lower-case V
	6	167	w	lower-case W
	7	170	x	lower-case X
11–8				
	none	121	Q	upper-case Q
(DEC 029)	2	135	]	close square bracket
(DEC 026)		072	:	colon
	3	044	$	dollar sign

(Continued on next page)

Table 10–18: DEC 026/DEC 029 Card Code Conversions (Cont.)

Zone	Digit	Octal	Character	Name
	4	052	*	asterisk
(DEC 029)	5	051	)	close parenthesis
(DEC 026)		133	[	open square bracket
(DEC 029)	6	073	;	semi-colon
(DEC 026)		076	>	close angle bracket
(DEC 029)	7	136	^	uparrow (circumflex)
(DEC 026)		046	&	ampersand
11-9				
	none	122	R	upper-case R
	1	021	CTRL/Q	DC1
	2	022	CTRL/R	DC2
	3	023	CTRL/S	DC3
	6	010	CTRL/H	BS
0-8				
	null	131	Y	upper-case Y
(DEC 029)	2	134	\	backslash
(DEC 026)		073	;	semi-colon
	3	054	,	comma
(DEC 029)	4	045	%	percent sign
(DEC 026)		050	(	open parenthesis
(DEC 029)	5	137	_	backarrow (underscore)
(DEC 026)		042	"	double quote
(DEC 029)	6	076	>	close angle bracket
(DEC 026)		043	#	number sign
(DEC 029)	7	077	?	question mark
(DEC 026)		045	%	percent sign
0-9				
	null	132	Z	upper-case Z
	5	012	CTRL/J	LF
	6	027	CTRL/W	ETB
	7	033		ESC
9-8				
	4	024	CTRL/T	DC4
	5	025	CTRL/U	NAK
	7	032	CTRL/Z	SUB
12-9-8				
	3	013	CTRL/K	VT
	4	014	CTRL/L	FF
	5	015	CTRL/M	CR
	6	016	CTRL/N	SO
	7	017	CTRL/O	SI
11-9-8				
	none	030	CTRL/X	CAN
	1	031	CTRL/Y	EM
	4	034	CTRL/\	FS
	5	035	CTRL/]	GS
	6	036	CTRL/^	RS
	7	037	CTRL/_	US
0-9-8				
	5	005	CTRL/E	ENQ
	6	006	CTRL/F	ACK
	7	007	CTRL/G	BEL

(Continued on next page)

Table 10–18: DEC 026/DEC 029 Card Code Conversions

Zone	Digit	Octal	Character	Name
12-0-8	none	150	h	lower-case H
12-0-9	none	151	i	lower-case I
12-11-8	none	161	q	lower-case Q
12-11-9	none	162	r	lower-case R
11-0-8	none	171	y	lower-case Y
11-0-9	none	172	z	lower-case Z
12-11-9-8				
	1	020	CTRL/P	DLE
12-0-9-8				
	1	000		NUL

10.5 High-Speed Paper Tape Reader and Punch: PC

RT–11 provides support for the PR11 High Speed Reader and the PC11 High Speed Reader/Punch through the PC handler. The PC handler distributed with RT–11 supports both the paper tape reader and punch. A handler supporting only the paper tape reader can be created through system generation. The PC handler does not print an uparrow (^) on the terminal when it is entered for input the first time, as did the PR handler for earlier versions of RT–11. The tape must be in the reader when the command is issued, or an input error occurs. This prohibits any two-pass operations from using PC as an input device. For example, linking and assembling from PC does not work; an input error occurs when the second pass is initiated. The correct procedure is to transfer the paper tape file to disk or DECtape and then perform the operation on the new file.

On input, the PC handler zero-fills the buffer when no more tape is available to read. On the next read request to the PC handler, the EOF bit is set and the carry bit is set on return from the I/O completion.

10.6 Console Terminal Handler: TT

The console terminal can be treated as a peripheral device by using the TT handler. Observe the following conventions and restrictions:

1. An uparrow (^) is typed when the handler is ready for input.
2. CTRL/Z can be used to specify the end of input to TT. No carriage return is required after the CTRL/Z. If CTRL/Z is not typed, the TT handler accepts characters until the word count of the input request is satisfied.
3. CTRL/O, typed while output is directed to TT, causes an entire output buffer (all characters currently queued) to be ignored.
4. A single CTRL/C typed while typing input to TT causes a return to the monitor. If output is directed to TT, a double CTRL/C is required to return to the monitor if FB is running. If the SJ monitor is running, only a single CTRL/C is required to terminate output.

5. The TT handler can be in use for only one job (foreground or background) at a time, and for only one function (input or output) at a time. The terminal communication for the job not using TT is not affected at all.

6. You can type ahead to TT; characters are obtained from the input ring buffer before the keyboard is referenced. The terminating CTRL/Z can also be typed ahead.

7. If the mainline code of a job is using TT for input, and a completion routine issues a .TTYIN request, typed characters are passed unpredictably to the .TTYIN and TT.

8. If a job sends data to TT for output and then issues a .TTYOUT request or a .PRINT request, the output from the latter is delayed until the handler completes its transfer. If a TT output operation is started when the monitor's terminal output ring buffer is not empty (before the print-ahead is complete), the handler completes the transfer operation before the buffer contents are printed.

10.7 RK06/07 Disk Handler: DM

The RK06/07 disk handler has some features that are not standard for most RT–11 handlers. Among these nonstandard features are the following:

1. Support of bad block replacement

2. .SPFUN requests to read and write absolute blocks on disk

3. .SPFUN request to initialize the bad block replacement table

4. .SPFUN error return information

5. .SPFUN request to determine the size of a volume mounted in a particular device unit. (The RK06 and RK07 disks share the same controller and handler. The RK07 has twice as many blocks as the RK06 volume.)

10.7.1 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 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 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.

10.7.2 .SPFUN Requests

The RK06/07 handler accepts the .SPFUN programmed request with the following function codes:

Code	Action
377	Read operation without doing bad block replacement; returns definitive error data.
376	Write operation without doing bad block replacement; returns definitive error data.
374	Re-read the bad block replacement table in the handler (the program changed it).
373	Determine the size, in 256-word blocks, of a particular volume.

The format of the .SPFUN request is the same as explained in the *RT–11 Programmer's Reference Manual*, except as follows: for function codes 377 and 376, the buffer size for reads and writes must be one word larger than required for the data. The first word of the buffer contains the error information returned from the .SPFUN request. This information is returned for a

.SPFUN read or write, and the data transferred follows the error information. The error codes and information are as follows:

Code	Meaning
100000	The I/O operation is successful.
100200	A bad block is detected (BSE error).
100001	An ECC error is corrected.
100002	An error was recovered on a retry.
100004	An error was recovered through an offset retry.
100010	An error was recovered after recalibration.
1774xx	An error was not recovered.

For function code 374, the *buf*, *wcnt*, and *blk* arguments should be 0. For function code 373, *buf* is a one-word buffer where the size of the specified volume in 256-word blocks is returned. The *wcnt* argument should be 1, and the *blk* argument should be 0.

10.8 RL01/02 Disk Handler: DL

The RL01/02 disk handler includes the following special features:

1. .SPFUN requests to read and write absolute blocks on the disk (without invoking the bad block replacement scheme)
2. Support of automatic bad block replacement
3. .SPFUN request to initialize the bad block replacement table
4. .SPFUN request to determine the size of a volume mounted in a particular device unit

The codes for the .SPFUN requests are as follows:

Code	Action
377	Read operation without doing bad block replacement; returns definitive error data.
376	Write operation without doing bad block replacement; returns definitive error data.
374	Re-read the bad block replacement table in the handler (the program changed it).
373	Determine the size, in 256-word blocks, of a particular volume.

Unlike the DM handler, the read and write .SPFUN requests for the DL handler do not return an error status in the first word of the buffer.

Bad block replacement for the RL01 and RL02 is similar to the bad block support for the RK06 and RK07. 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 RL01/02 disk contains a table of the bad sectors that were discovered during manufacture of the disk. The 10 blocks preceding this table (the last 10 blocks in the second-to-last track) are set aside for bad block replacements. The maximum number of bad blocks – 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 RL01/02 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 10–4.

Figure 10–4: Bad Block Replacement Table

BAD BLOCK	12	WORD 0
ITS REPLACEMENT	10210	
BAD BLOCK	37	WORD 2
ITS REPLACEMENT	10211	
BAD BLOCK	553	WORD 4
ITS REPLACEMENT	10212	
END OF LIST	0	WORD 6

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 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:

1. 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.
2. 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.
3. If an .SPFUN 374 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.

10.9 Null Handler: NL

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.

10.10 Dectape II Handler: DD

DECtape II is a random-access mass storage device that uses DECtape II magnetic tape data cartridges. RT–11 supports this device as a file-structured random-access device and as a system device. The following sections describe some general characteristics of DECtape II.

10.10.1 Write-Protect Feature

Each cartridge has a write-protect tab (the word RECORD and an arrow are embossed on the tab). To write enable the cartridge, slide the tab in the direction of the arrow. Slide the tab in the other direction to write protect the cartridge. You can also remove the tab altogether to permanently write protect the cartridge.

10.10.2 Data Storage

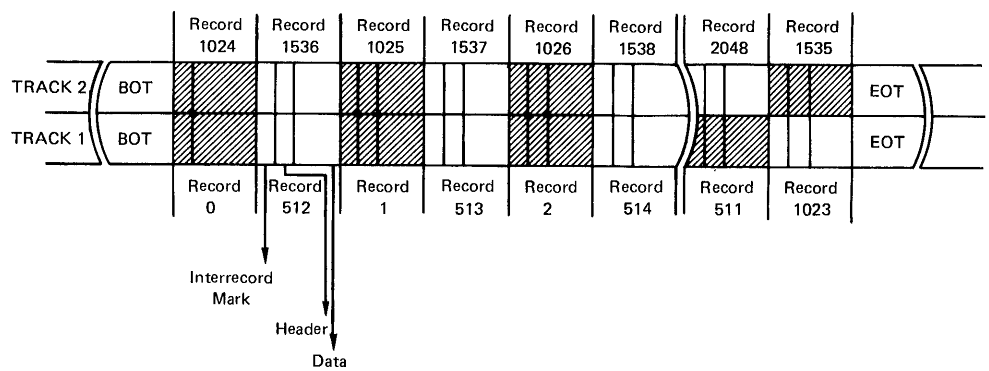
Cartridges have two magnetic tape tracks. DECtape II writes data in the same direction on each track and stores data in data records. It writes data records in a specific sequence and pattern; to write an entire cartridge, for example, it:

1. Writes alternate data records on the first track

2. Rewinds to return to the beginning of tape (BOT) mark
3. Writes data records skipped on the first pass
4. Rewinds
5. Writes alternate data records on the second track
6. Rewinds
7. Writes data records skipped on the first pass of the second track

Figure 10–5 illustrates this interleaved format.

Figure 10–5: DECtape II Tape Format



RT–11 accesses blocks, which on DECtape II consist of four records. Each cartridge stores 512 blocks, each block containing 256 words (64 words per record).

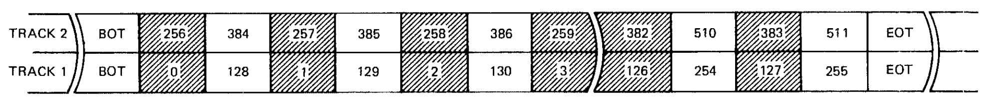
In some circumstances, DECtape II's interleaved tape format may adversely affect performance. If, for instance, the monitor file on a system volume happened to overlap from the end of tape to the beginning of tape, the number of rewinds would increase and, consequently, seek times would increase. Following the suggestions in the next section can help you to avoid such overlap.

10.10.3 Adding Bad Blocks to Avoid Excessive Rewinds

If your system volume is a DECtape II cartridge, you may encounter performance problems (slow response time) due to excessive rewinds of the magnetic tape. You can actually improve system performance by creating dummy bad blocks in strategic locations. Performance degradation occurs when a file (particularly a monitor file) overlaps from the end of tape to the beginning of tape — for example, it extends from the last portion of the second pass on track 1 to the first portion of the first pass on track 2. Slow reponse time results from the specific sequence and pattern in which DECtape II writes data records on the cartridge.

You can avoid this overlap by creating dummy bad blocks in three locations. (Figure 10–6 illustrates the locations of blocks on the tape.) Use DUP to create a bad block at the beginning of the second pass on track 1 (block 128.), at the beginning of the first pass on track 2 (block 256.), and at the beginning of the second pass on track 2 (block 384.). In this way, you can prevent the system from writing across rewinds, since RT–11 requires contiguous free space in which to write files. However, this technique prevents you from creating any file over 127 blocks long, and also increases fragmentation.

Figure 10–6: Bad Block Locations on DECtape II



10.11 MSCP Disk Handler: DU

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 device handler can access any kind of MSCP disk.

Two kinds of disks which use this protocol and are supported by RT–11 Version 5 are the RA80 and the RC25. Each RA80 disk contains about 237,200 blocks. The RC25 is a two-disk system; each disk contains about 49,350 blocks. The two disks of an RC25 are always assigned sequential MSCP unit numbers: the first disk, which is removable, always has an even unit number (n); the second disk, which is fixed, always has an odd unit number (n + 1).

10.11.1 Addressing an MSCP Disk

You identify an MSCP disk to the DU handler by specifying:

1. The MSCP unit number, in the range 0 through 253;
2. The controller port number, in the range 0 through 3;
3. The disk partition number, in the range 0 through 255.

During normal operation, 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. 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.

10.11.1.1 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.

10.11.1.2 Controller Port Numbers — The controller port number provides a way of logically identifying a particular vector/CSR pair of a multi-vector unibus disk adapter (UDA) or Q-bus disk adapter (QDA). In the past, a single disk controller has been on a single board. Now, technology allows multiple controllers — termed multi-port controllers — on a single board. Each port is addressed by its own vector and CSR. Although all the posts reside on the same board and may in fact share components, each is logically separate and operates independently. (DIGITAL does not currently offer a multi-port MSCP controller for PDP–11s, but the DU handler contains the code to support multiple ports in case multi-port controllers for PDP–11s are available in the future.)

You can access a multi-port UDA or QDA through the DU handler in one of two ways. One way is to copy the handler to another file name. 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. You can then use the original DU handler to access disks connected through the first port, and the new copy of the handler to access disks connected through the second 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 is 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
```

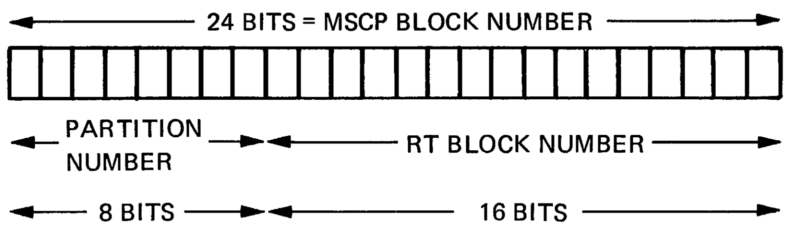
10.11.1.3 Disk Partition Numbers — Disk partition numbers allow RT–11 to use disks having more than 65,535 blocks. The disk partition number can be thought of as a high-order block number, as shown in Figure 10–7.

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 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.

1. Although RT–11 block numbers can be 0 through 177777(octal), or a total of 65,536 decimal blocks (200000 octal, or 000000 in 16 bits since the 17th bit is lost), the size of a partition is defined as 65,535 decimal blocks (177777 octal), 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

Figure 10–7: MSCP Disk Block Number



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 10.11.2 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
```

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).

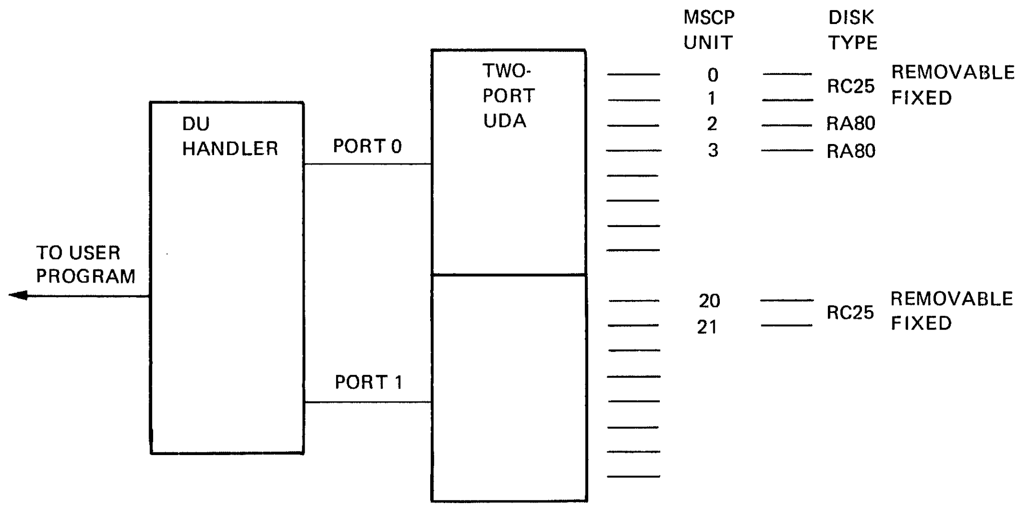
Although most RT–11 systems may never have more than one or two MSCP disks, an example using several disks may help to clarify these concepts. Consider the example of a two-port UDA controller interfaced to six disks, shown in Figure 10–8.

The user of the system illustrated issues the following SET command:

```
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

Figure 10–8: Two-Port DU Handler



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.

10.11.2 .SPFUN Requests

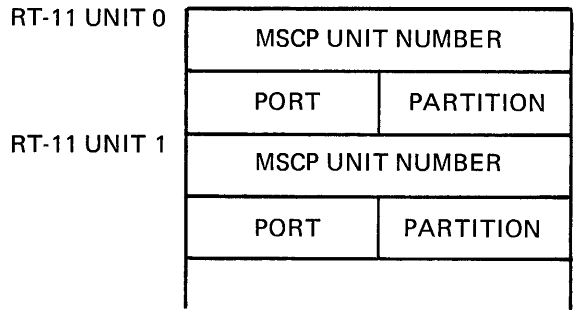
The DU handler supports .SPFUN requests that return the volume size of the particular disk being used (code 373), return the RT-to-MSCP translation table (code 372), and provide direct access to all MSCP features (code 371). For further information, see the hardware manual appropriate for your disk; for example; the UDA50 Programmer's Documentation Kit (Order No. QP–905–GZ).

.SPFUN with code 373 returns the volume size in the word pointed to by the *buf* argument of the .SPFUN call. This function is identical to that provided in the DL, DM, DY, and LD handlers.

.SPFUN with code 372 returns, beginning at the address pointed to by the *buf* argument of the .SPFUN call, a 16-word table consisting of eight 2-word entries. These define the correspondence between RT–11 unit numbers DU0: through DU7: and MSCP unit numbers, ports, and partitions. The format of the table is given in Figure 10–9.

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.

Figure 10–9: DU Handler Translation Table



.SPFUN with code 371 bypasses all automatic unit number translation and allows direct access to the MSCP port. The buffer address in the .SPFUN request must point to a 52-word area in the user's job. The first 26 words are used to hold:

1. The response packet length (in bytes);
2. The virtual circuit identifier;
3. The end packet when the command is complete.

The second 26 words must be set up by the caller to contain a length word, a virtual circuit identifier, and a valid MSCP command. Except for port initialization, the user program must do all command packet sequencing, error handling, and reinitialization when the bypass operations are complete. In XM, the 52-word control block must reside in low memory. 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 Bufffer (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

10.12 Virtual Memory Handler: VM

The Virtual Memory handler (VM) allows extended memory to be accessed as though it were a block-replacable disk. In general, you use VM just like a disk handler. You use the INITIALIZE command to initialize the extended memory reserved for the VM area before copying files to the area, and you can get a directory of the files in the VM area using the DIR command. You can even copy an SJ or FB monitor and bootstrap to VM and run an SJ or FB system from it.

The VM handler supported in RT–11 Version 5 is not an adaptation of previous VM handlers available through DECUS or other sources. It has been rewritten; only the concept is the same. Features of the Version 5 handler include:

- Memory sizing at handler installation time, not handler load time.
- A base address that can be set to any extended memory address. The default base address for the SJ and FB version of the VM handler is at the 28K-word boundary, the low limit of extended memory; the default base address for the XM version of the VM handler is at the 128K-word boundary.
- Support for VM under the XM monitor as well as the SJ and FM monitors. The XM monitor supports VM as a data volume, but not as a system volume.

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 change the amount of memory available to VM by changing the base address, you must remove and reinstall the handler. Most other handlers require only that you reload the handler after a SET command, not reinstall it. You must also reinitialize VM after changing the base address by using the INIT VM: command. RT–11 prints a warning, *VM-W-Remove and reinstall*, whenever you change the base address by issuing the SET VM BASE = n command.

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.

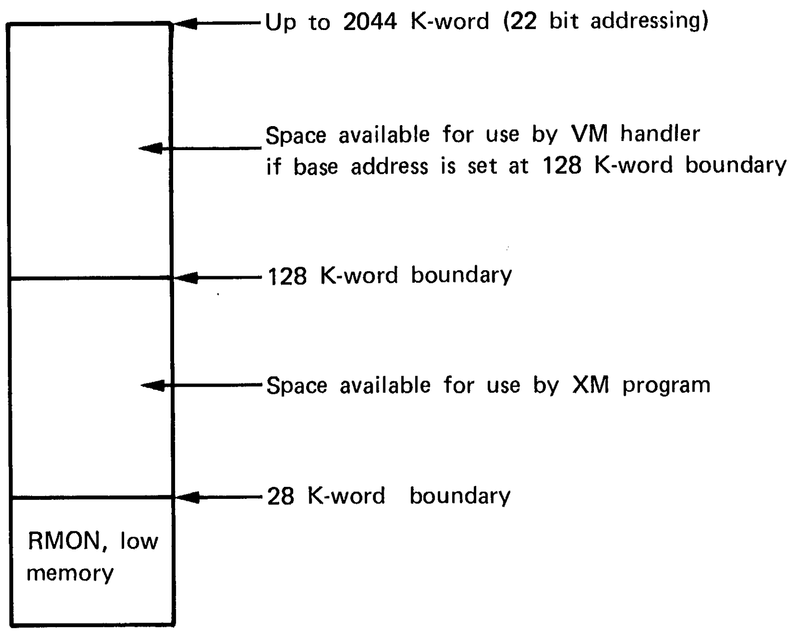
The base address (n) used in the SET VM BASE = n command is the desired base address in octal, divided by 100(octal). For example, use the value 1600 to set the base address at the 28K-word address boundary, or 10000 to set the base address at the 128K-word address boundary; any other value between 1600 and the physical memory high limit is also acceptable. The list below gives 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

If you have a 22-bit system and use the XM monitor's VM handler set to its default base address, Version 4 programs that use the XM monitor will run without modification. Since the default base address of the XM version of the VM handler is at the 128K-word boundary, all memory previously available to an XM program will still be available. You can use extended memory as you did in Version 4 and use any extra memory above 128K words as a VM volume. Figure 10–10 shows a 22-bit system with a VM base address of 10000 (128K words).

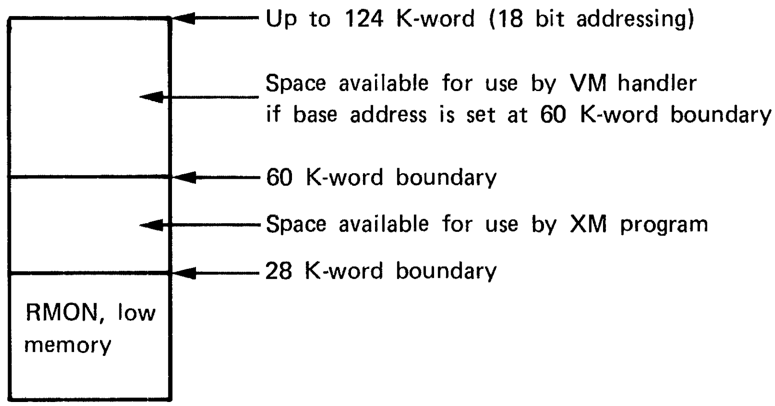
Figure 10–10: VM Handler in a 22-bit System



If you are using the XM 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 XM system. You can still use extended memory for both an XM program and a VM volume, but the space available for one will be reduced by the space occupied by the other. Refer to Figure 10–11, showing an 18-bit system with the VM base address set to 3600 (60K words).

Many DMA devices – for example, RK05s – cannot handle more than an 18-bit address (128K words). If you have one of these devices and have more than 128K words of memory on your system, setting the VM base at the 128K-word boundary ensures that none of your programs will create a data buffer that your disk cannot read or write, while allowing effective use of the memory above 128K words through the VM handler.

Figure 10–11: VM Handler in an 18-bit System



10.13 Logical Disk Handler: LD

The Logical Disk handler implements logical disk support in RT–11 Version 5. The LD handler accepts I/O requests just as does any other disk handler. By means of imbedded 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.

10.13.1 LD Translation Tables

There are four translation tables within the LD handler. The first translation table starts at the label HANDLR and contains one word for each LD unit number, up to a maximum of eight. This table is indexed by the LD unit number times 2.

Bits 0 through 5 of each word in the HANDLR table contain an index into the the handler tables in RMON for the physical device corresponding to the LD unit number.

Bit 6 is a flag to signal that the entry in the OFFSET table for the LD unit may be incorrect. This bit is set whenever a volume is squeezed. The LD handler, when it uses an LD unit, checks this bit; if the bit is set, the handler verifies the unit's OFFSET table entry before proceeding.

Bit 7 is a flag to signal that the index entry, bits 0 through 5, may be incorrect and should be checked and updated. The LD handler sets bit 7 for all units if, upon entry, the LDREL$ bit in CONFG2 is set.

Bits 8 through 10 contain the unit number of the physical disk assigned to the logical disk unit.

Bits 11, 12, and 14 are unused.

Bit 13 is the write-lock bit. If set, the LD unit is read-only.

Bit 15 is the allocation bit. If set, the LD unit is assigned; if 0, the LD unit has not been assigned.

The second translation table starts at the label OFFSET and contains one word for each LD unit number, up to a maximum of eight. This table is indexed by LD unit number times 2. Each word contains the offset in blocks from the beginning of the assigned physical disk to the start of the area on the physical disk assigned to that LD unit number.

The third translation table starts at the label SIZE and contains one word for each LD unit number, up to a maximum of eight. This table is indexed by LD unit number times 2. Each word contains the size in blocks of the area on physical disk assigned to the logical disk unit.

The fourth translation table starts at the label NAME and contains four words for each LD unit number. This table is indexed by LD unit number times 8. The first word of each four-word entry contains the Radix–50 two-character name of the physical disk assigned to the logical disk unit. This word must be the actual device name, not a logical assignment name, and it must not have a unit number as part of the name. The second, third, and fourth words of each four-word entry contain the Radix–50 filename and file type of the file assigned as the logical disk.

10.13.2 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.

10.13.3 Special LD Option: /$

When you issue a MOUNT keyboard monitor command to assign a logical disk unit, MOUNT builds the command line to assign the logical disk unit, places the command in the chain area, and chains to LD.SYS. MOUNT includes in the command line the /L option, which specifies logical unit assignment. MOUNT also inserts the /$ option.

LD.SYS receives the command line and uses the /L option code to assign the logical disk unit. Then, if the /$ option was included in the command line, LD checks to see if the device handler required by that logical unit is in memory. If the device handler is not loaded, LD builds a LOAD command for that handler, places the command in the chain area, and passes the command back to KMON for execution. Thus, any handlers required for logical units assigned by the MOUNT command are automatically loaded. Although the /$ option is primarily for use by MOUNT, you can use /$ with /L if you run LD.SYS yourself to assign logical units.

Appendix A
RK, DX, and PC Device Handlers

This appendix contains listings of the RK, DX, and PC device handlers. Figure A–1 shows the RK (RK05 disk) handler, which is relatively straightforward; Figure A–2 shows the DX (single-density diskette) handler, which is more complex than RK since it involves intricate calculations and uses a silo (a special buffer in the device itself); Figure A–3 shows the PC (paper-tape contraption) handler, a handler for a character-oriented device. For information on writing a device handler, read Chapter 7.

The listings in this appendix were produced by assembling the conditional file CND.MAC together with the handler source files. The handler files were edited slightly before assembly so that all comments would fit on an 80-character line. The command strings to produce these assemblies and listing files are as follows:

Keyboard monitor command:

```
MACRO/LIST:dd.LST/NOOBJECT/SHOW:ME:TTM CND+dd
```

MACRO program commands:

```
R MACRO
,dd/L:ME:TTM=CND,dd
```

dd represents the two-character device handler name.

In all listings, comments that are part of the source file consist of upper-case characters; each line begins with a semicolon (;). The source file text is shown in dot-matrix type face. Explanatory comments that were added as documentation in this appendix are upper- and lower-case characters, and are printed in regular type face.

The file CND.MAC was created especially for these examples. It was assembled together with the handler source files to produce code for the three system generation conditions: memory management, error logging, and device time-out. Normally, you assemble a device handler file with the system conditional file, SYCND.MAC, to ensure that the handler has the same system generation parameters as the current monitor.

Figure A–1: RK Disk Handler

```
RK - RK05(RK11) DISK HANDLER    MACRO Y05.00b Monday 04-Oct-82 12:41
Table of contents

    2-   3      COPYRIGHT NOTICE
    4-   1      MACROS AND DEFINITIONS
    5-   1      DRIVER EDIT LEVEL
    6-   1      SET OPTIONS
    7-   1      DRIVER ENTRY
    8-   1      INTERRUPT ENTRY POINT
    9-   1      BOOTSTRAP DRIVER

     1                  ;CONDITIONAL FILE FOR HANDLER EXAMPLES
     2                  ;
     3                  ;CND.MAC
     4                  ;
     5                  ;SGW
     6                  ;
     7                  ;ASSEMBLE WITH HANDLER .MAC FILE TO ENABLE
     8                  ;18-BIT I/O, TIME-OUT, AND ERROR LOGGING
     9                  ;FOR HANDLER.
    10                  ;
    11        000001    MMG$T   = 1             ;18-BIT I/O
    12        000001    ERL$G   = 1             ;ERROR LOGGING
    13        000001    TIM$IT  = 1             ;TIME-OUT
     1                          .TITLE  RK - RK05(RK11) DISK HANDLER
     2                          .IDENT  /X05.02/
     3                          .SBTTL COPYRIGHT NOTICE
     4
     5                  ;              COPYRIGHT (c) 1982, 1983 BY
     6                  ;     DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
     7                  ;                 ALL RIGHTS RESERVED.
     8                  ;
     9                  ; THIS  SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE
    10                  ; USED   AND  COPIED   ONLY  IN  ACCORDANCE   WITH   THE
    11                  ; TERMS  OF  SUCH  LICENSE  AND  WITH  THE  INCLUSION OF
    12                  ; THE  ABOVE  COPYRIGHT  NOTICE.  THIS  SOFTWARE OR  ANY
    13                  ; OTHER COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE
    14                  ; MADE  AVAILABLE  TO ANY OTHER PERSON.  NO TITLE TO AND
    15                  ; OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
    16                  ;
    17                  ; THE INFORMATION  IN THIS SOFTWARE IS SUBJECT TO CHANGE
    18                  ; WITHOUT  NOTICE  AND   SHOULD   NOT  BE  CONSTRUED  AS
    19                  ; A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
    20                  ;
    21                  ; DIGITAL  ASSUMES  NO  RESPONSIBILITY  FOR  THE  USE OR
    22                  ; RELIABILITY OF ITS SOFTWARE ON EQUIPMENT WHICH  IS NOT
    23                  ; SUPPLIED BY DIGITAL.
     1                  .SBTTL  MACROS AND DEFINITIONS
     2
```

Preamble Section

Each macro you use in the handler requires the .MCALL statement, as line 3 shows. Since .DRDEF issues most of the .MCALL statements for you, you need issue only the .MCALL for .DRDEF, and for .SYNCH if you plan to use it.

```
     3                  .MCALL   .DRDEF
     4
```

The .DRDEF macro performs most of the work of the preamble section:

```
     5 000000           .DRDEF   RK,0,FILST$,4800.,177400,220
```

The .DRDEF macro generates the .MCALL statements for the other system macros:

```
                        .MCALL   .DRAST,.DRBEG,.DRBOT,.DREND,.DRFIN,.DRSET,
                        .MCALL   .DRVTB,.FORK,.QELDF
```

The code in this handler contains many conditional assembly directives. They test for the presence or absence of time-out support, extended memory support, and error logging. Code is generated differently depending on which of those system generation features are present in the system. If there is no conditional file assembled with the handler file, the conditionals are turned off by the following lines of code. For this example, the three following conditionals were set to 1 by the file CND.MAC.

```
                .IIF NDF RTE$M, RTE$M=0
                .IIF NE RTE$M, RTE$M=1
                .IIF NDF TIM$IT, TIM$IT=0
        000001  .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
        000001  .IIF NE ERL$G, ERL$G=1
```

If device time-out support is part of your system, the .DRDEF macro also issues the .MCALL statement for the .TIMIO and .CTIMIO macros:

```
                .IIF NE TIM$IT, .MCALL  .TIMIO,.CTIMI
```

The .DRDEF macro invokes the .QELDF macro to define queue element offsets and convenient symbols:

```
        000000  .QELDF
                .IIF NDF MMG$T, MMG$T=0
        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$'X=Q.'X-4
                .ENDR
        177774  Q$LINK=Q.LINK-4
        177776  Q$CSW=Q.CSW-4
        000000  Q$BLKN=Q.BLKN-4
        000002  Q$FUNC=Q.FUNC-4
        000003  Q$JNUM=Q.JNUM-4
        000003  Q$UNIT=Q.UNIT-4
        000004  Q$BUFF=Q.BUFF-4
        000006  Q$WCNT=Q.WCNT-4
        000010  Q$COMP=Q.COMP-4
                .IF EQ MMG$T
                Q.ELGH=^O16
                .IFF
        000016  Q.PAR=^O16
        000012  Q$PAR=^O12
        000024  Q.ELGH=^O24
                .ENDC
        000001  HDERR$=1
        020000  EOF$=20000
        000400  VARSZ$=400
        001000  ABTIO$=1000
        002000  SPFUN$=2000
        004000  HNDLR$=4000
        010000  SPECL$=10000
        020000  WONLY$=20000
        040000  RONLY$=40000
        100000  FILST$=100000
```

The size of the device, its device-identifier byte, and its status word are defined:

```
011300  RKDSIZ=4800.
000000  RK$COD=0
100000  RKSTS=<0>!<FILST$>
```

The default CSR and vector are defined:

```
        .IIF NDF RK$CSR, RK$CSR=177400
        .IIF NDF RK$VEC, RK$VEC=220
        .GLOBL  RK$CSR,RK$VEC
```

The following direct assignment statements set up names for the device control registers. The register names, locations, and operation codes can be found in the *PDP–11 Peripherals Handbook* and in the hardware manual for the RK disk.

```
177400  RKDS    = RK$CSR                 ;DRIVE STATUS REGISTER
177402  RKER    = RKDS+2                 ;ERROR REGISTER
177404  RKCS    = RKDS+4                 ;CONTROL & STATUS
                                         ;REGISTER
177406  RKWC    = RKDS+6                 ;WORD COUNT
177410  RKBA    = RKDS+10                ;BUS ADDRESS
177412  RKDA    = RKDS+12                ;DISK ADDRESS
```

The symbol RKCNT represents the number of times to retry an I/O transfer should an error occur.

```
000010  RKCNT   = 8.                     ;NUMBER OF ERROR RETRYS
```

The symbol RKNREG represents the number of device registers to record in the error log.

```
000007  RKNREG  = 7                      ;NUMBER OF CSR'S TO
                                         ;READ FOR ERROR LOG
```

The following symbols define bits in the registers:

```
        ; BITS IN DRIVE STATUS REGISTER (RKDS)

160000  DSID    = 160000                 ;ID OF INTERRUPTING
                                         ;DRIVE (MASK)
010000  DSDPL   =  10000                 ;DRIVE POWER LOW
004000  DSRK05  =   4000                 ;SET => DRIVE IS RK05
002000  DSDRU   =   2000                 ;DRIVE UNSAFE
001000  DSSIN   =   1000                 ;SEEK INCOMPLETE
000400  DSSOK   =    400                 ;SECTOR COUNTER OK
000200  DSDRY   =    200                 ;DRIVE READY
000100  DSREDY  =    100                 ;READ/WRITE/SEEK READY
000040  DSWPS   =     40                 ;WRITE PROTECT STATUS
000020  DSSCOK  =     20                 ;SECTOR COUNTER = SECTOR
                                         ;ADDRESS
000017  DSSC    =     17                 ;SECTOR COUNTER MASK
                                         ;(LOOK AHEAD)

        ; BITS IN ERROR REGISTER (RKER)

100000  ERDRE   = 100000                 ;DRIVE ERROR
040000  EROVR   =  40000                 ;OVERRUN
```

```
           020000  ERWLO   =  20000                ;WRITE LOCK OUT VIOLATION
           010000  ERSKE   =  10000                ;SEEK ERROR
           004000  ERPGE   =   4000                ;PROGRAMMING ERROR
           002000  ERNXM   =   2000                ;NON-EXISTENT MEMORY
           001000  ERDLT   =   1000                ;DATA LATE
           000400  ERTE    =    400                ;TIMING ERROR
           000200  ERNXD   =    200                ;NON-EXISTENT DISK
           000100  ERNXC   =    100                ;NON-EXISTENT CYLINDER
           000040  ERNXS   =     40                ;NON-EXISTENT SECTOR
           000002  ERCSE   =      2                ;CHECKSUM ERROR
           000001  ERWCK   =      1                ;WRITE CHECK ERROR

                   ; BITS IN CONTROL AND STATUS REGISTER (RKCS)

           100000  CSERR   = 100000                ;ERROR
           040000  CSHE    =  40000                ;HARD ERROR
           020000  CSSCP   =  20000                ;SEARCH COMPLETE
           004000  CSINHB  =   4000                ;INHIBIT BUS ADDRESS
                                                   ;INCREMENT
           002000  CSFMT   =   2000                ;FORMAT
           000400  CSSSE   =    400                ;STOP ON SOFT ERROR
           000200  CSRDY   =    200                ;CONTROL READY
           000100  CSIE    =    100                ;INTERRUPT ENABLE
           000060  CSBA67  =     60                ;BUS ADDRESS BITS 16-17
           000020  CSBA16  =     20                ;BUS ADDRESS BIT 16
           000016  CSFUN   =     16                ;FUNCTION CODE
           000001  CSGO    =      1                ;GO BIT
```

These are the operation codes:

```
                   ; FUNCTION CODES IN CSFUN

           000000  FNRST   = 0*2                   ;CONTROL RESET
           000002  FNWRITE = 1*2                   ;WRITE
           000004  FNREAD  = 2*2                   ;READ
           000006  FNWCHK  = 3*2                   ;WRITE CHECK
           000010  FNSEEK  = 4*2                   ;SEEK
           000012  FNRCHK  = 5*2                   ;READ CHECK
           000014  FNDRST  = 6*2                   ;DRIVE RESET
           000016  FNWLK   = 7*2                   ;WRITE LOCK

                   ; BITS IN DISK ADDRESS REGISTER

           160000  DAUNIT  = 160000                ;DRIVE SELECT BITS
           017740  DACYL   =  17740                ;CYLINDER BITS
           000020  DASUR   =     20                ;SURFACE
           000017  DASC    =     17                ;SECTOR BITS
```

The parameter tables for the SET options are generated by the .DRSET macro:

```
                           .SBTTL  SET OPTIONS

  000112                   .DRSET  CSR,    160000, O.CSR,  OCT
  000112           .ASECT
                   .IF LE  .-400
          000400   .=400
                   .IFF
                   .=.-2
                   .ENDC
  000400  160000           160000
          000402   ...V2=.
  000402  012712           .RAD50  CSR
          000406   .=...V2+4
  000406     021           .BYTE   <O.CSR-400>/2
          000000   ...V2=0
                   .IRP X,<OCT>
                   .IF IDN <X>,<NUM>
                   ...V2=...V2!100
                   .IFF
                   .IF IDN <X>,<NO>
                   ...V2=...V2!200
                   .IFF
```

```
                        .IF IDN <X>,<OCT>
                        ...V2=...V2!140
                        .IFF
                        .ERROR  ;ILLEGAL PARAMETER X
                        .ENDC
                        .ENDC
                        .ENDC
                        .ENDR
                        .IF IDN <OCT>,<NUM>
                        ...V2=...V2!100
                        .IFF
                        .IF IDN <OCT>,<NO>
                        ...V2=...V2!200
                        .IFF
                        .IF IDN <OCT>,<OCT>
              000140    ...V2=...V2!140
                        .IFF
                        .ERROR  ;ILLEGAL PARAMETER OCT
                        .ENDC
                        .ENDC
                        .ENDC
     000407      140            .BYTE   ...V2
     000410   000000            .WORD   0
     000412                     .DRSET  VECTOR, 500,    O.VEC,  OCT
     000412             .ASECT
                        .IF LE  .-400
                        .=400
                        .IFF
              000410    .=.-2
                        .ENDC
     000410   000500            500
              000412    ...V2=.
     000412   105113            .RAD50  VECTOR
     000414   077552
              000416    .=...V2+4
     000416      071            .BYTE   <O.VEC-400>/2
              000000    ...V2=0
                        .IRP X,<OCT>
                        .IF IDN <X>,<NUM>
                        ...V2=...V2!100
                        .IFF
                        .IF IDN <X>,<NO>
                        ...V2=...V2!200
                        .IFF
                        .IF IDN <X>,<OCT>
                        ...V2=...V2!140
                        .IFF
                        .ERROR  ;ILLEGAL PARAMETER X
                        .ENDC
                        .ENDC
                        .ENDC
                        .ENDR
                        .IF IDN <OCT>,<NUM>
                        ...V2=...V2!100
                        .IFF
                        .IF IDN <OCT>,<NO>
                        ...V2=...V2!200
                        .IFF
                        .IF IDN <OCT>,<OCT>
              000140    ...V2=...V2!140
                        .IFF
                        .ERROR  ;ILLEGAL PARAMETER OCT
                        .ENDC
                        .ENDC
                        .ENDC
     000417      140            .BYTE   ...V2
     000420   000000            .WORD   0

     000422                     .DRSET  RETRY,  RKCNT,  O.RTRY, NUM
     000422             .ASECT
                        .IF LE  .-400
                        .=400
                        .IFF
              000420    .=.-2
                        .ENDC
     000420   000010            RKCNT
              000422    ...V2=.
     000422   070534            .RAD50  RETRY
     000424   072150
              000426    .=...V2+4
```

```
000426     101              .BYTE   <O.RTRY-400>/2
         000000     ...V2=0
                    .IRP X,<NUM>
                    .IF IDN <X>,<NUM>
                    ...V2=...V2!100

                    .IFF
                    .IF IDN <X>,<NO>
                    ...V2=...V2!200
                    .IFF
                    .IF IDN <X>,<OCT>
                    ...V2=...V2!140
                    .IFF
                    .ERROR  ;ILLEGAL PARAMETER X
                    .ENDC
                    .ENDC
                    .ENDC
                    .ENDR
                    .IF IDN <NUM>,<NUM>
         000100     ...V2=...V2!100
                    .IFF
                    .IF IDN <NUM>,<NO>
                    ...V2=...V2!200
                    .IFF
                    .IF IDN <NUM>,<OCT>
                    ...V2=...V2!140
                    .IFF
                    .ERROR  ;ILLEGAL PARAMETER NUM
                    .ENDC
                    .ENDC
                    .ENDC
000427     100              .BYTE   ...V2
000430  000000              .WORD   0

                            .IF NE ERL$G
000432                      .DRSET  SUCCES, -1,      O.SUCC, NO
000432              .ASECT
                    .IF LE  .-400
                    .=400
                    .IFF
         000430     .=.-2
                    .ENDC
000430  177777              -1
         000432     ...V2=.
000432  075013              .RAD50  SUCCES
000434  011633
         000436     .=...V2+4
000436     107              .BYTE   <O.SUCC-400>/2
         000000     ...V2=0
                    .IRP X,<NO>
                    .IF IDN <X>,<NUM>
                    ...V2=...V2!100
                    .IFF
                    .IF IDN <X>,<NO>
                    ...V2=...V2!200
                    .IFF
                    .IF IDN <X>,<OCT>
                    ...V2=...V2!140
                    .IFF
                    .ERROR  ;ILLEGAL PARAMETER X
                    .ENDC
                    .ENDC
                    .ENDC
                    .ENDR

                    .IF IDN <NO>,<NUM>
                    ...V2=...V2!100
                    .IFF
                    .IF IDN <NO>,<NO>
         000200     ...V2=...V2!200
                    .IFF
                    .IF IDN <NO>,<OCT>
                    ...V2=...V2!140
                    .IFF
                    .ERROR  ;ILLEGAL PARAMETER NO
                    .ENDC
                    .ENDC
                    .ENDC
000437     200              .BYTE   ...V2
000440  000000              .WORD   0
                            .ENDC

```

After defining the SET option tables, write the code to process each SET option:

```
         002410           BTCSR    = <RKEND-RKSTRT>+<BOTCSR-RKBOOT>+1000

000442   020003   O.CSR:  CMP      R0,R3              ;IS CSR IN RANGE?
                                                      ;(>160000)
000444   103444           BLO      O.BAD              ;NOPE...
000446   010067           MOV      R0,176             ;YES, INSTALLATION
         177524
                                                      ;CODE NEEDS IT

                  ; NOW WE ENSURE THAT THE BOOTSTRAP REFLECTS ANY
                  ; CHANGE IN THE CSR

000452   010701           MOV      PC,R1              ;R1->READ/WRITE EMT
                                                      ;AREA
000454   062701           ADD      #BAREA-.+4,R1      ;(BUFFER ADDRESS WORD)
         000160
000460   010702           MOV      PC,R2              ;R2->BUFFER FOR
                                                      ;READ/WRITE
000462   062702           ADD      #1000-.,R2         ;(OVERWRITES CORE COPY
         000316
                                                      ;OF BLOCK 1)
000466   010211           MOV      R2,(R1)            ;SET BUFFER ADDRESS,
000470   012741           MOV      #BTCSR/1000,-(R1)  ;BOOT BLOCK TO
         000002
                                                      ;READ/WRITE
000474   005741           TST      -(R1)              ;R1->EMT AREA
000476   010003           MOV      R0,R3              ;SAVE CSR ELSEWHERE,
                                                      ;EMT NEEDS R0
000500   062703           ADD      #RKDA-RKDS,R3      ;(MUST POINT TO RKDA)
         000012
000504   010100           MOV      R1,R0              ;R0->EMT AREA FOR READ
000506   104375           EMT      375                ; *** (.READW) ***
000510   103422           BCS      O.BAD
000512   010362           MOV      R3,<BTCSR&777>(R2) ;SET THE BOOTSTRAP
         000410
                                                      ;CSR (RKDA)
000516   010100           MOV      R1,R0              ;R0->EMT AREA FOR WRITE
000520   105260           INCB     1(R0)              ;CHANGE FROM 'READ'
         000001
                                                      ;TO 'WRITE'
000524   104375           EMT      375                ; *** (.WRITW) ***
000526   103413           BCS      O.BAD
000530   010100           MOV      R1,R0              ;R0->EMT AREA FOR READ
                                                      ;(LAST TIME)
000532   105360           DECB     1(R0)              ;CHANGE BACK TO 'READ'
         000001
000536   012760           MOV      #1,2(R0)           ;OF HANDLER BLOCK 1
         000001
         000002
000544   104375           EMT      375                ; *** (.READW) ***
000546   103403           BCS      O.BAD

                  ; Now we update the handler CSR. We do it now because
                  ; the area that contains block 1 is used as a buffer
                  ; for reading, updating and rewriting the boot block.

000550   010367           MOV      R3,RKCSR           ;SET THE HANDLER CSR
         000124'
                                                      ;(RKDA)
000554   005727   O.GOOD: TST      (PC)+              ;GOOD RETURN (CARRY
                                                      ;CLEAR)
000556   000261   O.BAD:  SEC                         ;ERROR RETURN (CARRY
                                                      ;SET)
000560   000207           RTS      PC

000562   020003   O.VEC:  CMP      R0,R3              ;IS VECTOR IN RANGE?
                                                      ;(<500)
000564   103374           BHIS     O.BAD              ;NOPE...
000566   032700           BIT      #3,R0              ;IS IT ON A VECTOR
         000003
                                                      ;BOUNDRY?
000572   001371           BNE      O.BAD              ;NOPE...
```

```
   000574  010067           MOV     R0,RKSTRT       ;YES, PLACE IN ENTRY
           000000'
                                                    ;AREA
   000600  000765           BR      O.GOOD

   000602  020003  O.RTRY:  CMP     R0,R3           ;ASKING FOR TOO MANY?
   000604  101364           BHI     O.BAD           ;YES...
   000606  010067           MOV     R0,DRETRY       ;LOOKS REASONABLE,
           000014'
                                                    ;TRY IT
   000612  001360           BNE     O.GOOD          ;ITS OKAY...
   000614  000760           BR      O.BAD           ;CAN'T ASK FOR NO
                                                    ;RETRIES

                            .IF NE ERL$G
   000616  012703  O.SUCC:  MOV     #0,R3           ;'SUCCESS' ENTRY POINT
           000000
                                                    ; (MUST BE TWO WORDS)
   000622  010367           MOV     R3,SCSFLG       ;'NOSUCCESS' ENTRY POINT
           000202'
   000626  000752           BR      O.GOOD
                            .ENDC

   000630     017  BAREA:   .BYTE   17,10           ;CHANNEL 17, READ
   000631     010
   000632                   .BLKW                   ;BLOCK NUMBER
   000634                   .BLKW                   ;BUFFER ADDRESS
   000636  000400           .WORD   256.            ;WORD COUNT
   000640  000000           .WORD   0               ;COMPLETION (WAIT)

```

At the end of the SET option code, check to make sure that the code does not exceed one block in size:

```
                .IIF GT,<.-1000> .ERROR ;SET CODE IS TOO LARGE
```

Header Section

```
                .SBTTL  DRIVER ENTRY

```

The .DRBEG macro:

```
  000000                .DRBEG  RK
```

The following lines are generated by the .DRBEG macro:

```
000000          .ASECT
        000052  . = 52
                .GLOBL  RKEND,RKINT
000052  000546          .WORD   <RKEND-RKSTRT>
                .IF B   <>
000054  011300          .WORD   RKDSIZE
                .IFF
                        .WORD
                .ENDC
                .IF B   <>
000056  100000          .WORD   RKSTS
                .IFF
                        .WORD
                .ENDC
000060  000007          .WORD   ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>
        000176  . = 176
000176  177400  .IIF DF RK$CSR, .WORD   RK$CSR
000000          .PSECT  RKDVR
```

```
000000          RKSTRT::
                .IF NB
                .GLOBL
                        .WORD   <-.>/2. -1 + ^O100000
                .IFF
                .IF NB  <>
                .IIF NE &3      .ERROR          ;ODD OR ILLEGAL VECTOR
                        .WORD   &^C3
                .IFF
                .IF DF  RK$VTB
                .GLOBL  RK$VTB
                        .WORD   <RK$VTB-.>/2. -1 + ^O100000
                .IFF
                .IIF NE RK$VEC&3 .ERROR RK$VEC ;ODD OR ILLEGAL VECTOR
```

The first word of the handler is RK$VEC:

```
000000  000220          .WORD   RK$VEC&^C3
                .ENDC
                .ENDC
                .ENDC
```

The second word of the handler is the self-relative byte offset to the interrupt entry point RKINT. It is also used by the monitor abort I/O request code to find the abort entry point of the handler (it is the word immediately preceding the interrupt entry point).

The third word of the handler contains the PS to be inserted into the device vector. The high byte must be 0. The low byte should be 340, for priority 7. If the low byte is lower than 340, the .FETCH code forces it to the actual new PS in the vector in order to specify priority 7. The condition bits can be used to distinguish up to 16 different interrupts or controllers. They are copied in to the PS word of the vector and set in the PS when the device interrupts using that vector.

```
000002  000174          .WORD   RKINT-.,^O340
000004  000340
000006          RKSYS::
```

The address of the fourth word of the handler, RKLQE, is placed in the monitor $ENTRY table. RKLQE points to the last queue element in the queue for this handler, thus making it easier for the monitor to add elements to the end of the queue. If there are no more elements in the queue, this word is 0.

```
000006  000000  RKLQE:: .WORD   0
```

The fifth word of the handler, RKCQE, points to the third word, Q.BLKN, of the current queue element. If there is no current queue element, RKCQE is 0.

```
000010  000000  RKCQE:: .WORD   0
```

I/O Initiation Section

The next statement is the first executable statement of the handler. This point is reached after a .READ or .WRITE programmed request is issued in a program. The monitor queue manager calls the handler with a JSR PC instruction at the sixth word whenever a new queue element becomes the first element in the handler's queue. This situation occurs when an element is added to an empty queue, or when an element becomes first in the queue because a previous element was released. This section starts the I/O transfer. The I/O initiation code is executed at priority 0 in system state. All registers are available to use in this section. At the end of the section, control is returned to the monitor with an RTS PC instruction.

The MOV instruction sets the number of error retries to 8 and moves that value to RETRY. (The (PC)+ notation points to RETRY.) At this point the handler has a brand new queue element and no retry is in progress. (If bit 15 of the word at RETRY is 1, then a retry is in progress.)

```
000012  012727            MOV     (PC)+,(PC)+       ;SET RETRY COUNT
000014  000010  DRETRY:   .WORD   RKCNT             ; :MAXIMUM RETRY COUNT
000016  000000  RETRY:    .WORD   0                 ;SIGN BIT SET IF DRIVE
                                                    ;RESET IN PROGRESS
```

RKCQE points to the block number Q.BLKN in the queue element:

```
000020  016705            MOV     RKCQE,R5          ;GET CURRENT QUEUE
        177764
                                                    ;ELEMENT POINTER
000024  011502            MOV     @R5,R2            ;PICK UP BLOCK NUMBER
000026  016504            MOV     Q$UNIT-1(R5),R4   ;GET REQUESTED UNIT
        000002
                                                    ;NUMBER IN HIGH BYTE
```

The controller requires the unit number in the top three bits of the word loaded into RKDA:

```
000032  006204            ASR     R4                ;SHIFT UNIT NUMBER
000034  006204            ASR     R4                ; TO HIGH 3 BITS
000036  006204            ASR     R4                ;  OF LOW BYTE
000040  000304            SWAB    R4                ;PUT UNIT NUMBER IN HIGH
                                                    ;3 BITS OF WORD
000042  042704            BIC     #^C<DAUNIT>,R4    ;ISOLATE UNIT IN DRIVE
        017777
                                                    ;SELECT BITS
000046  000404            BR      2$                ;ENTER COMPUTATION LOOP
```

The device unit and block number are known; the disk address for a read or write request must be calculated. Once determined, the disk address is stored in DISKAD in case it must be used again during a retry. The RK disk has 12 blocks per track, and two tracks per cylinder. To find the disk address, the block number is divided by 12, and the quotient and remainder are separated.

```
000050  060204  1$:       ADD     R2,R4             ;ADD 16R TO ADDRESS
000052  006202            ASR     R2                ;R2 = 8R
000054  006202            ASR     R2                ;R2 = 4R
000056  060302            ADD     R3,R2             ;R2 = 4R+S = NEW N
000060  010203  2$:       MOV     R2,R3             ;R3 = N = 16R+S
000062  042703            BIC     #^C<17>,R3        ;R3 = S
        177760
000066  040302            BIC     R3,R2             ;R2 = 16R
000070  001367            BNE     1$                ;LOOP IF R <> 0
000072  022703            CMP     #12.,R3           ;IF S < 12.
        000014
000076  003002            BGT     3$                ;   THEN F(S) = S
000100  062703            ADD     #4,R3             ;   ELSE F(S)=F(12+S')=
        000004
                                                    ;        16+S'=4+S
000104  060304  3$:       ADD     R3,R4             ;MERGE SECTOR INTO CYL
                                                    ;TO GET DISK ADDRESS
000106  010467            MOV     R4,DISKAD         ;SAVE IT
        000016
```

The next statement points R5 to a queue element, since perhaps this is a retry and R5 is not already set up:

```
   000112  016705   AGAIN:  MOV     RKCQE,R5;POINT TO QUEUE ELEMENT
           177672
```

This statement sets up the operation code for a write:

```
   000116  012703           MOV     #CSIE!FNWRITE!CSGO,R3 ;ASSUME A WRITE
           000103
                                                  ;FUNCTION
   000122  012704           MOV     (PC)+,R4      ;POINT TO DISK ADDRESS

                                                  ;REGISTER
   000124  177412  RKCSR:  .WORD   RKDA
```

The disk address is saved in DISKAD. The significance of the bits in DISKAD, from high order to low order, is as follows: unit, cylinder, track, and sector.

```
   000126  012714           MOV     (PC)+,@R4     ;LOAD DISK ADDRESS &
                                                  ;UNIT SELECT INTO RKDA
   000130  000000  DISKAD:  .WORD   0             ;SAVED COMPUTED DISK
                                                  ;ADDRESS
   000132  022525           CMP     (R5)+,(R5)+   ;ADVANCE TO BUFFER
                                                  ;ADDRESS IN QUEUE ELT
```

Much of the code in the handler is assembled based on the value of certain conditionals, such as MMG$T. The IF statement controls the assembly of the code that follows. If the handler is assembled with MMG$T equal to 1, code following the .IFF statements is assembled. If the handler does not have extended memory support enabled (that is, if MMG$T equals 0), code following the .IFT statements is assembled. Code following the .IFTF statements is always assembled, regardless of the value of MMG$T.

```
                   .IF EQ   MMG$T
                            MOV     (R5)+,-(R4)   ;PUT BUFFER ADDRESS INTO
                                                  ;RKBA
                   .IFF
```

$MPPTR is a pointer to the monitor routine $MPPHY. This routine is available for NPR device handlers to use. It converts the virtual buffer address supplied in the queue element into an 18-bit physical address that is returned on the stack. The monitor supplies the virtual address in two words: Q.PAR and Q.BUFF. This form is used because it can be directly used by character-oriented (programmed transfer) devices. NPR devices such as the RK disk must convert this pair of words into an 18-bit physical address consisting of a 16-bit low part and a two-bit extension. The extension bits are in positions 4 and 5 for use with UNIBUS controllers. The extension bits must be ORed into the command word being built for RKCS. Because the RK controller cannot accept a 22-bit address, the handler checks for a 22-bit address and returns an error if one is specified.

```
   000134  004777           JSR     PC,@$MPPTR    ;CONVERT USER VIRTUAL
           000366
                                                  ;ADDRESS TO PHYSICAL
   000140  012644           MOV     (SP)+,-(R4)   ;PUT LOW 16 BITS IN
                                                  ;RKBA,HIGH BITS ON STACK
   000142  012600           MOV     (SP)+,R0      ;GET HIGH-ORDER ADDRESS
```

```
                                                ;BITS <21:18>
   000144  032700         BIT     #1700,R0      ;22-BIT ADDRESS
           001700
                                                ;SPECIFIED?
   000150  001140         BNE     HERROR        ;YES, NOT VALID WITH
                                                ;THIS CONTROLLER
                  .ENDC ;EQ MMG$T
```

The next statement moves the word count Q.WCNT from the queue element into RKWC, the device word count register. RT–11 can transfer up to 32767 words per operation. However, it can never transfer an odd number of bytes.

```
   000142  012544         MOV     (R5)+,-(R4)   ;PUT WORD COUNT INTO RKWC
   000144  001407         BEQ     7$            ;0 COUNT => SEEK
```

The RK controller requires that all word counts be negative.

```
   000146  100403         BMI     5$            ;NEGATIVE => WRITE,
                                                ;SO ALL SET UP
   000150  005414         NEG     @R4           ;POSITIVE => READ, FIX
                                                ;COUNT FOR CONTROLLER
```

The next statement sets up the operation code for a read:

```
   000152  012703         MOV     #CSIE!FNREAD!CSGO,R3 ;FUNCTION IS READ
           000105
   000156         5$:
                  .IF NE  MMG$T
   000156  052603         BIS     R0,R3         ;MERGE HIGH ORDER ADDRESS
                                                ;BITS INTO FUNCTION
                  .ENDC ;NE MMG$T
   000160  010344         MOV     R3,-(R4)      ;START THE OPERATION
```

The next statement returns control to the monitor. The I/O transfer begins.

```
   000162  000207  6$:    RTS     PC            ;AWAIT INTERRUPT
```

The next statement sets up the operation code for a seek:

```
   000164  012744  7$:    MOV     #CSIE!FNSEEK!CSGO,-(R4) ;START UP A SEEK
           000111
   000172  000773         BR      6$            ;AWAIT INTERRUPT
```

Interrupt Service Section

The following code is reached when an interrupt occurs:

```
                  .SBTTL  INTERRUPT ENTRY POINT
```

The .DRAST macro:

```
   000174                 .DRAST  RK,5
                  .GLOBL  $INPTR
                  .IF B <>
   000174  000207         RTS     %7
                  .IFF
                          BR
                  .ENDC
```

The abort entry point is the word preceding RKINT. Since no abort entry point was specified in the .DRAST macro, an RTS PC instruction was generated.

At interrupt time, the new PC (RKINT) and new PS (340) are used. the handler calls the monitor through $INPTR in the handler to $INTEN in the monitor. The monitor switches to system state, lowers priority from 7 to 5, and calls the handler back.

```
000176  004577  RKINT:: JSR     %5,@$INPTR
        000340
000202  000100          .WORD   ^C<5*^040>&^0340
```

The monitor calls the handler back at this point. Execution is now at priority 5 and is in system state. The hardware has now finished the I/O operation, and the handler must determine whether the transfer was successful or whether there was an error.

```
000214  016705          MOV     RKCSR,R5
        177704
000220  062705          ADD     #RKER-RKDA,R5   ;R5->ERROR STATUS
        177770
                                                ;REGISTER
000210  012504          MOV     (R5)+,R4        ;GET ERROR REGISTER,
                                                ;POINT TO RKCS
```

The value of RETRY is negative if a drive reset was just done. Bit 15 is the retry flag.

```
000212  005767          TST     RETRY           ;WERE WE DOING A DRIVE
        177600
                                                ;RESET?
000216  100013          BPL     NORMAL          ;NO, NORMAL OPERATION
```

Bit 15 of RKCS is the error summary bit. If there was an error during a drive reset, it is handled in the same way as an error that occurred during an I/O transfer.

```
000220  005715          TST     @R5             ;ANY ERROR ON DRIVE
                                                ;RESET?
000222  100411          BMI     NORMAL          ;YES, HANDLE NORMALLY
```

R5 points to RKCS, the device control and status register:

```
000224  032715          BIT     #CSSCP,@R5      ;RESET DONE YET (SEARCH
        020000
                                                ;COMPLETE)?
```

The RK device interrupts twice during a drive reset. The first interrupt should be ignored.

```
000230  001472          BEQ     RTSPC           ;NO, INTERRUPT AGAIN
                                                ;WHEN RESET COMPLETE
```

The .FORK macro causes the code that follows it to be executed at priority 0 after all interrupts have been serviced, but before any jobs or their completion routines execute. This avoids executing lengthy code in the handler at high processor priority.

```
   000232                    .FORK    RKFBLK          ;CONTINUE RETRY AFTER
   000232  004577            JSR      %5,@$FKPTR
           000306
   000236  000240            .WORD    RKFBLK - .
                                                      ;RESET AT FORK LEVEL
   000240  105067  RKRETR:   CLRB     RETRY+1         ;CLEAR RESET-IN-PROGRESS
           177553
                                                      ;FLAG
   000244  000722            BR       AGAIN           ;RETRY THE OPERATION (AT
                                                      ;FORK LEVEL)

   000246  021527  NORMAL:   CMP      @R5,#CSRDY!CSIE!FNSEEK ;FIRST OF 2
           000310
                                                      ;INTERRUPTS CAUSED BY
                                                      ;SEEK?
```

The RK device interrupts twice for a seek. The first interrupt should be ignored by the handler. The seek is complete after the second interrupt has occurred.

```
   000252  001461            BEQ      RTSPC           ;YES, IGNORE IT.
                                                      ;INTERRUPT AGAIN WHEN
                                                      ;COMPLETE
```

The next statement is reached when I/O is complete, or when there is an I/O error. The sign bit, bit 15, of RKCS is an error summary bit. If RKCS is negative, there was an error in the I/O transfer.

```
   000254  005715            TST      @R5             ;ANY ERRORS?
   000256  100065            BPL      DONE            ;NO, WE'RE ALL DONE
```

Errors are processed at fork level, priority 0.

```
   000260                    .FORK    RKFBLK          ;PROCESS ERRORS AT FORK
   000260  004577            JSR      %5,@$FKPTR
           000260
   000264  000212            .WORD    RKFBLK - .
                                                      ;LEVEL
```

The following block of code is generated if the system supports error logging:

```
                    .IF NE   ERL$G
```

R4 contains errors from RKER, the device error register. Unrecoverable errors that do not indicate hardware faults are not logged.

```
   000266  032704            BIT      #EROVR!ERWLO!ERNXM!ERNXD!ERNXC!ERNXS,R4
           062340
                                                      ;USER ERROR?
   000272  001027            BNE      RKERR           ;YES, DON'T LOG IT
```

Other types of errors are logged:

```
   000274  010705            MOV      PC,R5           ;GET REGISTER SAVE AREA
                                                      ; ADDRESS
   000276  062705            ADD      #RKRBUF-.,R5    ; IN A PIC WAY
           000210
   000302  010502            MOV      R5,R2           ;SAVE ADDRESS IN FOR
                                                      ;ERROR LOGGER
   000320  016703            MOV      RKCSR,R3
           177600
   000324  062703            ADD      #RKDS-RKDA,R3   ;R3->CONTROLLER
           177766
                                                      ;REGISTERS
   000310  012704            MOV      #RKNREG,R4      ;GET COUNT OF REGISTERS
           000007
```

```
                                                        ;TO COPY
   000314  012325  RKRREG: MOV     (R3)+,(R5)+          ;MOVE REGISTERS TO BUFFER
   000316  005304          DEC     R4                   ;DONE?
   000320  001375          BNE     RKRREG               ;NO, MORE
   000342  016703          MOV     DRETRY,R3
           177446
   000346  000303          SWAB    R3
   000350  062703          ADD     #RKNREG,R3           ;R3= MAX RETRY COUNT/
           000007
                                                        ;NUMBER OF REGISTERS
   000326  016705          MOV     RKCQE,R5             ;POINT TO THIRD WORD OF
           177456
                                                        ;QUEUE ELEMENT
   000332  116704          MOVB    RETRY,R4             ;GET RK$COD(=0)/RETRY
           177460
                                                        ;COUNT
   000336  005304          DEC     R4                   ;RETRY COUNT VALUE AFTER
                                                        ;IT IS DECREMENTED
   000340  004777          JSR     PC,@$ELPTR           ;CALL ERROR LOGGER
           000172
   000372  016705          MOV     RKCSR,R5
           177526
   000376  062705          ADD     #RKER-RKDA,R5        ;RESET REGISTERS
           177770
   000350  012504          MOV     (R5)+,R4             ; ON RETURN
                   .ENDC ;NE ERL$G
```

The next section of code retries both soft (such as checksum) and hard (hardware malfunction) errors. R5 points to RKCS.

```
   000352  012715  RKERR:  MOV     #FNRST!CSGO,@R5 ;RESET CONTROLLER
           000001
```

When the controller is ready, it sets bit 7 of the low byte of RKCS.

```
   000356  105715  3$:     TSTB    @R5                  ;WAIT
   000360  100376          BPL     3$                   ; FOR RESET TO TAKE
   000362  105367          DECB    RETRY                ;ANY RETRIES LEFT?
           177430
   000366  001414          BEQ     HERROR               ;NO, CALL IT A HARD ERROR
   000370  032704          BIT     #ERDRE!ERSKE,R4      ;SEEK INCOMPLETE OR DRIVE
           110000
```

Both seek incomplete and drive error require a drive reset before the operation can be retried.

```
                                                        ;ERROR?
```

Common errors for which the I/O transfer should be retried are checksum errors, data late errors, and timing errors.

```
   000374  001721          BEQ     RKRETR               ;NO, JUST RETRY OPERATION
```

The next statement is reached if there is a seek incomplete or drive error condition. RKDA was cleared by the controller reset above, but the disk address is saved in DISKAD.

```
   000376  016737          MOV     DISKAD,@RKCSR        ;YES, RESELECT DRIVE
           177526
           177412
```

The flag in RETRY is set here so that on the next interrupt, the handler will know that a drive reset, and not an I/O transfer, was the last operation done.

```
000404  052767           BIS     #100000,RETRY    ;SET RESET-IN-PROGRESS
        100000
        177404
                                                  ;FLAG
000412  012715           MOV     #CSIE!FNDRST!CSGO,@R5 ;START A DRIVE
        000115
                                                  ;RESET
```

The next statement returns control to the monitor to wait for the drive reset or seek to finish.

```
000416  000207  RTSPC:   RTS     PC               ;AWAIT INTERRUPT
```

The next statement is reached when there has been an I/O error that has been retried and could not be corrected.

```
000420  016705  HERROR:  MOV     RKCQE,R5         ;HARD ERROR, POINT TO
        177364
                                                  ;QUEUE ELEMENT
```

The handler reports the error to the user program by setting bit 0 (the hard error bit) in the Channel Status Word. R5 points to Q.BLKN; R5, decremented by 2, points to the address of the CSW.

```
000424  052755           BIS     #HDERR$,@-(R5)   ;SET HARD ERROR STATUS
        000001
                                                  ;IN CHANNEL
                .IF NE   ERL$G
000430  000411           BR      RKEXIT           ;EXIT AFTER ERROR

```

The following section is reached after a successful transfer. Successful transfers are logged at fork level, priority 0.

```
000432          DONE:    .FORK   RKFBLK           ;CALL ERROR LOG AT FORK
000432  004577           JSR     %5,@$FKPTR
        000106
000436  000040           .WORD   RKFBLK - .
                                                  ;LEVEL FOR SUCCESS
000472  005767           TST     SCSFLG           ;LOGGING SUCCESSES?
        177504
000476  001006           BNE     RKEXIT           ;NOPE...
000500  012704           MOV     (PC)+,R4         ;SUCCESS, SET RK ID
                                                  ;CODE IN HIGH BYTE,
000502     377           .BYTE   377,RK$COD       ; -1 IN LOW BYTE FOR
000503     000
                                                  ;SUCCESS
000444  016705           MOV     RKCQE,R5         ;POINT TO THIRD WORD OF
        177340
                                                  ;QUEUE ELEMENT
000450  004777           JSR     PC,@$ELPTR       ;CALL ERROR LOGGER
        000062
                .IFF
                DONE:
                .ENDC
000454  005067  RKEXIT:  CLR     RETRY            ;CLEAR RETRY AND RESET
        177336
                                                  ;FLAGS
```

I/O Completion Section

The .DRFIN macro:

```
   000460                 .DRFIN  RK              ;EXIT TO COMPLETION
```

The .DRFIN macro generates the following block of code. This section lets the monitor know that the I/O operation is complete so that the queue element can be returned to the available element list. Control returns to the monitor with the JMP instruction. The monitor alerts the program if it was waiting for this transfer to finish, or it runs the program's completion routine, if any.

```
                       .GLOBL  RKCQE
   000460  010704              MOV     %7,%4
   000462  062704              ADD     #RKCQE-.,%4
           177326
   000466  013705              MOV     @#^054,%5
           000054
   000472  000175              JMP     @^0270(5)
           000270

   000476  000000      RKFBLK: .WORD   0,0,0,0         ;FORK QUEUE BLOCK
   000500  000000
   000502  000000
   000504  000000
                       .IF NE ERL$G
   000506              RKRBUF: .BLKW   RKNREG          ;ERROR LOG STORAGE FOR
                                                       ;REGISTERS
                       .ENDC
```

Bootstrap Driver

```
                       .SBTTL  BOOTSTRAP DRIVER

```

The .DRBOT macro:

```
   000524                 .DRBOT  RK,BOOT1,READ
```

Termination Section

The .DREND macro is generated by .DRBOT:

```
   000524                 .DREND  RK
   000524          .PSECT  RKDVR
                   .IIF NDF RK$END, RK$END:
                   .IF EQ  .-RK$END
   000524          RK$END::
                   .IF NE MMG$T
```

Since the handler is for a system device, .DRBOT invokes .DREND to allocate the following table of pointers. The pointers are to routines in the Resident Monitor. Some of the following pointers are optional, and their assembly depends on which system conditionals are defined.

```
   000524  000000  $RLPTR:: .WORD  0
   000526  000000  $MPPTR:: .WORD  0
```

```
000530  000000  $GTBYT:: .WORD  0
000532  000000  $PTBYT:: .WORD  0
000534  000000  $PTWRD:: .WORD  0
                .ENDC
                .IF NE ERL$G
000536  000000  $ELPTR:: .WORD  0
                .ENDC
                .IF NE TIM$IT
000540  000000  $TIMIT:: .WORD  0
                .ENDC
000542  000000  $INPTR:: .WORD  0
000544  000000  $FKPTR:: .WORD  0
                .GLOBL  RKSTRT
```

The following line marks the end of the loadable portion of the handler. It is used to determine the handler's length.

```
        000546' RKEND ==
                .ENDC
                .IIF NDF TPS, TPS=177564
                .IIF NDF TPB, TPB=177566
        000012  LF=12
        000015  CR=15
        001000  B$BOOT=1000
        004716  B$DEVN=4716
        004722  B$DEVU=4722
        004730  B$READ=4730
                .IF EQ  MMG$T
                B$DNAM=^RRK
                .IFF
        071120  B$DNAM=^RRKX
                .ENDC
 000200         .ASECT
        000062  .=62
 000062 000000'         .WORD   RKBOOT,RKBEND-RKBOOT,READ-RKBOOT
 000064 001000
 000066 000210
 000000         .PSECT  RKBOOT
 000000 000240  RKBOOT::NOP
 000002 000416          BR      BOOT1

        000040' .       = RKBOOT+40             ;PUT THE JUMP BOOT INTO
                                                ;SYSCOM AREA
 000040 000137  BOOT1:  JMP     @#BOOT-RKBOOT   ;START THE BOOTSTRAP
        000574

        000210' .       = RKBOOT+210
 000210 012703  READ:   MOV     #12.,R3 ;PHYSICAL BLOCK TO RK
        000014
                                                ;DISK ADDRESS
 000214 000402          BR      2$              ;ENTER BLOCK NUMBER
                                                ;COMPUTATION

 000216 062703  1$:     ADD     #20,R3          ;CONVERT DISK ADDRESS
        000020
 000222 162700  2$:     SUB     #12.,R0
        000014
 000226 100373          BPL     1$
 000230 060300          ADD     R3,R0           ;R0 HAS DISK ADDRESS
 000232 016703          MOV     BOTCSR,R3       ;POINT TO HARDWARE DISK
        000344
                                                ;ADDRESS REGISTER
 000236 042713          BIC     #^C<DAUNIT>,@R3 ;LEAVE THE UNIT NUMBER
        017777
 000242 050013          BIS     R0,@R3          ;PUT DISK ADDRESS INTO
                                                ;CONTROLLER
 000244 010243          MOV     R2,-(R3)        ;BUFFER ADDRESS
 000246 010143          MOV     R1,-(R3)        ;WORD COUNT
 000250 005413          NEG     @R3             ;(NEGATIVE)
 000252 012743          MOV     #FNREAD!CSGO,-(R3) ;START DISK READ
        000005
 000256 105713  3$:     TSTB    @R3             ;WAIT UNTIL COMPLETE
 000260 100376          BPL     3$
 000262 005713          TST     @R3             ;ANY ERRORS
 000264 100577          BMI     BIOERR          ;HARD HALT ON ERROR
                        ;CLC                    ;CARRY IS CLEAR FROM
                                                ;'TST' ABOVE
```

```
   000266  000207           RTS     PC

           000574'  .       = RKBOOT+574
   000574  012706  BOOT:    MOV     #10000,SP          ;SET STACK POINTER
           010000
   000600  013746           MOV     @(PC)+,-(SP)       ;GET THE RK UNIT NUMBER
   000602  177412  BOTCSR:  .WORD   RKDA
   000604  006116           ROL     @SP                ;SHIFT IT
   000606  006116           ROL     @SP                ; AROUND THE TOP
   000610  006116           ROL     @SP                ;  TO THE LOW 3 BITS
   000612  006116           ROL     @SP                ;    OF THE WORD
   000614  042716           BIC     #^C<7>,@SP         ;EXTRACT THE UNIT NUMBER
           177770
   000620  012700           MOV     #2,R0              ;READ IN SECOND PART OF
           000002
                                                       ;BOOT FROM BLOCK 2
   000624  012701           MOV     #<4*400>,R1        ;EVERY BLOCK BUT THE ONE
           002000
                                                       ;WE ARE IN (4 BLOCKS)
   000630  012702           MOV     #1000,R2           ;INTO LOCATION 1000
           001000
   000634  004767           JSR     PC,READ            ;GO READ IT IN
           177350
   000640  012737           MOV     #READ-RKBOOT,@#B$READ ;STORE START
           000210
           004730
                                                       ;LOCATION FOR READ
                                                       ;ROUTINE
   000646  012737           MOV     #B$DNAM,@#B$DEVN ;STORE RAD50 DEVICE NAME
           071120
           004716
   000654  012637           MOV     (SP)+,@#B$DEVU   ;STORE THE UNIT NUMBER
           004722
   000660  000137           JMP     @#B$BOOT           ;START SECONDARY BOOT
           001000

   000664                   .DREND  RK
   000546          .PSECT  RKDVR
                   .IIF NDF RK$END, RK$END:
                   .IF EQ  .-RK$END
                   RK$END::
                   .IF NE MMG$T
                   $RLPTR:: .WORD  0
                   $MPPTR:: .WORD  0
                   $GTBYT:: .WORD  0
                   $PTBYT:: .WORD  0
                   $PTWRD:: .WORD  0
                   .ENDC
                   .IF NE ERL$G
                   $ELPTR:: .WORD  0
                   .ENDC
                   .IF NE TIM$IT
                   $TIMIT:: .WORD  0
                   .ENDC
                   $INPTR:: .WORD  0
                   $FKPTR:: .WORD  0
                   .GLOBL  RKSTRT
                   RKEND == .
                   .IFF
   000664          .PSECT  RKBOOT
                   .IIF LT <RKBOOT-.+664>, .ERROR  ;PRIMARY BOOT TOO LARGE
           000664' .        = RKBOOT+664
   000664  004167  BIOERR: JSR     R1,REPORT
           000002
   000670  000766           .WORD   IOERR-RKBOOT
```

The following routine is entered when an error occurs. It prints the fatal part of the message, followed by the message text, a carriage return, and two line feeds.

```
000672  012700  REPORT: MOV     #BOOTF-RKBOOT,R0
        000746
000676  004167          JSR     R1,REP
        000030
000702  012100          MOV     (R1)+,R0
000704  004167          JSR     R1,REP
        000022
```

```
000710  012700            MOV     #CRLFLF-RKBOOT,R0
        000762
000714  004167            JSR     R1,REP
        000012
000720  000005            RESET
000722  000000            HALT
000724  000776            BR      .-2
```

The following routine prints the error message:

```
000726  112037  REPOR:    MOVB    (R0)+,@#TPB
        177566
000732  105737  REP:      TSTB    @#TPS
        177564
000736  100375            BPL     REP
000740  105710            TSTB    @R0
000742  001371            BNE     REPOR
000744  000201            RTS     R1

000746     015  BOOTF:    .ASCIZ  <CR><LF>"?BOOT-U-"<200>
000762     015  CRLFLF:   .ASCIZ  <CR><LF><LF>
000766     111  IOERR:    .ASCIZ  "I/O ERROR"
                          .EVEN
001000          RKBEND::
                .ENDC
59
60      000001            .END
```

The symbol table is generated at the end of the assembly listing:

```
Symbol table

ABTIO$= 001000          ERNXM  = 002000          REPOR   000726R      003
AGAIN   000112R     002 ERNXS  = 000040          REPORT  000672R      003
BAREA   000630          EROVR  = 040000          RETRY   000016R      002
BIOERR  000664R     003 ERPGE  = 004000          RKBA   = 177410
BOOT    000574R     003 ERSKE  = 010000          RKBEND  001000RG     003
BOOTF   000746R     003 ERTE   = 000400          RKBOOT  000000RG     003
BOOT1   000040R     003 ERWCK  = 000001          RKCNT  = 000010
BOTCSR  000602R     003 ERWLO  = 020000          RKCQE   000010RG     002
BTCSR = 002410          FILST$= 100000           RKCS   = 177404
B$BOOT= 001000          FNDRST= 000014           RKCSR   000124R      002
B$DEVN= 004716          FNRCHK= 000012           RKDA   = 177412
B$DEVU= 004722          FNREAD= 000004           RKDS   = 177400
B$DNAM= 071120          FNRST  = 000000          RKDSIZ= 011300
B$READ= 004730          FNSEEK= 000010           RKEND  = 000606RG    002
CR    = 000015          FNWCHK= 000006           RKER   = 177402
CRLFLF  000762R     003 FNWLK  = 000016          RKERR   000404R      002
CSBA16= 000020          FNWRIT= 000002           RKEXIT  000514R      002
CSBA67= 000060          HDERR$= 000001           RKFBLK  000536R      002
CSERR = 100000          HERROR  000452R      002 RKINT   000206RG     002
CSFMT = 002000          HNDLR$= 004000           RKLQE   000006RG     002
CSFUN = 000016          IOERR   000766R      003 RKNREG= 000007
CSGO  = 000001          LF     = 000012          RKRBUF  000546R      002
CSHE  = 040000          MMG$T  = 000001          RKRETR  000254R      002
CSIE  = 000100          NORMAL  000262R      002 RKRREG  000334R      002
CSINHB= 004000          O.BAD   000556           RKSTRT  000000RG     002
CSRDY = 000200          O.CSR   000442           RKSTS  = 100000
CSSCP = 020000          O.GOOD  000554           RKSYS   000006RG     002
CSSSE = 000400          O.RTRY  000602           RKWC   = 177406
DACYL = 017740          O.SUCC  000616           RK$COD= 000000
DASC  = 000017          O.VEC   000562           RK$CSR= 177400 G
DASUR = 000020          Q$BLKN= 000000           RK$END  000564RG     002
DAUNIT= 160000          Q$BUFF= 000004           RK$VEC= 000220 G
DISKAD  000130R     002 Q$COMP= 000010           RONLY$= 040000
DONE    000464R     002 Q$CSW  = 177776          RTE$M  = 000000
DRETRY  000014R     002 Q$FUNC= 000002           RTSPC   000450R      002
DSDPL = 010000          Q$JNUM= 000003           SCSFLG  000202R      002
DSDRU = 002000          Q$LINK= 177774           SPECL$= 010000
DSDRY = 000200          Q$PAR  = 000012          SPFUN$= 002000
DSID  = 160000          Q$UNIT= 000003           TIM$IT= 000001
DSREDY= 000100          Q$WCNT= 000006           TPB    = 177566
DSRK05= 004000          Q.BLKN= 000004           TPS    = 177564
DSSC  = 000017          Q.BUFF= 000010           VARSZ$= 000400
DSSCOK= 000020          Q.COMP= 000014           WONLY$= 020000
DSSIN = 001000          Q.CSW  = 000002          $ELPTR  000576RG     002
DSSOK = 000400          Q.ELGH= 000024           $FKPTR  000604RG     002
DSWPS = 000040          Q.FUNC= 000006           $GTBYT  000570RG     002
ELRK  = 000002          Q.JNUM= 000007           $INPTR  000602RG     002
```

```
EOF$   = 020000          Q.LINK= 000000           $MPPTR  000566RG    002
ERCSE  = 000002          Q.PAR  = 000016          $PTBYT  000572RG    002
ERDLT  = 001000          Q.UNIT= 000007           $PTWRD  000574RG    002
ERDRE  = 100000          Q.WCNT= 000012           $RLPTR  000564RG    002
ERL$G  = 000001          READ    000210R      003 $TIMIT  000600RG    002
ERNXC  = 000100          REP     000732R      003 ...V2 = 000200
ERNXD  = 000200

. ABS.  000642    000    (RW,I,GBL,ABS,OVR)
        000000    001    (RW,I,LCL,REL,CON)
RKDVR   000606    002    (RW,I,LCL,REL,CON)
RKBOOT  001000    003    (RW,I,LCL,REL,CON)

Errors detected:  0

*** Assembler statistics

Work  file  reads: 0
Work  file writes: 0
Size of work file: 10114 Words  ( 40 Pages)
Size of core pool: 15104 Words  ( 59 Pages)

Elapsed time: 00:00:18.00
,RK/L:ME:TTM=CND,RK
DX V04.01        MACRO V04.00  17-OCT-79 23:30:12
```

Figure A–2: DX Diskette Handler

TABLE OF CONTENTS


```
                        ;CONDITIONAL FILE FOR HANDLER EXAMPLES
                        ;
                        ;CND.MAC
                        ;
                        ;SGW
                        ;
                        ;ASSEMBLE WITH HANDLER .MAC FILE TO ENABLE
                        ;18-BIT I/O, TIME-OUT, AND ERROR LOGGING
                        ;FOR HANDLER.
                        ;
        000001  MMG$T   = 1                ;18-BIT I/O
        000001  ERL$G   = 1                ;ERROR LOGGING
        000001  TIM$IT  = 1                ;TIME-OUT
                        .TITLE  DX V04.01       RX01 FLOPPY DISK HANDLER
                        .IDENT  /X05.07/
                        .SBTTL  COPYRIGHT NOTICE
                        .ENABL  LC

                ;            COPYRIGHT (c) 1982, 1983 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 WHICH  IS NOT
                ; SUPPLIED BY DIGITAL.
```

Preamble Section

```
        .SBTTL  MACROS AND DEFINITIONS

        .MCALL  .DRDEF

```

Monitor offsets and SYSCOM locations are defined with mnemonics so that references to them can be found easily:

```
                ; RT-11 SYSCOM LOCATIONS

000044          JSW     = 44    ;JOB STATUS WORD
000054          SYSPTR  = 54    ;POINTER TO BASE OF RMON

000274          SYUNIT  = 274   ;UNIT NUMBER OF SYSTEM
                                ;DEVICE (HI BYTE)
000432          P1EXT   = 432   ;OFFSET FROM $RMON TO
                                ;EXTERNAL ROUTINE ADDRESS
000404          PNPTR   = 404   ;OFFSET FROM $RMON TO
                                ;$PNAME TABLE
```

The following two macros are used for consistency checks within the handler. They generate P errors at assembly time when inconsistencies exist.

```
        .MACRO  .ASSUME A1,CND,A2
        .IF     CND     <A1>-<A2>
        .IFF
        .ERROR  ;"A1 CND A2" IS NOT TRUE
        .ENDC
        .ENDM   .ASSUME

        .MACRO  .BR     TO
        .IF DF  TO
        .IF NE  .-<TO>
        .ERROR  ;NOT AT LOCATION "TO"
        .ENDC
        .ENDC
        .ENDM   .BR
```

If DXT$O = 1, there are two controllers:

```

        ; RX01 CONTROLLER DEFAULTS

        .IIF NDF DXT$0, DXT$0=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:

```
000000                  .DRDEF  DX,22,FILST$!SPFUN$,756,177170,264
                .MCALL  .DRAST,.DRBEG,.DRBOT,.DREND,.DRFIN,.DRSET
                .MCALL  .DRVTB,.FORK,.QELDF
                .IIF NDF RTE$M, RTE$M=0
                .IIF NE RTE$M, RTE$M=1
                .IIF NDF TIM$IT, TIM$IT=0
```

```
        000001  .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
        000001  .IIF NE ERL$G, ERL$G=1
                .IIF NE TIM$IT, .MCALL  .TIMIO,.CTIMI
000000          .QELDF
        000000  Q.LINK=0
        000002  Q.CSW=2.
        000004  Q.BLKN=4.
        000006  Q.FUNC=6.
        000007  Q.JNUM=7.
        000007  Q.UNIT=7.
        000010  Q.BUFF=^O10
        000012  Q.WCNT=^O12
        000014  Q.COMP=^O14
                .IRP    X,<LINK,CSW,BLKN,FUNC,JNUM,UNIT,BUFF,WCNT,COMP>
                Q$'X=Q.'X-4
                .ENDR
        177774  Q$LINK=Q.LINK-4
        177776  Q$CSW=Q.CSW-4
        000000  Q$BLKN=Q.BLKN-4
        000002  Q$FUNC=Q.FUNC-4
        000003  Q$JNUM=Q.JNUM-4
        000003  Q$UNIT=Q.UNIT-4
        000004  Q$BUFF=Q.BUFF-4
        000006  Q$WCNT=Q.WCNT-4
        000010  Q$COMP=Q.COMP-4
                .IF EQ MMG$T
                Q.ELGH=^O16
                .IFF
        000016  Q.PAR=^O16
        000012  Q$PAR=^O12
        000024  Q.ELGH=^O24
                .ENDC
        000001  HDERR$=1
        020000  EOF$=20000
        000400  VARSZ$=400
        001000  ABTIO$=1000
        002000  SPFUN$=2000
        004000  HNDLR$=4000
        010000  SPECL$=10000
        020000  WONLY$=20000
        040000  RONLY$=40000
        100000  FILST$=100000
        000756  DXDSIZ=494.
        000022  DX$COD=22
        102022  DXSTS=<22>!<FILST$!SPFUN$>
                .IIF NDF DX$CSR, DX$CSR=177170
                .IIF NDF DX$VEC, DX$VEC=264
                .GLOBL  DX$CSR,DX$VEC

                ; 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
```

Internal consistency checks:

```
000000          .ASSUME CSRD&2  NE 0          ;2 BIT MUST BE ON IN READ
                .IF     NE      <CSRD&2>-<0>
                .IFF
                .ERROR  ;"CSRD&2 NE 0" IS NOT TRUE
                .ENDC
000000          .ASSUME CSWRT&2 EQ 0          ;2 BIT MUST BE OFF IN WRITE
                .IF     EQ      <CSWRT&2>-<0>
                .IFF
                .ERROR  ;"CSWRT&2 EQ 0" IS NOT TRUE
                .ENDC
000000          .ASSUME CSWRTD&2 EQ 0         ;2 BIT MUST BE OFF IN WRITE
                .IF     EQ      <CSWRTD&2>-<0>
                .IFF
                .ERROR  ;"CSWRTD&2 EQ 0" IS NOT TRUE
                .ENDC

                ; 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 FUNCTION 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 PROVIDES 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

000000          .ASECT

        000200  .       = 200                 ;INSTALLATION CHECK GOES
                                              ;HERE

000200  000240          NOP                   ;SAME CHECK FOR SYSTEM
                                              ;AND NON-SYSTEM HANDLER
000202  032777          BIT     #CSRX02,@176  ;IS THE RX02 BIT ON?
        004000
        177766
000210  001001          BNE     1$            ;YES, THIS ISN'T AN RX01,
                                              ;SO DON'T INSTALL IT
```

```
 000210   001563          BEQ     O.GOOD              ;NOPE, IS AN RX01,
                                                      ;INSTALL IT
 000212   000563          BR      O.BAD               ;YES, AN RX02, DON'T
                                                      ;INSTALL IT

                  ; Routine to find the entry for DX in the monitor
                  ; device tables

 000214   013701  FINDRV: MOV     @#SYSPTR,R1         ;R1->$RMON
          000054
 000220   066101          ADD     PNPTR(R1),R1        ;R1->$PNAME TABLE
          000404
 000224   010102          MOV     R1,R2               ;SAVE THE POINTER
 000226   022127  10$:    CMP     (R1)+,#-1           ;SEARCHING FOR END OF
          177777
                                                      ;$ENTRY TABLE
 000232   001375          BNE     10$                 ;HAVEN'T FOUND IT YET...
 000234   005741          TST     -(R1)               ;FOUND, BACK UP TO IT
 000236   160201          SUB     R2,R1               ;R1=LENGTH OF $PNAME
                                                      ;AND $ENTRY TABLES
 000240   006201          ASR     R1                  ;R1=LENGTH OF A DEVICE
                                                      ;TABLE
 000242   022227  20$:    CMP     (R2)+,#<^RDX >      ;SEARCH FOR 'DX' ENTRY
          016300
 000246   001375          BNE     20$                 ;HAVEN'T FOUND IT YET...
 000250   005742          TST     -(R2)               ;FOUND, BACK UP TO IT
 000252   060102          ADD     R1,R2               ;R2->$ENTRY ENTRY
 000254   011201          MOV     (R2),R1             ;R1->HANDLER ENTRY
                                                      ;POINT
 000256   001140          BNE     O.GOOD              ;IT'S LOADED...
 000260   000540          BR      O.BAD               ;NOT LOADED...

                  ; The emt area for reads/writes of the handler is
                  ; placed here to leave room for code for the set options

 000262      017  BAREA:  .BYTE   17,10               ;CHANNEL 17, READ
 000263      010
 000264                   .BLKW                       ;BLOCK NUMBER
 000266                   .BLKW                       ;BUFFER
 000270   000400          .WORD   256.                ;WORD COUNT
 000272   000000          .WORD   0                   ;COMPLETION (WAIT)

                  .IIF GT,<.-356> .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.

 000274                   .DRSET  CSR,    160000, O.CSR,  OCT
 000274           .ASECT
                  .IF LE  .-400
          000400  .=400
                  .IFF
                  .=.-2
                  .ENDC
 000400   160000          160000
          000402  ...V2=.
 000402   012712          .RAD50  CSR
          000406  .=...V2+4
 000406      025          .BYTE   <O.CSR-400>/2
          000000  ...V2=0
                  .IRP X,<OCT>
                  .IF IDN <X>,<NUM>
                  ...V2=...V2!100
                  .IFF
                  .IF IDN <X>,<NO>
                  ...V2=...V2!200
                  .IFF
                  .IF IDN <X>,<OCT>
                  ...V2=...V2!140
```

```
                       .IFF
                       .ERROR  ;ILLEGAL PARAMETER X
                       .ENDC
                       .ENDC
                       .ENDC
                       .ENDR
                       .IF IDN <OCT>,<NUM>
                       ...V2=...V2!100
                       .IFF
                       .IF IDN <OCT>,<NO>
                       ...V2=...V2!200
                       .IFF
                       .IF IDN <OCT>,<OCT>
               000140  ...V2=...V2!140
                       .IFF
                       .ERROR  ;ILLEGAL PARAMETER OCT
                       .ENDC
                       .ENDC
                       .ENDC
  000407          140          .BYTE   ...V2
  000410       000000          .WORD   0
  000412                       .DRSET  VECTOR, 500,    O.VEC,  OCT
  000412               .ASECT
                       .IF LE  .-400
                       .=400
                       .IFF
               000410  .=.-2
                       .ENDC
  000410       000500          500
               000412  ...V2=.
  000412       105113          .RAD50  VECTOR
  000414       077552
               000416  .=...V2+4
  000416          073          .BYTE   <O.VEC-400>/2
               000000  ...V2=0
                       .IRP X,<OCT>
                       .IF IDN <X>,<NUM>
                       ...V2=...V2!100
                       .IFF
                       .IF IDN <X>,<NO>
                       ...V2=...V2!200
                       .IFF
                       .IF IDN <X>,<OCT>
                       ...V2=...V2!140
                       .IFF
                       .ERROR  ;ILLEGAL PARAMETER X
                       .ENDC
                       .ENDC
                       .ENDC
                       .ENDR
                       .IF IDN <OCT>,<NUM>
                       ...V2=...V2!100
                       .IFF
                       .IF IDN <OCT>,<NO>
                       ...V2=...V2!200
                       .IFF
                       .IF IDN <OCT>,<OCT>
               000140  ...V2=...V2!140
                       .IFF
                       .ERROR  ;ILLEGAL PARAMETER OCT
                       .ENDC
                       .ENDC
                       .ENDC
  000417          140          .BYTE   ...V2
  000420       000000          .WORD   0

                               .IF NE DXT$O
                               .DRSET  CSR2,   160000, O.CSR2, OCT
                               .DRSET  VEC2,   500,    O.VEC2, OCT
                               .ENDC;NE DXT$O

  000422                       .DRSET  RETRY,  RETRY,  O.RTRY, NUM
  000422               .ASECT
                       .IF LE  .-400
                       .=400
                       .IFF
               000420  .=.-2
                       .ENDC
```

```
   000420  000010          RETRY
           000422  ...V2=.
   000422  070534          .RAD50  RETRY
   000424  072150
           000426  .=...V2+4
   000426     103          .BYTE   <0.RTRY-400>/2
           000000  ...V2=0
                   .IRP X,<NUM>
                   .IF IDN <X>,<NUM>
                   ...V2=...V2!100
                   .IFF
                   .IF IDN <X>,<NO>
                   ...V2=...V2!200
                   .IFF
                   .IF IDN <X>,<OCT>
                   ...V2=...V2!140
                   .IFF
                   .ERROR  ;ILLEGAL PARAMETER X
                   .ENDC
                   .ENDC
                   .ENDC
                   .ENDR
                   .IF IDN <NUM>,<NUM>
           000100  ...V2=...V2!100
                   .IFF
                   .IF IDN <NUM>,<NO>
                   ...V2=...V2!200
                   .IFF
                   .IF IDN <NUM>,<OCT>
                   ...V2=...V2!140
                   .IFF
                   .ERROR  ;ILLEGAL PARAMETER NUM
                   .ENDC
                   .ENDC
                   .ENDC
   000427     100          .BYTE   ...V2
   000430  000000          .WORD   0

                           .IF NE ERL$G
   000432                  .DRSET  SUCCES, -1,     O.SUCC, NO

   000432          .ASECT
                   .IF LE  .-400
                   .=400
                   .IFF
           000430  .=.-2
                   .ENDC
   000430  177777          -1
           000432  ...V2=.
   000432  075013          .RAD50  SUCCES
   000434  011633
           000436  .=...V2+4
   000436     111          .BYTE   <O.SUCC-400>/2
           000000  ...V2=0
                   .IRP X,<NO>
                   .IF IDN <X>,<NUM>
                   ...V2=...V2!100
                   .IFF
                   .IF IDN <X>,<NO>
                   ...V2=...V2!200
                   .IFF
                   .IF IDN <X>,<OCT>
                   ...V2=...V2!140
                   .IFF
                   .ERROR  ;ILLEGAL PARAMETER X
                   .ENDC
                   .ENDC
                   .ENDC
                   .ENDR
                   .IF IDN <NO>,<NUM>
                   ...V2=...V2!100
                   .IFF
                   .IF IDN <NO>,<NO>
           000200  ...V2=...V2!200
                   .IFF
                   .IF IDN <NO>,<OCT>
                   ...V2=...V2!140
                   .IFF
                   .ERROR  ;ILLEGAL PARAMETER NO
                   .ENDC
                   .ENDC
                   .ENDC
```

```
     000437     200          .BYTE    ...V2
     000440  000000          .WORD    0
                             .ENDC


     000442                  .DRSET   WRITE,  DXT$O*2+1, O.WP,NO
     000442          .ASECT
                     .IF LE   .-400
                     .=400
                     .IFF
             000440  .=.-2
                     .ENDC
     000440  000001          DXT$O*2+1
             000442  ...V2=.
     000442  111231          .RAD50   \WRITE\
     000444  076710
             000446  .=...V2+4
     000446     116          .BYTE    <O.WP-400>/2
             000000  ...V2=0
                     .IRP X,<NO>
                     .IF IDN <X>,<NUM>
                     ...V2=...V2!100
                     .IFF
                     .IF IDN <X>,<NO>
                     ...V2=...V2!200
                     .IFF
                     .IF IDN <X>,<OCT>
                     ...V2=...V2!140
                     .IFF
                     .ERROR   ;ILLEGAL PARAMETER X
                     .ENDC
                     .ENDC
                     .ENDC
                     .ENDR
                     .IF IDN <NO>,<NUM>
                     ...V2=...V2!100
                     .IFF
                     .IF IDN <NO>,<NO>
             000200  ...V2=...V2!200
                     .IFF
                     .IF IDN <NO>,<OCT>
                     ...V2=...V2!140
                     .IFF
                     .ERROR   ;ILLEGAL PARAMETER NO
                     .ENDC
                     .ENDC
                     .ENDC
     000447     200          .BYTE    ...V2
     000450  000000          .WORD    0

```

The code to process each SET option follows the .DRSET macro calls. Normally, SET options change only the disk-resident copy of a handler, not the memory-resident copy of a handler. The DX handler SET options include special code to modify both the memory-resident as well as the disk-resident copy of the handler.

```
             002366          BTCSR = <DXEND-DXSTRT>+<BOTCSR-DXBOOT>+1000

     000452  020003  O.CSR:  CMP      R0,R3            ;IS CSR IN RANGE?
                                                       ;(>160000)
     000454  103442          BLO      O.BAD            ;NOPE...
     000456  010067          MOV      R0,176           ;YES, INSTALLATION
             177514
                                                       ;CODE NEEDS IT


                     ; When the csr for units 0 and 1 is changed, the
                     ; bootstrap must be altered such that it will use the
                     ; correct controller.

     000462  010701          MOV      PC,R1            ;R1->READ/WRITE EMT AREA
     000464  062701          ADD      #BAREA-.+4,R1    ;(BUFFER ADDRESS WORD)
             177602
     000470  010702          MOV      PC,R2            ;BUILD ADDRESS OF BUFR
     000472  062702          ADD      #1000-.,R2       ;(WHICH WILL OVERWRITE
             000306
```

```
                                                             ;CORE COPY OF BLOCK 1)
000476  010211            MOV     R2,(R1)                    ;SET THE BUFFER ADDRESS
000500  012741            MOV     #BTCSR/1000,-(R1)          ;SET TO BLOCK NUMBER
        000002
                                                             ;TO READ/WRITE   (BOOT
                                                             ;BLOCK THAT NEEDS
                                                             ;MODIFICATION)
000504  005741            TST     -(R1)                      ;R1->EMT AREA
000506  010003            MOV     R0,R3                      ;SAVE CSR ELSEWHERE,
                                                             ;EMT NEEDS R0
000510  010100            MOV     R1,R0                      ;R0->EMT AREA FOR READ
000512  104375            EMT     375                        ; *** (.READW) ***
000514  103422            BCS     O.BAD
000516  010362            MOV     R3,<BTCSR&777>(R2)         ;SET THE NEW CSR
        000366
000522  010100            MOV     R1,R0                      ;R0->EMT AREA FOR WRITE
000524  105260            INCB    1(R0)                      ;BUMP 'READ' TO 'WRITE'
        000001
000530  104375            EMT     375                        ; *** (.WRITW) ***
000532  103413            BCS     O.BAD
000534  010100            MOV     R1,R0                      ;R0->EMT AREA
000536  105360            DECB    1(R0)                      ;BUMP 'WRITE' TO 'READ'
        000001
000542  012760            MOV     #1,2(R0)                   ;OF BLOCK 1 OF HANDLER
        000001
        000002
000550  104375            EMT     375                        ; *** (.READW) ***
000552  103403            BCS     O.BAD

                          .IF EQ DXT$O
000554  010367            MOV     R3,RXCSA
        000476'
                          .IFF
                          MOV     R3,DXCSR
                          .ENDC; EQ DXT$O

000560  005727   O.GOOD:  TST     (PC)+                      ;GOOD RETURN (C CLEAR)
000562  000261   O.BAD:   SEC                                ;ERROR RETURN (C SET)
000564  000207            RTS     PC

000566  020003   O.VEC:   CMP     R0,R3                      ;VECTOR IN RANGE?
000570  103374            BHIS    O.BAD                      ;NOPE...
000572  032700            BIT     #3,R0                      ;YES, BUT ON A VECTOR
        000003
                                                             ;BOUNDRY?
000576  001371            BNE     O.BAD                      ;NOPE...
                          .IF EQ DXT$O
000600  010067            MOV     R0,DXSTRT                  ;YES, SET IT IN ENTRY
        000000'
                                                             ;AREA
                          .IFF
                          MOV     R0,DX$VTB                  ;PLACE IT IN MULTI-
                                                             ;VECTOR TABLE
                          .ENDC;NE DXT$O

000604  000765            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
                          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 BOUNDRY?
                          BNE     O.BAD                      ;NOPE...
                          MOV     R0,DX$VTB+6                ;YES, PLACE IN MULTI-
                                                             ;VECTOR TABLE
                          BR      O.GOOD
                          .ENDC

000606  020003   O.RTRY:  CMP     R0,R3                      ;ASKING FOR TOO MANY?
000610  101364            BHI     O.BAD                      ;YES, USER IS BEING
                                                             ;UNREASONABLE
000612  010067            MOV     R0,DRETRY                  ;NOPE, SO TELL THE
        000034'
                                                             ;HANDLER
```

```
000616  001360            BNE     O.GOOD            ;OKAY IF NON-ZERO
000620  000760            BR      O.BAD             ;CAN'T ASK FOR NO
                                                    ;RETRIES
                          .IF NE ERL$G
000622  012703   O.SUCC:  MOV     #0,R3             ;'SUCCESS' ENTRY POINT
        000000
                                                    ; (MUST BE TWO WORDS)
000626  010367            MOV     R3,SCSFLG         ;'NOSUCCES' ENTRY POINT
        000016'
000632  000752            BR      O.GOOD
                          .ENDC
000634  000240   O.WP:    NOP                       ;'WRITE' ENTRY POINT
000636  005727            TST     (PC)+
000640  000261            SEC                       ;'NOWRITE' ENTRY POINT
000642  006127            ROL     (PC)+             ;SAVE USER'S SELECTION
000644  000000   O.WPF:    .WORD  0                 ;:WRITE-PROTECT SELECT
000646  042767            BIC     #<^C1>,O.WPF      ;DISCARD OLD SELECTION
        177776
        177770
000654  110100            MOVB    R1,R0             ;MOVE UNIT NUMBER TO
                                                    ;WHERE WE NEED IT
000656  020003            CMP     R0,R3             ;IS UNIT WITHIN RANGE?
000660  101340            BHI     O.BAD             ;NOPE...

                 ;NOW TO ALTER THE ON-DISK COPY OF THE PROTECTION TABLE

000662  010046            MOV     R0,-(SP)          ;SAVE THE SELECTED UNIT
                                                    ;NUMBER
000664  060700            ADD     PC,R0             ;ASSEMBLE, IN A PIC
                                                    ;FASHION, A POINTER TO
000666  062700            ADD     #DXWPRO-.,R0      ;THE PROTECTION TABLE
        177126'
000672  116710            MOVB    O.WPF,(R0)        ;SET THE WRITE-PROTECT
        177746
                                                    ;STATUS
000676  012600            MOV     (SP)+,R0          ;RESTORE PREVIOUSLY
                                                    ;SAVED UNIT

                 ;NOW TO ALTER THE IN-CORE COPY OF THE PROTECTION TABLE

000700  004767            JSR     PC,FINDRV         ;IS THE HANDLER LOADED?
        177310
000704  103725            BCS     O.GOOD            ;NOPE...
000706  062701            ADD     #DXWPRO-DXLQE,R1  ;YES, ADD OFFSET FROM
        000006
                                                    ;ENTRY TO TABLE
000712  060001            ADD     R0,R1             ;ADD IN UNIT OFFSET
000714  116711            MOVB    O.WPF,(R1)        ;SET THE WRITE-PROTECT
        177724
                                                    ;STATUS
000720  000717            BR      O.GOOD
```

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:

```
5 000220                          .DRBEG  DX
  000220                 .ASECT
              000052     . = 52
                         .GLOBL   DXEND,DXINT
```

```
000052  001142          .WORD   <DXEND-DXSTRT>
                .IF B   <>
000054  000756          .WORD   DXDSIZE
                .IFF
                        .WORD
                .ENDC
                .IF B   <>
000056  102022          .WORD   DXSTS
                .IFF
                        .WORD
                .ENDC
000060  000007          .WORD   ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>
        000176  . = 176
000176  177170  .IIF DF DX$CSR, .WORD   DX$CSR
000000          .PSECT  DXDVR
000000          DXSTRT::
                .IF NB
                .GLOBL
                        .WORD   <-.>/2. -1 + ^O100000
                .IFF
                .IF NB  <>
                .IIF NE &3      .ERROR  ;ODD OR ILLEGAL VECTOR
                        .WORD   &^C3
                .IFF
                .IF DF  DX$VTB
                .GLOBL  DX$VTB
                        .WORD   <DX$VTB-.>/2. -1 + ^O100000
                .IFF
                .IIF NE DX$VEC&3 .ERROR DX$VEC ;ODD OR ILLEGAL VECTOR
000000  000264          .WORD   DX$VEC&^C3
                .ENDC
                .ENDC
                .ENDC
000002  000414          .WORD   DXINT-.,^O340
000004  000340
000006          DXSYS::
000006  000000  DXLQE:: .WORD   0
000010  000000  DXCQE:: .WORD   0
```

I/O Initiation Section

```
000012  000402          BR      DXENT           ;BRANCH AROUND
                                                ;PROTECTION TABLE
000014          DXWPRO:
        000001          .REPT   DXT$O+1
                        .BYTE   0,0
                        .ENDR
000014     000          .BYTE   0,0
000015     000

                        .IF NE ERL$G
000016  000000  SCSFLG: .WORD   0               ;:SUCCESSFUL LOGGING
                                                ;FLAG (DEFAULT=YES)
                                                ;=0 - LOG SUCCESSES,
                                                ;<>0 - DON'T LOG
                                                ;SUCCESSES
                        .ENDC

000020          DXENT:
                .IF NE  MMG$T
000020  013704          MOV     @#SYSPTR,R4     ;R4 -> MONITOR BASE
        000054
000024  016427          MOV     P1EXT(R4),(PC)+ ;GET ADDRESS OF EX-
        000432
                                                ;TERNALIZATION ROUTINE
000030  000432  $P1EXT: .WORD   P1EXT           ;POINTER TO EX-
                                                ;TERNALIZATION ROUTINE
                .ENDC ;NE MMG$T

000032  012727          MOV     (PC)+,(PC)+     ;INITIALIZE RETRY COUNT
000034  000010  DRETRY: .WORD   RETRY           ; :RETRY MAXIMUM
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.

```
000020  016703          MOV     DXCQE,R3;GET POINTER TO QUEUE
        177764
                                                ;ELEMENT
000024  012305          MOV     (R3)+,R5        ;GET BLOCK NUMBER
000026  012704          MOV     #CSRD!CSGO,R4   ;GUESS THAT CONTROLLER
        000007
                                                ;FUNCTION IS READ
000032  112301          MOVB    (R3)+,R1        ;PICK UP SPECIAL FUNCTION
                                                ;CODE (SIGN EXTENDED)
000034  112300          MOVB    (R3)+,R0        ;PICK UP THE UNIT NUMBER
000036  106200          ASRB    R0              ;SHIFT IT TO CHECK FOR
                                                ;ODD UNIT
000040  103002          BCC     1$              ;BRANCH IF EVEN UNIT
000042  052704          BIS     #CSUNIT,R4      ;SELECT ODD UNIT FOR
        000020
                                                ;TRANSFER
000046          1$:
                .IF EQ  DXT$0   ;ONE CONTROLLER
000046  132700          BITB    #6/2,R0         ;ANY UNITS BUT 0 OR 1?
        000003
000052  001153          BNE     RXERR           ;BRANCH IF YES, ERROR
                .IFF
                        MOV     (PC)+,-(SP)     ;ASSUME FIRST DX
                                                ;CONTROLLER
                DXCSR = .
                          .WORD DX$CSR
                        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$CS2
                2$:     MOV     (SP)+,RXCSA
                        ASRB    R0              ;BUT WAS IT UNIT 4 TO 7?
                        BCS     RXERR           ;ERROR IF SO
                .ENDC ;EQ DXT$0
000054  012300          MOV     (R3)+,R0        ;GET THE USER'S BUFFER
                                                ;ADDRESS
000056  012302          MOV     (R3)+,R2        ;GET WORD COUNT
000060  100002          BPL     3$              ;POSITIVE MEANS READ,
                                                ;SO ALL SET UP

                ; HERE TO CHECK IF UNIT IS WRITE-PROTECTED

000102  005046          CLR     -(SP)           ;SET TO GET UNIT
000104  156316          BISB    Q.UNIT-Q.COMP(R3),(SP) ;GET IT (PLUS
        177773
                                                ;OTHER CRUFT
000110  042716          BIC     #<^C3>,(SP)     ;WHICH WE DISCARD NOW)
        177774
000114  060716          ADD     PC,(SP)         ;ADD ADDRESS OF WRITE-
                                                ;PROTECT TABLE
000116  062716          ADD     #DXWPRO-.,(SP)  ;TO UNIT OFFSET
        177676
000122  105736          TSTB    @(SP)+          ;CHECK UNIT WRITE STATUS
000124  001143          BNE     RXERR           ;IT'S WRITE-PROTECTED,
                                                ;USER CAN'T DO THIS
000062  124444          CMPB    -(R4),-(R4)     ;CHANGE CSRD (3*2) TO
                                                ;CSWRT (2*2) FOR WRITE
```

Ensure that a write equals a read code minus 2:

```
000064          .ASSUME CSWRT   EQ      CSRD-2
                .IF     EQ      <CSWRT>-<CSRD-2>
                .IFF
                .ERROR  ;"CSWRT EQ CSRD-2" IS NOT TRUE
                .ENDC
000064  005402          NEG     R2              ;AND MAKE WORD COUNT
                                                ;POSITIVE
000066  006301  3$:     ASL     R1              ;DOUBLE THE SPECIAL
                                                ;FUNCTION CODE
000070  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:

```
000072  066104          ADD     CHGTBL-.(R1),R4 ;MODIFY THE CODE, SET
        001006
                                                ;SIGN BIT IF SPFUN
000076  010467          MOV     R4,RXFUN2       ;SAVE THE FUNCTION CODE
        000332
                                                ;AND SPFUN FLAG
000102  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.

```
000104  110267          MOVB    R2,FILLCT       ;SAVE WORD COUNT IN CASE
        000507
                                                ;WE HAVE TO FILL
000110  105367          DECB    FILLCT          ;EXTRA SECTORS ON WRITE
        000503
000114  006302          ASL     R2              ;MAKE WORD COUNT UNSIGNED
                                                ;BYTE COUNT
000116  006305          ASL     R5              ;NORMAL READ/WRITE, COM-
                                                ;PUTE REAL SECTOR NUMBER
000120  006305          ASL     R5              ;AS BLOCK*4
000122  012704          MOV     (PC)+,R4        ;LOOP COUNT FOR 8 BIT
                                                ;DIVISION
000124     371           .BYTE  -7,-26.         ;COUNT BECOMES 1, -26 IN
000125     346
                                                ;HIGH BYTE FOR LATER
000126  022705  4$:     CMP     #26.200,R5      ;DOES 26 GO INTO DIVIDEND?
        006400
000132  101002          BHI     5$              ;BRANCH IF NOT, C CLEAR
000134  062705          ADD     #-26.*200,R5    ;SUBTRACT 26 FROM
        171400
                                                ;DIVIDEND, SET C
000140  006105  5$:     ROL     R5              ;SHIFT DIVIDEND AND
                                                ;QUOTIENT
000142  105204          INCB    R4              ;DECREMENT LOOP COUNT
000144  003770          BLE     4$              ;BRANCH UNTIL DIVIDE DONE
000146  110501          MOVB    R5,R1           ;COPY TRACK NUMBER 0:75,
                                                ;ZERO EXTEND
000150  060405          ADD     R4,R5           ;BUMP TRACK TO 1-76,
                                                ;MAKE SECTOR<0
000152  010104          MOV     R1,R4           ;COPY TRACK NUMBER
000154  006301          ASL     R1              ;MULTIPLY
000156  060401          ADD     R4,R1           ; BY
000160  006301          ASL     R1              ;  6
000162  162701  6$:     SUB     #26.,R1 ;REDUCE TRACK NUMBER * 6
        000032
                                                ; MOD 26
000166  003375          BGT     6$              ; TO FIND OFFSET FOR
                                                ; THIS TRACK, -26:0
000170  010167          MOV     R1,TRKOFF       ;SAVE IT
        000144
000174  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.

```
000176  000305  7$:     SWAB    R5              ;PUT PHYSICAL SECTOR IN
                                                ; HIGH BYTE
000200  150205          BISB    R2,R5           ; AND PHYSICAL TRACK IN
                                                ; LOW BYTE
```

```
000202  012702          MOV     #128.,R2        ;SET THE BYTE COUNT TO
        000200
                                                ;128
                .IF EQ  MMG$T
                        CLR     (R0)+           ;CLEAR DELETED DATA FLAG
                                                ;WORD, BUMP USER ADDR
                .IFF
000206  016704          MOV     DXCQE,R4        ;POINT TO QUEUE ELEMENT
        177576
                                                ;AT Q.BLKN
000212  005046          CLR     -(SP)           ;STACK A ZERO AND STORE
                                                ; IT IN FIRST WORD OF
000214  004777          JSR     PC,@$PTWRD      ; BUFFER. NOTE THAT
        000710
                                                ; Q.BUFF GETS BUMPED BY 2
000220  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 XM systems:

```
000222  010027  8$:     MOV     R0,(PC)+        ;SAVE BUFFER ADDRESS
000224  000000  BUFRAD:  .WORD  0               ; : USER VIRTUAL BUFFER
                                                ;   ADDRESS
000226  010567          MOV     R5,TRACK        ;SAVE IT FOR STARTING I/O
        000140
000232  010227          MOV     R2,(PC)+        ;  AND BYTE COUNT.
000234  000000  BYTCNT:  .WORD  0               ; : BYTE COUNT FOR
                                                ;   TRANSFER
                .IF NE  MMG$T
000236  005723          TST     (R3)+           ;SKIP THE COMPLETION
                                                ;ROUTINE ADDRESS
000304  011367          MOV     @R3,PARVAL      ;SAVE THE PAR1 VALUE
        000646
                                                ;FOR MAPPING USER BUFFER

                .ENDC ;NE MMG$T
000244                  .BR     RXINIT          ;GO TO FORK LEVEL AND
                .IF DF  RXINIT
                .IF NE  .-<RXINIT>
                .ERROR  ;NOT AT LOCATION "RXINIT"
                .ENDC
                .ENDC
                                                ;START IT UP

                .DSABL  LSB

                .SBTTL  START TRANSFER OR RETRY

                .ENABL  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$:

```
000244  012767  RXINIT: MOV     #100000,RXIRTN  ;SET RETURN AFTER INITIAL
        177600
        000204
                                                ;INTERRUPT
000252  016704          MOV     RXCSA,R4        ;ENSURE THAT WE POINT TO
        000166
                                                ;THE CSR
000256  000446          BR      RXIENB          ;GO INTERRUPT, RETURN TO
                                                ;1$ LATER

000264  032700  1$:     BIT     #2,R0           ;READ OR WRITE FUNCTION?
        000002
000270  001010          BNE     3$              ;IF READ, GO FILL THE
                                                ;SILO FROM DISK
000272  004067  2$:     JSR     R0,SILOFE       ;WRITE, LOAD THE SILO
        000476
                                                ;FROM THE USER BUFFER
```

Parameters for SILOFE routine:

```
000276  000001          .WORD   CSFBUF!CSGO       ;FILL BUFFER COMMAND
000300  112215          MOVB    (R2)+,@R5         ;MOVB TO BE PLACED
                                                  ;IN-LINE IN SILOFE
000302  010115          MOV     R1,@R5            ;ZERO-FILL INSTRUCTION
                                                  ;FOR SHORT WRITES
```

The following routine changes a sector number to an interleaved sector number:

```
000312  116702  3$:     MOVB    SECTOR,R2         ;GET THE SECTOR NUMBER
        000055
000316  003014          BGT     5$                ;POSITIVE MEANS SPFUN,
                                                  ;DON'T INTERLEAVE
000320  162702          SUB     #-14.,R2          ;ADD 14 TO DO INTER-
        177762
                                                  ;LEAVING
000324  003003          BGT     4$                ;IF > 0, MAP -13:-1 TO
                                                  ; 2:26, NOTE C=0
000326  062702          ADD     #12.,R2           ; ELSE MAP -26:-14 TO
        000014
                                                  ; 1:25
000332  000261          SEC                       ;ADD 1 WHEN DOUBLING
000334  006102  4$:     ROL     R2                ;DOUBLE AND INTERLEAVE,
                                                  ;SECTOR 1:26
000336  062702          ADD     (PC)+,R2          ;ADD IN THE TRACK OFFSET,
                                                  ;SECTOR -25:26
000340  000000  TRKOFF: .WORD   0                 ; : TRACK OFFSET =
                                                  ; TRACK*6 MOD 26,
                                                  ; RANGE -26:0
000342  003002          BGT     5$                ;NO MODULUS PROBLEMS
000344  062702          ADD     #26.,R2           ;FIX TO PUT SECTOR IN
        000032
                                                  ;1:26 RANGE
000350  010014  5$:     MOV     R0,@R4            ;SET THE FUNCTION IN THE
                                                  ;FLOPPY CONTROLLER
000352  105714  6$:     TSTB    @R4               ;WAIT FOR
000354  001776          BEQ     6$                ;TRANSFER READY
000356  100146          BPL     RXRTRY            ;TRANSFER DONE WITHOUT
                                                  ;TRANSFER READY, ERROR
000360  010215          MOV     R2,@R5            ;SET SECTOR NUMBER
000362  105714  7$:     TSTB    @R4               ;WAIT AGAIN FOR
000364  001776          BEQ     7$                ;TRANSFER READY
000366  100142          BPL     RXRTRY            ;TRANSFER DONE WITHOUT
                                                  ;TRANSFER READY, ERROR
000370  112715          MOVB    (PC)+,@R5         ;SET THE TRACK NUMBER
000372     000  TRACK:  .BYTE   0                 ;TRACK NUMBER
000373     000  SECTOR: .BYTE   0                 ;SECTOR NUMBER, KEPT < 0
                                                  ;UNLESS SPFUN
```

Start the operation and return to the monitor:

```
000374  052714  RXIENB: BIS     #CSINT,@R4        ;SET IE TO CAUSE AN
        000100
                                                  ;INTERRUPT WHEN DONE
                                                  ;IS UP
000400  000207          RTS     PC                ;RETURN, WE'LL BE BACK
                                                  ;WITH AN INTERRUPT

000402  016704  RXERR:  MOV     DXCQE,R4          ;R4 -> CURRENT QUEUE
        177402
                                                  ;ELEMENT
000406  052754          BIS     #HDERR$,@-(R4)    ;SET HARD ERROR IN CSW
        000001
000412  000511          BR      13$               ;EXIT ON HARD ERROR
```

Interrupt Service Section

The .DRAST macro:

```
   000414                    .DRAST  DX,5,RXABRT       ;AST ENTRY POINT TABLE
                     .GLOBL  $INPTR
                     .IF B <RXABRT>
                             RTS     %7
                     .IFF
```

The abort entry point:

```
   000414  000521            BR      RXABRT
                     .ENDC
   000416  004577    DXINT:: JSR     %5,@$INPTR
           000514
   000422  000100            .WORD   ^C<5*^040>&^0340
```

Drop to fork level rather than device priority because the routine is lengthy and it needs all the registers.

```
   000424                    .FORK   DXFBLK            ;REQUEST FORK LEVEL
   000424  004577            JSR     %5,@$FKPTR
           000510
   000430  000452            .WORD   DXFBLK - .
                                                       ;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.)

```
   000432  012700            MOV     (PC)+,R0;GET A VERY USEFUL FLAG
                                                       ;WORD
   000434  000000  RXFUN2:   .WORD   0                 ; : READ OR WRITE COMMAND
                                                       ;   ON CORRECT UNIT
   000436  012703            MOV     #128.,R3          ;LOAD A HANDY CONSTANT
           000200
   000442  012704            MOV     (PC)+,R4          ;GET ADDRESS OF RX
                                                       ;CONTROLLER
   000444  177170  RXCSA:    .WORD   DX$CSR            ; : ADDRESS OF CONTROLLER
   000446  010405            MOV     R4,R5             ;POINT R5 TO RX DATA
                                                       ;BUFFER
   000450  005725            TST     (R5)+             ;CHECK FOR ERROR, R5 ->
                                                       ;DX REGISTER WITH ERROR
   000452  100510            BMI     RXRTRY            ;ERROR, PROCESS IT
   000506  006327            ASL     (PC)+             ;NO ERROR, DISPATCH
                                                       ;AFTER INTERRUPT
   000510  000000  RXIRTN:   .WORD   0                 ;OFFSET TO INTERRUPT
                                                       ;CONTINUATION
   000512  103704            BCS     1$                ;FIRST INTERRUPT,
                                                       ;START I/O
   000514  032700            BIT     #2,R0             ;READ OR WRITE?
           000002
   000520  001437            BEQ     10$               ;WRITE, DON'T EMPTY
                                                       ;SILO

   000460  005700            TST     R0                ;READ, IS THIS A SPECIAL
                                                       ;FUNCTION?
```

The silo is a 128-byte (decimal) storage area in the diskette logic.

```
   000462  100016            BPL     9$                ;NO, SIMPLY EMPTY THE
                                                       ;SILO THAT WAS JUST READ
   000464  032715            BIT     #ESDD,@R5         ;IF SPFUN READ, IS
           000100
```

```
                                              ;DELETED DATA FLAG
                                              ;PRESENT?
  000470  001413          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 .SPFUN read.

```
                  .IF EQ  MMG$T
                          MOV     BUFRAD,R2        ;GET ADDRESS OF USER
                                                   ;BUFFER AREA
                          INC     -(R2)            ;SET FLAG WORD TO 1 TO
                                                   ;INDICATE DELETED DATA
                  .IFF
  000472  010401          MOV     R4,R1            ;SAVE R4
  000474  016704          MOV     DXCQE,R4         ;POINT TO QUEUE ELEMENT
          177310
  000500  012746          MOV     #1,-(SP)         ;STACK A 1 TO PUT INTO
          000001
                                                   ;FLAG WORD
  000546  162764          SUB     #2,Q$BUFF(R4)    ;MOVE BUFFER POINTER
          000002
          000004
                                                   ;BACK TO FIRST WORD
  000554  026427          CMP     Q$BUFF(R4),#20000;POINTER OUT OF THIS
          000004
          020000
                                                   ;PAR'S RANGE?
  000562  103006          BHIS    85$              ;NOPE...
  000564  062764          ADD     #20000,Q$BUFF(R4);YES, GET IT BACK
          020000
          000004
                                                   ;IN RANGE
  000572  162764          SUB     #200,Q$PAR(R4)   ;IN THE PREVIOUS PAR.
          000200
          000012
  000512  004777  85$:    JSR     PC,@$PTWRD       ;STORE IN 1ST WORD.
          000412
                                                   ;Q.BUFF IS AGAIN
                                                   ;ORIGINAL+2
  000516  010104          MOV     R1,R4            ;RESTORE R4.
                  .ENDC ;EQ MMG$T
  000520  004067  9$:     JSR     R0,SILOFE        ;FOR READ, MOVE THE DATA
          000250
                                                   ; FROM SILO TO BUFFER
  000524  000003           .WORD  CSEBUF!CSGO      ; EMPTY BUFFER COMMAND
  000526  111522           MOVB   @R5,(R2)+        ; MOVB TO BE PLACED IN
                                                   ; LINE IN SILOFE
  000530  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.

```
  000532  105267  10$:    INCB    SECTOR           ;RETURN HERE AFTER WRITES.
          177635
                                                   ;BUMP SECTOR NUMBER
  000536  001012          BNE     11$              ;NOT OFF END OF TRACK YET
  000540  062767          ADD     #-26.*400+1,TRACK ;RESET SECTOR, BUMP TO
          163001
          177624
                                                   ;NEXT TRACK
  000546  062767          ADD     #6,TRKOFF        ;BUMP TRACK OFFSET VALUE
          000006
          177564
  000554  003403          BLE     11$              ;OK IF STILL IN RANGE
                                                   ;-25:0
  000556  162767          SUB     #26.,TRKOFF      ;RESET TO PROPER RANGE
          000032
          177554
                                                   ;MOD 26
```

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.

```
000564          11$:
                .IF EQ  MMG$T
                        ADD     R3,BUFRAD       ;UPDATE BUFFER ADDRESS
                .IFF
000564  062767          ADD     #2,PARVAL       ;CHANGE MAP TO BUMP
        000002
        177450
                                                ;ADDRESS FOR NEXT TIME
                .ENDC ;EQ MMG$T
000572  160367          SUB     R3,BYTCNT       ;REDUCE THE AMOUNT LEFT
        177436
                                                ;TO TRANSFER
000576  101230          BHI     1$              ;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.

```
000600  005067          CLR     BYTCNT          ;FIX BYTE COUNT SO THAT
        177430
                                                ;WRITES ARE ALL 0-FILLS
000604  032700          BIT     #2!SPFUNC,R0    ;READ OR SPECIAL FUNCTION
        100002
                                                ;OPERATION?
000610  001004          BNE     12$             ;IF SO, NO ZERO-FILLING,
                                                ;SO WE'RE DONE
```

The operation was a write. The routine may need to zero-fill up to three sectors (see FILLCT above).

```
000612  062727          ADD     #040000,(PC)+   ;CHECK ORIGINAL WORD
        040000
                                                ; COUNT FOR # OF SECTORS
000616     000          .BYTE   0               ;   FILLER
000617     000  FILLCT: .BYTE   0               ; : ORIGINAL WORD COUNT
                                                ; LOW BYTE IN HIGH BYTE
000620  103224          BCC     2$              ;YES, LOOP FOR ZERO-
                                                ;FILLING ON WRITE
000622          12$:                            ;AHH, A SUCCESSFUL
                                                ;TRANSFER IS DONE
                .IF NE  ERL$G
```

Log a successful transfer:

```
000710  005767          TST     SCSFLG          ;LOGGING SUCCESSFUL
        177102
                                                ;TRANSFERS?
000714  001006          BNE     13$             ;NOPE...
000622  012704          MOV     #DX$COD*400+377,R4 ;SET UP R4 = ID/-1
        011377
000626  016705          MOV     DXCQE,R5        ;AND R5 -> CURRENT
        177156
                                                ;QUEUE ELEMENT
000632  004777          JSR     PC,@$ELPTR      ;CALL ERROR LOGGER TO
        000274
                                                ;REPORT SUCCESS
                .ENDC ;EQ ERL$G
000636  005077  13$:    CLR     @RXCSA          ;DISABLE FLOPPY
        177602
                                                ;INTERRUPTS
```

I/O Completion Section

The .DRFIN macro:

```
 000642              14$:     .DRFIN  DX              ;GO TO I/O COMPLETION
                     .GLOBL   DXCQE
    000642  010704            MOV     %7,%4
    000644  062704            ADD     #DXCQE-.,%4
            177144
    000650  013705            MOV     @#^054,%5
            000054
    000654  000175            JMP     @^0270(5)
            000270
```

The abort routine:

```
                     ; ABORT TRANSFER
 000660     012777   RXABRT: MOV     #CSINIT,@RXCSA  ;PERFORM AN RX11
            040000
            177556
                                                     ;INITIALIZE
 000666     005067           CLR     DXFBLK+2        ;CLEAR FORK BLOCK TO
            000212
                                                     ;AVOID A DISPATCH
```

Go to .DRFIN (no error logging):

```
 000672     000763           BR      14$             ;AND FINISH UP THIS I/O
                     .DSABL  LSB
                     .IF NE  DXT$O
                             .DRVTB  DX,DX$VEC,DXINT
                             .DRVTB  ,DX$VC2,DXINT
                     .ENDC ;NE DXT$O
```

If there was an error, log it:

```
                     ; TRANSFER ERROR HANDLING
 000674              RXRTRY:
                     .IF NE  ERL$G
 000674     010703           MOV     PC,R3
 000676     062703           ADD     #DXRBUF-.,R3    ;R3 -> LOCATION TO STORE
            000214
                                                     ;REGISTER INFO.
 000702     010302           MOV     R3,R2           ;SAVE IN R2 FOR LATER
 000704     011423           MOV     @R4,(R3)+       ;STORE RXCS
 000706     011523           MOV     @R5,(R3)+       ;STORE STATUS RXES
 000710     012714           MOV     #CSMAIN!CSGO,@R4 ;READ ERROR REGISTER
            000017
                                                     ;(NO INTERRUPTS)
 000714     032714   1$:     BIT     #CSDONE,@R4     ;WAIT FOR READ COMPLETION
            000040
 000720     001775           BEQ     1$
 000722     011513           MOV     @R5,@R3         ;STORE IN BUFFER
 001020     016703           MOV     DRETRY,R3
            177010
 001024     000303           SWAB    R3
 001026     062703           ADD     #DXNREG,R3      ;R3 = MAX RETRIES/#
            000003
                                                     ;OF REGS
 000730     012704           MOV     #DX$COD*400,R4  ;R4 = DEVICE ID IN HIGH
            011000
                                                     ;BYTE
 000734     156704           BISB    RXTRY,R4        ;AND CURRENT RETRY
            177056
                                                     ;COUNT IN LOW BYTE
 000740     105304           DECB    R4              ; -1 FOR THIS ERROR
```

```
000742  016705         MOV     DXCQE,R5          ;R5 -> QUEUE ELEMENT
        177042
000746  004777         JSR     PC,@$ELPTR        ;CALL ERROR LOGGER
        000160
000752  016704         MOV     RXCSA,R4          ;RESTORE R4 = RXCS
        177466
                                                 ;ADDRESS
               .ENDC ;NE ERL$G
```

See if a retry is allowed:

```
000756  005367         DEC     RXTRY             ;SHOULD WE TRY AGAIN?
        177034
001064  003002         BGT     2$                ;YES
001066  000167         JMP     RXERR             ;NOPE, REPORT AN ERROR
        177342

000764  012714  2$:    MOV     #CSINIT,@R4       ;START A RECALIBRATE
        040000
```

Retry the operation:

```
000770  000167         JMP     RXINIT            ;EXIT THROUGH START
        177250
                                                 ;OPERATION CODE
               .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
               ;-
```

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.

```
000774  012014  SILOFE: MOV    (R0)+,@R4         ;INITIATE FILL OR EMPTY
                                                 ;BUFFER COMMAND
000776  012067         MOV     (R0)+,3$          ;PUT CORRECT MOV
        000042
                                                 ;INSTRUCTION IN FOR
                                                 ;FILL/EMPTY
001002  012067         MOV     (R0)+,5$          ;PUT IN INSTRUCTION TO
        000056
                                                 ;SLUFF DATA
001006  016701         MOV     BYTCNT,R1         ;GET BYTE COUNT
        177222
```

```
001012  001421          BEQ     4$              ;IF ZERO, WE ARE SEEKING
                                                ;OR ZERO FILLING
001026  020103          CMP     R1,R3           ;IS THE BYTE COUNT
                                                ;<= 128?
001030  101401          BLOS    1$              ;OK IF SO
001032  010301          MOV     R3,R1           ;DO ONLY 128 BYTES AT A
                                                ;TIME
001034  016702  1$:     MOV     BUFRAD,R2       ;GET USER VIRTUAL BUFFER
        177164
                                                ;ADDRESS IN R2
```

The following section of code can be executed in two different ways. If the handler is assembled for SJ or FB, the code between the tags 2$: and PARVAL: is simply executed in-line. If the handler is assembled for XM, the JSR to P1EXT and the word PARVAL are included. In this situation, the routine P1EXT copies the code between 2$: 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$T
001134  004077          JSR     R0,@$P1EXT      ;If XM, let the monitor
        176670
                                                ;execute the following
                                                ;code.
001140  000016          .WORD   PARVAL-.        ;Number of instructions
                                                ;in bytes plus 2.
                .ENDC ;NE MMG$T
001142  105714  2$:     TSTB    @R4             ;*EXT* TRY FOR THE TRDY
001144  100376          BPL     2$              ;*EXT* TRANSFER READY
001146  000000  3$:     HALT                    ;*EXT* INSTRUCTION TO
                                                ; MOV OR SLUFF DATA FROM
001150  105714          TSTB    @R4             ;*EXT* TOUCH THE CSR
                                                ; TO GET IT READY
001152  105301          DECB    R1              ;*EXT* CHECK FOR COUNT
                                                ; DONE
001154  001372          BNE     2$              ;*EXT* STILL MORE TO
                                                ; TRANSFER
                .IF NE  MMG$T
001156  000000  PARVAL: .WORD   0               ;using this value for
                                                ;the PAR 1 bias.
                .ENDC ;NE MMG$T
```

The slough routine:

```
001056  105714  4$:     TSTB    @R4             ;WAIT FOR TRANSFER READY
                                                ;OR TRANSFER DONE
001060  003003          BGT     6$              ;TDONE UP WITH NO TRDY,
                                                ;SO ALL DONE
001062  001775          BEQ     4$              ;LOOP
001064  000000  5$:     HALT                    ;TRANSFER READY, SO
                                                ;SLUFF DATA
001066  000773          BR      4$              ;LOOP TO SLUFF MORE

001070  000200  6$:     RTS     R0              ;RETURN
                        .DSABL  LSB
                .SBTTL  TABLES, FORK BLOCK, END OF DRIVER

                ; CHANGES TO CSR CODE FOR SPECIAL FUNCTIONS

001072  100006          .WORD   CSWRTD-CSRD+SPFUNC ;375: READ+GO ->
                                                   ;WRITE DELETED+GO
001074  077776          .WORD   CSWRT-CSRD+SPFUNC  ;376: READ+GO ->
                                                   ;WRITE+GO
001076  100000          .WORD   CSRD-CSRD+SPFUNC   ;377: READ+GO ->
                                                   ;READ+GO
001100  000000  CHGTBL: .WORD   0               ;READ/WRITE STAY THE
                                                ;SAME
```

```
   001102  000000  DXFBLK: .WORD   0,0,0,0          ;DX FORK QUEUE ELEMENT
   001104  000000
   001106  000000
   001110  000000

                   .IF NE  ERL$G
   001112          DXRBUF: .BLKW   DXNREG           ;ERROR LOG STORAGE
                   .ENDC ;NE ERL$G
```

Primary Driver

```
                   .SBTTL  BOOTSTRAP DRIVER

```

The .DRBOT macro:

```
   001120                  .DRBOT  DX,BOOT1,READ
```

Termination Section

The .DREND macro generated by .DRBOT:

```
   001120                  .DREND  DX
   001120          .PSECT  DXDVR
                   .IIF NDF DX$END, DX$END:
                   .IF EQ  .-DX$END
   001120          DX$END::
                   .IF NE MMG$T
   001120  000000  $RLPTR:: .WORD  0
   001122  000000  $MPPTR:: .WORD  0
   001124  000000  $GTBYT:: .WORD  0
   001126  000000  $PTBYT:: .WORD  0
   001130  000000  $PTWRD:: .WORD  0
                   .ENDC
                   .IF NE ERL$G
   001132  000000  $ELPTR:: .WORD  0
                   .ENDC
                   .IF NE TIM$IT
   001134  000000  $TIMIT:: .WORD  0
                   .ENDC
   001136  000000  $INPTR:: .WORD  0
   001140  000000  $FKPTR:: .WORD  0
                   .GLOBL  DXSTRT
```

The following line marks the end of the loadable portion of the handler. It is used to determine the handler's length.

```
           001142' DXEND == .
                   .ENDC
                   .IIF NDF TPS, TPS=177564
                   .IIF NDF TPB, TPB=177566
           000012  LF=12
           000015  CR=15
           001000  B$BOOT=1000
           004716  B$DEVN=4716
           004722  B$DEVU=4722
           004730  B$READ=4730
                   .IF EQ  MMG$T
                   B$DNAM=^RDX
                   .IFF
           016330  B$DNAM=^RDXX
                   .ENDC
   000200          .ASECT
           000062  .=62
```

```
 000062  000000'          .WORD   DXBOOT,DXBEND-DXBOOT,READ-DXBOOT
 000064  001000
 000066  000224
 000000          .PSECT  DXBOOT
 000000  000240  DXBOOT::NOP
 000002  000414          BR      BOOT1

         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
         000056
         000066
                                                   ;FUNCTION FOR CORRECT
                                                   ;UNIT
 000042  011706  REETRY: MOV     @PC,SP            ;INIT SP WITH NEXT
                                                   ;INSTRUCTION
 000044  012702          MOV     #200,R2           ;AREA TO READ IN NEXT
         000200
                                                   ;PART OF BOOT
 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
         000003
                                                   ;FUNCTION INTO RXCS
         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  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
        000300
        004730
                                                 ;WILL GO TO READ1
000246  012737           MOV     #TRWAIT-DXBOOT,@#20 ;LETS HANDLE ERRORS
        000416
        000020
                                                 ;DIFFERENTLY
000254  005037           CLR     @#JSW           ;CLEAR JSW SINCE THE DX
        000044
                                                 ;BOOT IN SYSCOM AREA
000260  005767           TST     HRDBOT          ;DID WE REACH HERE VIA A
        000346
                                                 ;HARDWARE BOOT?
000264  001405           BEQ     READ1           ;YES, DON'T SET UP UNIT
                                                 ;NUMBER
000266  013703           MOV     @#B$DEVU,R3     ;NO, SET UP UNIT NUMBER
        004722
000272  116367           MOVB    UNITRD-DXBOOT(R3),RDCMD ;STORE UNIT
        000056
        177630
                                                 ;NUMBER
000300  006300  READ1:   ASL     R0              ;CONVERT BLOCK TO LOGICAL
                                                 ;SECTOR
000302  006300           ASL     R0              ;LSN=BLOCK*4
000304  006301           ASL     R1              ;MAKE WORD COUNT BYTE
                                                 ;COUNT
000306  010046  1$:      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$:      SUB     #23.,R3         ;PERFORM MAGIC TRACK
        000027
                                                 ;DISPLACEMENT
000324  005200  3$:      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
        177762
                                                 ;1-13
000340  006103           ROL     R3              ;PERFORM 2:1 INTERLEAVE
000342  162703  4$:      SUB     #26.,R3         ;ADJUST SECTOR INTO
        000032
                                                 ;RANGE -1,-26
000346  100375           BPL     4$              ;(DIVIDE FOR REMAINDER
                                                 ;ONLY)
000350  062703           ADD     #27.,R3         ;NOW PUT SECTOR INTO
        000033
                                                 ;RANGE 1-26
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
        000001
                                                ;WD CNT
                                                ;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
        000002
                                                ;BOOT
000620  012701          MOV     #<4*400>,R1     ;EVERY BLOCK BUT THE ONE
        002000
                                                ;WE ARE IN
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          JSR     PC,READ         ;GO READ IT IN
        177364
000640  012737          MOV     #READ1-DXBOOT,@#B$READ ;STORE START
        000300
        004730
                                                ;LOCATION FOR READ
                                                ;ROUTINE
000646  012737          MOV     #B$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
001142          .PSECT  DXDVR
                .IIF NDF DX$END, DX$END:
                .IF EQ  .-DX$END
                DX$END::

                .IF NE MMG$T
                $RLPTR:: .WORD  0
                $MPPTR:: .WORD  0
                $GTBYT:: .WORD  0
                $PTBYT:: .WORD  0
                $PTWRD:: .WORD  0
                .ENDC
                .IF NE ERL$G
                $ELPTR:: .WORD  0
                .ENDC
                .IF NE TIM$IT
                $TIMIT:: .WORD  0
                .ENDC
                $INPTR:: .WORD  0
                $FKPTR:: .WORD  0
                .GLOBL  DXSTRT
                DXEND == .
                .IFF
```

```
000664                  .PSECT  DXBOOT
                        .IIF LT <DXBOOT-.+664>, .ERROR  ;PRIMARY BOOT TOO LARGE

                000664'  .       = DXBOOT+664
000664  004167  BIOERR: JSR     R1,REPORT
        000002
000670  000766           .WORD  IOERR-DXBOOT

000672  012700  REPORT: MOV     #BOOTF-DXBOOT,R0
        000746
000676  004167          JSR     R1,REP
        000030
000702  012100          MOV     (R1)+,R0
000704  004167          JSR     R1,REP
        000022
000710  012700          MOV     #CRLFLF-DXBOOT,R0
        000762
000714  004167          JSR     R1,REP
        000012
000720  000005          RESET
000722  000000          HALT
000724  000776          BR      .-2
000726  112037  REPOR:  MOVB    (R0)+,@#TPB
        177566
000732  105737  REP:    TSTB    @#TPS
        177564
000736  100375          BPL     REP
000740  105710          TSTB    @R0
000742  001371          BNE     REPOR
000744  000201          RTS     R1
000746     015  BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"<200>
000762     015  CRLFLF: .ASCIZ  <CR><LF><LF>
000766     111  IOERR:  .ASCIZ  "I/O ERROR"
                        .EVEN
001000          DXBEND::
                        .ENDC

        000001  .END
```

Symbol table

```
ABTIO$= 001000          DX$CSR= 177170 G        Q.WCNT= 000012
BAREA   000262          DX$CS2= 177174 G        RDCMD   000130R     003
BIOERR  000664R     003 DX$END  001222RG    002 RDX     000150R     003
BOOT    000606R     003 DX$VC2= 000270 G        READ    000224R     003
BOOTF   000746R     003 DX$VEC= 000264 G        READF   000370R     003
BOOT1   000034R     003 ELDX  = 000007 G        READFX  000424R     003
BOOT2   000200R     003 EOF$  = 020000          READS   000120R     003
BOTCSR  000122R     003 ERL$G = 000001          READ1   000300R     003
BROFFS= 000220          ESCRC = 000001          REETRY  000042R     003
BTCSR = 002366          ESDD  = 000100          REP     000732R     003
BUFRAD  000270R     002 ESDRY = 000200 G        REPOR   000726R     003
BYTCNT  000300R     002 ESID  = 000004          REPORT  000672R     003
B$BOOT= 001000          ESPAR = 000002          RETRY = 000010
B$DEVN= 004716          FILLCT  000705R     002 RONLY$= 040000
B$DEVU= 004722          FILST$= 100000          RTE$M = 000000
B$DNAM= 016330          FINDRV  000214          RTIRET  000076R     003
B$READ= 004730          HDERR$= 000001          RXABRT  000754R     002
B2$     000170R     003 HNDLR$= 004000          RXCSA   000476R     002
CHGTBL  001202R     002 HRDBOT  000632R     003 RXERR   000434R     002
CR    = 000015          IOERR   000766R     003 RXFUN2  000466R     002
CRLFLF  000762R     003 JSW   = 000044          RXIENB  000426R     002
CSDONE= 000040          LF    = 000012          RXINIT  000310R     002
CSEBUF= 000002          MMG$T = 000001          RXIRTN  000510R     002
CSERR = 100000          O.BAD   000562          RXRTRY  000770R     002
CSFBUF= 000000          O.CSR   000452          RXTRY   000036R     002
CSGO  = 000001          O.GOOD  000560          SCSFLG  000016R     002
CSINIT= 040000          O.RTRY  000606          SECTOR  000425R     002
CSINT = 000100          O.SUCC  000622          SILOFE  001102R     002
CSMAIN= 000016          O.VEC   000566          SPECL$= 010000
CSRD  = 000006          O.WP    000634          SPFUNC= 100000
CSRDST= 000012          O.WPF   000644          SPFUN$= 002000
CSRX02= 004000          PARVAL  001156R     002 SYSPTR= 000054
CSTR  = 000200          PNPTR = 000404          SYUNIT= 000274
CSUNIT= 000020          P1EXT = 000432          TIM$IT= 000001
CSWRT = 000004          Q$BLKN= 000000          TPB   = 177566
CSWRTD= 000014          Q$BUFF= 000004          TPS   = 177564
DRETRY  000034R     002 Q$COMP= 000010          TRACK   000424R     002
```

```
DXBEND  001000RG    003 Q$CSW = 177776        TRKOFF  000372R     002
DXBOOT  000000RG    003 Q$FUNC= 000002        TRWAIT  000416R     003
DXCQE   000010RG    002 Q$JNUM= 000003        UNITRD  000056R     003
DXDSIZ= 000756          Q$LINK= 177774        VARSZ$= 000400
DXEND = 001244RG    002 Q$PAR = 000012        WAIT    000070R     003
DXENT   000020R     002 Q$UNIT= 000003        WONLY$= 020000
DXFBLK  001204R     002 Q$WCNT= 000006        $ELPTR  001234RG    002
DXINT   000450RG    002 Q.BLKN= 000004        $FKPTR  001242RG    002
DXLQE   000006RG    002 Q.BUFF= 000010        $GTBYT  001226RG    002
DXNREG= 000003          Q.COMP= 000014        $INPTR  001240RG    002
DXRBUF  001214R     002 Q.CSW = 000002        $MPPTR  001224RG    002
DXSTRT  000000RG    002 Q.ELGH= 000024        $PTBYT  001230RG    002
DXSTS = 102022          Q.FUNC= 000006        $PTWRD  001232RG    002
DXSYS   000006RG    002 Q.JNUM= 000007        $P1EXT  000030R     002
DXT$O = 000000          Q.LINK= 000000        $RLPTR  001222RG    002
DXWPRO  000014R     002 Q.PAR = 000016        $TIMIT  001236RG    002
DX$COD= 000022          Q.UNIT= 000007        ...V2 = 000200


. ABS.  000722    000    (RW,I,GBL,ABS,OVR)
        000000    001    (RW,I,LCL,REL,CON)
DXOVR   001244    002    (RW,I,LCL,REL,CON)
DXBOOT  001000    003    (RW,I,LCL,REL,CON)

Errors detected:  0

*** Assembler statistics

Work  file  reads: 0
Work  file writes: 0
Size of work file: 10218 Words  ( 40 Pages)
Size of core pool: 15104 Words  ( 59 Pages)

Elapsed time: 00:00:22.00
,DX/L:ME:TTM=CND,DX
```

Figure A–3: PC Paper Tape Handler

```
PC      PAPER TAPE HANDLER    MACRO X05.03 Thursday 30-Sep-82 10:57
Table of contents

   2-      3      RT-11 HIGH SPEED PAPER TAPE PUNCH AND READER (PC11) HANDLER
   3-      1      EDIT HISTORY
   4-      1      MACROS AND DEFINITIONS
   5-      1      DRIVER ENTRY
                          ;Conditional File For Handler Examples
                          ;
                          ;CND.MAC
                          ;
                          ;SGW
                          ;
                          ;Assemble with handler .MAC file to enable
                          ;memory management support, device timeout,
                          ;and error logging.
                          ;
              000001              MMG$T=  1        ;Memory Management
              000001              ERL$G=  1        ;Error Logging
              000001              TIM$IT= 1        ;Device Timeout

                          .TITLE PC       PAPER TAPE HANDLER
                          .IDENT  /V05.00/
                  .SBTTL RT-11 HIGH SPEED PAPER TAPE PUNCH AND READER (PC11) HANDLER

                          ;                COPYRIGHT (c) 1982, 1983 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 WHICH  IS NOT
; SUPPLIED BY DIGITAL.
```

Preamble Section

```
.SBTTL  MACROS AND DEFINITIONS

.MCALL  .DRDEF

```

A value of 0 means punch and reader combined; 1 means reader only.

```
.IIF NDF PR11$X, PR11$X=0          ;UNLESS SPECIFIED, DO
                                   ;NOT GENERATE READER ONLY
.IIF NE PR11$X, PR11$X=1           ;FORCE NON-ZERO VALUE
                                   ;TO 1

```

The .DRDEF macro:

```
000000            .DRDEF  PC,7,<PR11$X*RONLY$>,0,177550,70
                  .MCALL  .DRAST,.DRBEG,.DRBOT,.DREND,.DRFIN,.DRSET
                  .MCALL  .DRVTB,.FORK,.QELDF
                  .IIF NDF RTE$M, RTE$M=0
                  .IIF NE RTE$M, RTE$M=1
                  .IIF NDF TIM$IT, TIM$IT=0
           000001 .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
           000001 .IIF NE ERL$G, ERL$G=1
                  .IIF NE TIM$IT, .MCALL  .TIMIO,.CTIMI
000000            .QELDF
                  .IIF NDF MMG$T, MMG$T=0
           000001 .IIF NE MMG$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-4
                  .ENDR
           177774 Q$LINK=Q.LINK-4
           177776 Q$CSW=Q.CSW-4
           000000 Q$BLKN=Q.BLKN-4
           000002 Q$FUNC=Q.FUNC-4
           000003 Q$JNUM=Q.JNUM-4
           000003 Q$UNIT=Q.UNIT-4
           000004 Q$BUFF=Q.BUFF-4
           000006 Q$WCNT=Q.WCNT-4
           000010 Q$COMP=Q.COMP-4
                  .IF EQ MMG$T
                  Q.ELGH=^O16
                  .IFF
           000016 Q.PAR=^O16
           000012 Q$PAR=^O12
           000024 Q.ELGH=^O24
                  .ENDC
           000001 HDERR$=1
           020000 EOF$=20000
```

```
           000400   VARSZ$=400
           001000   ABTIO$=1000
           002000   SPFUN$=2000
           004000   HNDLR$=4000
           010000   SPECL$=10000
           020000   WONLY$=20000
           040000   RONLY$=40000
           100000   FILST$=100000
           000000   PCDSIZ=0
           000007   PC$COD=7
           000007   PCSTS=<7>!<PR11$X*RONLY$>
                    .IIF NDF PC$CSR, PC$CSR=177550
                    .IIF NDF PC$VEC, PC$VEC=70
                    .GLOBL  PC$CSR,PC$VEC

   000000           .QELDF
                    .IIF NDF MMG$T, MMG$T=0
           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$'X=Q.'X-4
                    .ENDR
           177774   Q$LINK=Q.LINK-4
           177776   Q$CSW=Q.CSW-4
           000000   Q$BLKN=Q.BLKN-4
           000002   Q$FUNC=Q.FUNC-4
           000003   Q$JNUM=Q.JNUM-4
           000003   Q$UNIT=Q.UNIT-4
           000004   Q$BUFF=Q.BUFF-4
           000006   Q$WCNT=Q.WCNT-4
           000010   Q$COMP=Q.COMP-4
                    .IF EQ MMG$T
                    Q.ELGH=^O16
                    .IFF
           000016   Q.PAR=^O16
           000012   Q$PAR=^O12
           000024   Q.ELGH=^O24
                    .ENDC

           177552   PCB     == PC$CSR+2                         ;DATA REGISTER

                    ; PAPER TAPE PUNCH CONTROL REGISTERS
                    .IIF NDF PP$VEC, PP$VEC == PC$VEC+4        ;PUNCH VECTOR
                                                                ;ADDR
                    .IIF NDF PP$CSR, PP$CSR == PC$CSR+4        ;PUNCH CONTROL
                                                                ;REGISTER
           177556   PPB     = PP$CSR+2                ;PUNCH DATA BUFFER

           000001   PRGO    = 1                       ;READER ENABLE BIT
           000101   PINT    = 101                     ;INTERRUPT ENABLE BIT
                                                      ;AND GO BIT

```

Header Section

```
                    .SBTTL  DRIVER ENTRY

```

The .DRBEG macro:

```
 000000                  .DRBEG   PC                  ;DEFINE ENTRY POINT AND
 000000             .ASECT
            000052  . = 52
                    .GLOBL  PCEND,PCINT
 000052     000324           .WORD   <PCEND-PCSTRT>
                    .IF B   <>
```

```
000054  000000          .WORD   PCDSIZE
                .IFF
                        .WORD
                .ENDC
                .IF B   <>
000056  000007          .WORD   PCSTS
                .IFF
                        .WORD
                .ENDC
000060  000007          .WORD   ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>
        000176  . = 176
000176  177550  .IIF DF PC$CSR, .WORD   PC$CSR
000000          .PSECT  PCDVR
000000          PCSTRT::
                .IF NB
                .GLOBL
                        .WORD   <-.>/2. -1 + ^O100000
                .IFF
                .IF NB  <>
                .IIF NE &3      .ERROR  ;ODD OR ILLEGAL VECTOR
                        .WORD   &^C3
                .IFF
                .IF DF  PC$VTB
                .GLOBL  PC$VTB
000000  100025          .WORD   <PC$VTB-.>/2. -1 + ^O100000
                .IFF
                .IIF NE PC$VEC&3 .ERROR PC$VEC ;ODD OR ILLEGAL VECTOR
                        .WORD   PC$VEC&^C3
                .ENDC
                .ENDC
                .ENDC
000002  000146          .WORD   PCINT-.,^O340
000004  000340
000006          PCSYS::
000006  000000  PCLQE:: .WORD   0
000010  000000  PCCQE:: .WORD   0
                                                ;QUEUE HEADS
```

I/O Initiation Section

```
000012  016704          MOV     PCCQE,R4        ;POINT TO CURRENT QUEUE
        177772
                                                ;ELEMENT
```

For a character-oriented device, the word count must be shifted left to change it to a byte count (this is the same as multiplying it by 2).

```
000016  006364          ASL     Q$WCNT(R4)      ;CONVERT WORD COUNT TO
        000006
                                                ;BYTE COUNT
000022  103410          BCS     PP              ;IF NEGATIVE, PUNCH (OR
                                                ;ERROR IN NO PUNCH)
000024  001505          BEQ     PCDONE          ;A REQUEST FOR 0 BYTES
                                                ;IS A SEEK, JUST EXIT
000026  012705          MOV     #PC$CSR,R5      ;READ REQUEST, GET THE
        177550
                                                ;CSR
000032  005725          TST     (R5)+           ;IS READER READY?
000034  100064          BPL     PCGORD          ;YES, START TRANSFER
000036  052754          BIS     #EOF$,@-(R4)    ;NOT READY ON ENTRY,
        020000
                                                ;SET EOF
000042  000504          BR      PCFIN           ;AND COMPLETE OPERATION

                ; START PUNCH OPERATION (IMMEDIATE ERROR IF NO PUNCH
                ; SUPPORT)

000044          PP:
                .IF EQ  PR11$X
000044  052737          BIS     #100,@#PP$CSR   ;CAUSES INTERRUPT,
        000100
        177554
                                                ;STARTING TRANSFER
000052  000207          RTS     PC
```

Table for two vectors:

```
                       ; PUNCH-READER VECTOR TABLE

```

The .DRVTB macro:

```
   000054                           .DRVTB   PC,PC$VEC,PCINT
                       .IF NB   PC
   000054              PC$VTB::
                       .IFF
                       .=.-2
                       .ENDC
   000054  000070                   .WORD    PC$VEC&^C3,PCINT-.,340!0,0
   000056  000072
   000060  000340
   000062  000000
   000064                           .DRVTB   ,PP$VEC,PPINT
                       .IF NB
                       $VTB::
                       .IFF
           000062'     .=.-2
                       .ENDC
   000062  000074                   .WORD    PP$VEC&^C3,PPINT-.,340!0,0
   000064  000010
   000066  000340
   000070  000000

```

Punch Interrupt Service Section

```
                       ; PUNCH INTERRUPT SERVICE

```

The .DRAST macro:

```
   000072                           .DRAST   PP,4,PCDONE
                       .GLOBL   $INPTR
                       .IF B <PCDONE>
                                    RTS      %7
                       .IFF
```

The abort entry point:

```
   000072  000462                   BR       PCDONE
                       .ENDC
   000074  004577      PPINT::      JSR      %5,@$INPTR
           000220
   000100  000140                   .WORD    ^C<4*^040>&^0340
   000102  016704                   MOV      PCCQE,R4;POINT TO CURRENT QUEUE
           177702
                                                                 ;ELEMENT
   000106  012705                   MOV      #PP$CSR,R5          ;POINT TO PUNCH STATUS
           177554
                                                                 ;REGISTER
```

Bit 15 in PP$CSR is the error bit. The possible errors for paper tape include device out of tape, and tape jammed.

```
   000112  005725                   TST      (R5)+               ;ERROR?
   000114  100411                   BMI      PPERR               ;YES, PUNCH OUT OF PAPER
                       .IF EQ       MMG$T
                                    ADD      #Q$WCNT,R4          ;POINT TO WORD COUNT
```

The transfer is done if the required number of bytes is transferred without error.

```
                               TST     @R4              ;ANY MORE CHARACTERS TO
                                                        ;OUTPUT?
                               BEQ     PCDONE           ;NO, TRANSFER DONE
                               INC     @R4              ;DECREMENT BYTE COUNT
                                                        ;(IT IS NEGATIVE)
                               MOVB    @-(R4),@R5       ;PUNCH CHARACTER
                               INC     @R4              ;BUMP POINTER
                       .IFF
   000116  005764              TST     Q$WCNT(R4)       ;ANY MORE CHARACTERS TO
           000006
                                                        ;OUTPUT?
   000122  001446              BEQ     PCDONE           ;NO, TRANSFER DONE
   000124  005264              INC     Q$WCNT(R4)       ;DECREMENT BYTE COUNT
           000006
                                                        ;(IT IS NEGATIVE)
```

$GTBYT is a pointer to the monitor $GETBYT routine.

```
   000130  004777              JSR     PC,@$GTBYT       ;GET A BYTE FROM USER
           000152
                                                        ;BUFFER
   000134  112615              MOVB    (SP)+,@R5        ;PUNCH IT
                       .ENDC ;EQ MMG$T
```

Return to the monitor:

```
   000136  000207              RTS     PC

                       .ENDC ;EQ PR11$X

```

Character-oriented devices should check for disabling conditions, such as no power on device, or no tape in reader or punch, and set the hard error bit (bit 0) in the Channel Status Word.

```
   000140  052754      PPERR:  BIS     #HDERR$,@-(R4)   ;SET HARD ERROR BIT
           000001
   000144  000443              BR      PCFIN            ;GO TO I/O COMPLETION
```

Reader Interrupt Service Section

```
                       ; READER INTERRUPT SERVICE

```

The .DRAST macro:

```
   000146                      .DRAST  PC,4,PCDONE
                       .GLOBL  $INPTR
                       .IF B <PCDONE>
                               RTS     %7
                       .IFF
```

The abort entry point:

```
   000146  000434              BR      PCDONE
                       .ENDC
   000150  004577      PCINT:: JSR     %5,@$INPTR
           000144
   000154  000140              .WORD   ^C<4*^040>&^0340
   000156  016704              MOV     PCCQE,R4         ;POINT TO CURRENT QUEUE
           177626
```

```
                                                        ;ELEMENT
                        .IF EQ  MMG$T
                                ADD     #Q$WCNT,R4      ; AT WORD COUNT
                        .ENDC ;EQ MMG$T
  000162  012705                MOV     #PC$CSR,R5      ;POINT TO READER STATUS
          177550
                                                        ;REGISTER
  000166  005725                TST     (R5)+           ;ANY ERRORS?
  000170  100411                BMI     PREOF           ;YES, ZERO-FILL BUFFER
                                                        ;(GIVE EOF NEXT TIME)
                        .IF EQ  MMG$T
                                MOVB    @R5,@-(R4)      ;PUT CHARACTER INTO
                                                        ;BUFFER
                                INC     (R4)+           ;BUMP BUFFER POINTER
                                DEC     @R4             ;DECREASE BYTE COUNT
                        .IFF
  000172  111546                MOVB    @R5,-(SP)       ;GET A CHARACTER
  000174  004777                JSR     PC,@$PTBYT      ;MOVE IT TO USER'S BUFFER
          000110
  000200  005364                DEC     Q$WCNT(R4)      ;DECREASE BYTE COUNT
          000006
                        .ENDC ;EQ MMG$T
  000204  001415                BEQ     PCDONE          ;IF ZERO, WE ARE DONE
                                                        ;WITH THIS READ REQUEST
  000206  052745        PCGORD: BIS     #PINT,-(R5)     ;ENABLE READER INTERRUPT,
          000101
                                                        ;GET A CHARACTER
```

Return to the monitor:

```
  000212  000207                RTS     PC
```

Stop the device if there are errors or if the end of tape is reached:

```
  000214  005045        PREOF:  CLR     -(R5)           ;DISABLE READER
                                                        ;INTERRUPTS
  000216                        .FORK   PCFBLK          ;REQUEST SYSTEM PROCESS
  000216  004577                JSR     %5,@$FKPTR
          000100
  000222  000050                .WORD   PCFBLK -
```

For character-oriented devices, it is necessary to clear the remainder of the user's buffer when end of file is reached (if CTRL/Z is typed on the console terminal, if there is no tape in the reader, and so on). The handler sets the EOF bit in the Channel Status Word the next time the handler is called to do a transfer. This convention makes character-oriented devices appear the same as random-access devices, and is in keeping with the RT–11 device independence philosophy.

```
  000224                1$:                             ;CLEAR REMAINDER OF USER
                                                        ;BUFFER
                        .IF EQ  MMG$T
                                CLRB    @-(R4)          ;CLEAR A BYTE
                                INC     (R4)+           ;BUMP BUFFER ADDRESS
                                DEC     @R4             ;COUNT DOWN BYTES
                                                        ;REMAINING
                        .IFF
  000224  005046                CLR     -(SP)           ;SET NULL BYTE
  000226  004777                JSR     PC,@$PTBYT      ;PUT BYTE INTO USER
          000056
                                                        ;BUFFER
  000232  005364                DEC     Q$WCNT(R4)      ;COUNT DOWN BYTES
          000006
                                                        ;REMAINING
                        .ENDC ;EQ MMG$T
  000236  001372                BNE     1$              ;LOOP UNTIL DONE
```

Branch here on abort also:

```
000240  005037  PCDONE: CLR     @#PC$CSR        ;TURN OFF THE READER
        177550
                                                ;INTERRUPT
                .IF EQ  PR11$X
000244  005037          CLR     @#PP$CSR        ;TURN OFF THE PUNCH
        177554
                                                ;INTERRUPT
                .ENDC ;EQ PR11$X
000250  005067          CLR     PCFBLK+2        ;CLEAR FORK BLOCK TO
        000020
                                                ;AVOID DISPATCH
```

I/O Completion Section

The .DRFIN macro:

```
000254          PCFIN:  .DRFIN  PC              ;GO TO I/O COMPLETION
                .GLOBL  PCCQE
000254  010704          MOV     %7,%4
000256  062704          ADD     #PCCQE-.,%4
        177532
000262  013705          MOV     @#^054,%5
        000054
000266  000175          JMP     @^0270(5)
        000270

000272  000000  PCFBLK: .WORD   0,0,0,0         ;FORK QUEUE BLOCK
000274  000000
000276  000000
000300  000000

```

Handler Termination Section

The .DREND macro:

```
000302                  .DREND  PC
000302          .PSECT  PCDVR
                .IIF NDF PC$END, PC$END:
                .IF EQ  .-PC$END
000302          PC$END::
                .IF NE MMG$T
000302  000000  $RLPTR:: .WORD  0
000304  000000  $MPPTR:: .WORD  0
000306  000000  $GTBYT:: .WORD  0
000310  000000  $PTBYT:: .WORD  0
000312  000000  $PTWRD:: .WORD  0
                .ENDC
                .IF NE ERL$G
000314  000000  $ELPTR:: .WORD  0
                .ENDC
                .IF NE TIM$IT
000316  000000  $TIMIT:: .WORD  0
                .ENDC
000320  000000  $INPTR:: .WORD  0
000322  000000  $FKPTR:: .WORD  0
                .GLOBL  PCSTRT
        000324  PCEND == .
                .IIF
                .PSECT  PCBOOT

                .IIF LT <PCBOOT-.+664>, .ERROR  ;PRIMARY BOOT TOO LARGE

                .       = PCBOOT+664
                BIOERR: JSR     R1,REPORT
                        .WORD   IOERR-PCBOOT
```

```
REPORT: MOV     #BOOTF-PCBOOT,R0
        JSR     R1,REP
        MOV     (R1)+,R0
        JSR     R1,REP
        MOV     #CRLFLF-PCBOOT,R0
        JSR     R1,REP
        RESET
        HALT
        BR      .-2

REPOR:  MOVB    (R0)+,@#TPB
REP:    TSTB    @#TPS
        BPL     REP
        TSTB    @R0
        BNE     REPOR
        RTS     R1

BOOTF:  .ASCIZ  <CR><LF>"?BOOT-U-"<200>
CRLFLF: .ASCIZ  <CR><LF><LF>
IOERR:  .ASCIZ  "I/O error"
        .EVEN
PCBEND::
.ENDC

        000001  .END
```

```
Symbol table

ABTIO$= 001000          PC$VTB  000054RG        002 Q.CSW = 000002
EOF$  = 020000          PINT  = 000101              Q.ELGH= 000024
ERL$G = 000001          PP      000044R         002 Q.FUNC= 000006
FILST$= 100000          PPB   = 177556              Q.JNUM= 000007
HDERR$= 000001          PPERR   000140R         002 Q.LINK= 000000
HNDLR$= 004000          PPINT   000074RG        002 Q.PAR = 000016
MMG$T = 000001          PP$CSR= 177554 G            Q.UNIT= 000007
PCB   = 177552 G        PP$VEC= 000074 G            Q.WCNT= 000012
PCCQE   000010RG   002  PREOF   000214R         002 RONLY$= 040000
PCDONE  000240R    002  PRGO  = 000001              RTE$M = 000000
PCDSIZ= 000000          PR11$X= 000000              SPECL$= 010000
PCEND = 000324RG   002  Q$BLKN= 000000              SPFUN$= 002000
PCFBLK  000272R    002  Q$BUFF= 000004              TIM$IT= 000001
PCFIN   000254R    002  Q$COMP= 000010              VARSZ$= 000400
PCGORD  000206R    002  Q$CSW = 177776              WONLY$= 020000
PCINT   000150RG   002  Q$FUNC= 000002              $ELPTR  000314RG   002
PCLQE   000006RG   002  Q$JNUM= 000003              $FKPTR  000322RG   002
PCSTRT  000000RG   002  Q$LINK= 177774              $GTBYT  000306RG   002
PCSTS = 000007          Q$PAR = 000012              $INPTR  000320RG   002
PCSYS   000006RG   002  Q$UNIT= 000003              $MPPTR  000304RG   002
PC$COD= 000007          Q$WCNT= 000006              $PTBYT  000310RG   002
PC$CSR= 177550 G        Q.BLKN= 000004              $PTWRD  000312RG   002
PC$END  000302RG   002  Q.BUFF= 000010              $RLPTR  000302RG   002
PC$VEC= 000070 G        Q.COMP= 000014              $TIMIT  000316RG   002

. ABS.  000200     000  (RW,I,GBL,ABS,OVR)
        000000     001  (RW,I,LCL,REL,CON)
PCDVR   000324     002  (RW,I,LCL,REL,CON)
Errors detected:   0


*** Assembler statistics

Work file reads    : 0
Work file writes   : 0
Size of work file  : 10068 words ( 40 pages)
Size of core pool  : 15104 words ( 59 pages)

Elapsed time: 00:00:07.00
,PC/L:ME:TTM=CND,PC
```

Appendix B
Converting Device Handlers to V05 Format

Handlers for data devices need no conversion to upgrade from V04 format to V05 format. If your system conditional file, SYCND.MAC, is the same for V04 and V05, you can use the .SYS handler files from V04 on the V05 system without reassembling or relinking them. The conditional files are the same if you use the distributed monitors, or if you perform a system generation and select the same support for memory management (MMG$T), error logging (ERL$G), and device time-out (TIM$IT).

The device handler macros (.DRBEG, .DRAST, and so on) are completely upward-compatible. This means that you can reassemble a V04 handler on a running V05 system without altering the source file.

There is one case requiring change. If you use a V05 XM monitor with .FETCH support and have a handler which does its own mapping to the user buffer by borrowing PAR1, the code in the handler which does the mapping must be changed if you want the handler to be fetchable. Refer to the listing of the DX handler in Appendix A for an example of the new V05 format to use for buffer mapping by a handler.

Even a V04 handler which maps to user buffers can be used with an XM monitor that has .FETCH support if the handler is always loaded first. There is some danger in this, however, because if the XM monitor has .FETCH support, the LOAD command does not check to see if the handler is loading into PAR1. In addition, if you forget to issue the LOAD command and the system attempts to do a .FETCH on the handler, the system will almost certainly crash. If you use this technique, you must make sure that any handler that does old-style mapping to a user buffer does not load into PAR1 space.

In summary:

Monitor	**Changes Required In V04 Handlers**
SJ, FB	None
XM without .FETCH support	None
XM with .FETCH support	None if handler does not do its own user buffer mapping. (Routines $MPPHY, $GETBYT, $PUTBYT, $PUTWRD are okay.) If handler does its own user buffer mapping through PAR1, change code that does PAR1 mapping to use $P1EXT routine instead. Or, always LOAD the handler, but be sure it does not load into PAR1 space.

Appendix C
Sample Application Program

This appendix contains a listing of ADDISK.MAC, a foreground program that collects data from an A/D converter and stores it in a buffer. The program, which also writes the buffer contents to mass storage, contains an example of an in-line interrupt service routine. ADDISK requires LPS hardware.

Figure C–1: Sample Application Program

```
ADDISK.MAC  TEST ANALOG SAMPLER MACRO V04.00  2-JAN-80 02:30:57
TABLE OF CONTENTS

1-   8      EQUATES AND MCALLS
1-  57      USR DATA
3-   1      PROGRAM INITIALIZATION
3-  27      SCAN INITIALIZATION
3-  48      START SAMPLING
5-   1      SCAN COMPLETION
7-   1      INTERRUPT SERVICE ROUTINE
7-  54      INTERRUPT COMPLETION ROUTINE
9-  31      PROGRAMMED REQUEST AREAS
9-  56      TEMPORARY STORAGE AND BUFFERS

                            .NLIST TTM
                            .NLIST CND
                            .TITLE ADDISK.MAC  TEST ANALOG SAMPLER (LPS)

                    ;       AN ANALOG SAMPLING PROGRAM - W.N-S.
                    ;          EDITED & MODIFIED  8/79 - L.C.P.

                            .SBTTL EQUATES AND MCALLS

                    ;       EQUATES SPECIFIC FOR THE RT-11 MONITOR

        000044              JSW=44                  ;ABSOLUTE LOCATION OF THE
                                                    ;JOB STATUS WORD
        000046              USRPTR=46               ;USR POINTER LOCATION

                    ;       EQUATES SPECIFIC FOR LPS HARDWARE

        000360              ADVEC=360               ;VECTOR FOR A/D INTERRUPT
        000006              ADCPRI=6                ;A/D HARDWARE PRIORITY
        000140              ADVAL=140               ;A/D START ON CLOCK
        000417              CLKVAL=417              ;60 LPS CLOCK TICKS
                                                    ;PER SECOND,
                                                    ;REPEAT MODE & GO BIT SET
        177772              BPVAL=-6.               ;# OF TICKS PER SAMPLE
        170400              ADSTAT=170400           ;A/D STATUS REGISTER
        170402              ADDATA=170402           ;A/D DATA REGISTER
        170402              LEDREG=ADDATA           ;LED DISPLAY REGISTER
        170404              CLKREG=170404           ;CLOCK STATUS REGISTER
        170406              BPREG=170406            ;CLOCK BUFFER PRESET
                                                    ;REGISTER

                    ;       EQUATES SPECIFIC TO THE BUFFERING
                    ;       *EDIT FOR CUSTOM CONFIGURATION*

        000004              TOTBUF=4                ;TOTAL BUFFERS PER 'RUN'
        000010              BLKBUF=10               ;MASS STORAGE BLOCKS PER
                                                    ;BUFFER
```

Figure C-1: Sample Application Program (Cont.)

```
        004000                  BUFSIZ=BLKBUF*400       ;WORDS (SAMPLES) PER
                                                        ;BUFFER

                        ;       EQUATES SPECIFIC TO ERROR FLAGGING

        000001                  PRERR=1                 ;PROGRAMMED REQUEST ERROR
        000002                  SYERR=2                 ;SYNCH ERROR, BUFFER
                                                        ;FILLED TOO SOON
        000004                  ADERR=4                 ;A/D ERROR, SLOW
                                                        ;INTERRUPT SERVICE
        000010                  WRERR=10                ;WRITE ERROR, SLOW
                                                        ;BUFFER OUTPUT

                        ;       MCALLS FOR PROGRAMMED REQUESTS AND MACROS

                                .MCALL  .PRINT,.EXIT,.TTYIN,.GTJB,.TTINR,.SPND
                                .MCALL  .RSUM,.ENTER,.WRITE,.PURGE,.WAIT,.CLOSE
                                .MCALL  .SYNCH,.DEVICE,.PROTECT,.INTEN,.QSET

                                .SBTTL  USR DATA

                        ;       THE USR DATA IS GROUPED HERE TO MAXIMIZE THE
                        ;       SPACE FOR SWAPPING BY THE USR AND MINIMIZE THE
                        ;       TOTAL FOREGROUND SPACE.

000000  075306          DEVBLK: .RAD50  /SYOADDISKDAT/  ;USED BY .ENTER CALL
000002  003344
000004  035503
000006  014474

000010                  EAREA:  .BLKW   5               ;EMT ARG BLOCK FOR .ENTER

                                .SBTTL PROGRAM INITIALIZATION

000022                  USRSWP:                         ;USR SWAP ADDRESS HERE...

000022                  LPSST:: .PRINT  #BGNMSG ;START PROGRAM HERE
000022  012700                  MOV     #BGNMSG,%0
        001205'
000026  104351                  EMT     ^O351
000030  012737                  MOV     #USRSWP,@#USRPTR ;FILL IN USR SWAP
        000022'
        000046
                                                        ;ADDRESS FOR MONITOR
000036                          .GTJB   #AREA,#JBBLK    ;GET THE JOB INFORMATION
000036  012700                  MOV     #AREA,%0
        001150'
000042  012710                  MOV     #16.*^O400+1,(0)
        010001
000046  012760                  MOV     #JBBLK,2.(0)
        001102'
        000002
000054  012760                  MOV     #-3,4.(0)
        177775
        000004
000062  104375                  EMT     ^O375
000064  016767                  MOV     GJOBNO,SJOBNO   ;GIVE JOB # TO .SYNCH
        001012
        001042

                        ;       PROTECT INTERRUPT VECTORS

000072                          .PROTECT #AREA,#ADVEC   ;PROTECT A/D INTERRUPT
000072  012700                  MOV     #AREA,%0
        001150'
000076  012710                  MOV     #25.*^O400+0,(0)
        014400
000102  012760                  MOV     #ADVEC,2.(0)
        000360
        000002
000110  104375                  EMT     ^O375
                                                        ;VECTOR
000112  103554                  BCS     PROERR          ;BRANCH IF ERROR
```

Figure C–1: Sample Application Program (Cont.)

```
000114  005037          CLR     @#ADSTAT         ;MAKE SURE A/D IS OFF!
        170400
000120  012737          MOV     #ADINT,@#ADVEC   ;SET UP INTERRUPT SERVICE
        000454'
        000360
                                                 ;ADDRESS IN VECTOR
000126  012737          MOV     #340,@#ADVEC+2   ;AT PRIORITY 7
        000340
        000362
000134  005037          CLR     @#CLKREG         ;MAKE SURE CLOCK IS OFF
        170404

                ;       THIS PROGRAMMED REQUEST GUARANTEES THAT THE A/D
                ;       WILL BE TURNED OFF WHEN WE EXIT THIS JOB...

000140                  .DEVICE #AREA,#ADDEV
000140  012700          MOV     #AREA,%0
        001150'
000144  012710          MOV     #12.*^0400+0,(0)
        006000
000150  012760          MOV     #ADDEV,2.(0)
        001074'
        000002
000156  104375          EMT     ^0375

                        .SBTTL SCAN INITIALIZATION

000160          SCAN:                            ; START THE SCANNING HERE
000160                  .ENTER  #EAREA,#0,#DEVBLK,#BLKBUF*TOTBUF ;OPEN
000160  012700          MOV     #EAREA,%0
        000010'
000164  012710          MOV     #0+<2.*^0400>,(0)
        001000
000170  012760          MOV     #DEVBLK,2.(0)
        000000'
        000002
000176  012760          MOV     #BLKBUF*TOTBUF,4.(0)
        000040
        000004
000204  104375          EMT     ^0375
                                                 ;DISK FILE
000206  103512          BCS     ENTERR           ;BRANCH IF ERROR
000210  005067          CLR     BLOCK            ;START AT BLOCK #0
        000746

                ;       INITIALIZE THE BUFFER POINTERS AND COUNTERS

000214  012767          MOV     #BUFSIZ,BUFCTR   ;SET UP SAMPLE COUNTER
        004000
        000742
000222  012767          MOV     #BUFA,LSTPTR     ;SET UP CURRENT BUFFER
        001776'
        000744
                                                 ;POINTER
000230  012767          MOV     #BUFB,NXTPTR     ;SET UP NEXT BUFFER
        011776'
        000734
                                                 ;POINTER (DOUBLE
                                                 ;BUFFERING)
000236  016767          MOV     LSTPTR,BUFPTR    ;A/D WILL START WITH
        000732
        000724
                                                 ;THIS BUFFER
000244  012767          MOV     #TOTBUF,BUFNO    ;INITIALIZE # BUFFERS
        000004
        000714
                                                 ;IN RUN

                        .SBTTL START SAMPLING

                ;       START THE SAMPLING

```

Figure C-1: Sample Application Program (Cont.)

```
   000252  012737            MOV     #ADVAL,@#ADSTAT ;SET UP FOR A/D SAMPLING
           000140
           170400
                                                      ;ON CLOCK OVERFLOW
   000260  012737            MOV     #BPVAL,@#BPREG ;SET UP CLOCK PRESET
           177772
           170406
                                                      ;BUFFER
   000266  012737            MOV     #CLKVAL,@#CLKREG ;START UP THE CLOCK !!!
           000417
           170404

                             .SBTTL SCAN COMPLETION

                     ;       WAIT HERE UNTIL FULL SCAN COMPLETES

   000274                    .PRINT  #SMESG          ;TELL USER WE'RE
   000274  012700            MOV     #SMESG,%0
           001256'
   000300  104351            EMT     ^0351
                                                      ;SUSPENDING...
   000302                    .SPND                    ;SUSPEND UNTIL RESUME
   000302  012700            MOV     #1*^0400,%0
           000400
   000306  104374            EMT     ^0374
                                                      ;ISSUED FROM
                                                      ;INTERRUPT SERVICE
                                                      ;ROUTINE
                                                      ;WHEN WE ARE AWAKENED,
                                                      ;STOP EVERYTHING !
   000310  005037            CLR     @#CLKREG        ;TURN OFF CLOCK...
           170404
   000314  005037            CLR     @#ADSTAT        ;TURN OFF A/D
           170400
   000320                    .CLOSE  #0              ;CLOSE THE CHANNEL TO
   000320  012700            MOV     #0+<6.*^0400>,%0
           003000
   000324  104374            EMT     ^0374
                                                      ;SAVE THE DATA
   000326  105767            TSTB    ERROR           ;ANY KIND OF ERROR?
           000645
   000332  001434            BEQ     EXITP           ;NO...EXIT NORMALLY

                     ;       PROCESS ANY ERRORS

   000334  132767    ERRPRO: BITB    #PRERR,ERROR    ;HARD WRITE ERROR?
           000001
           000635
   000342  001403            BEQ     ERRWR           ;NO, TRY ANOTHER
   000344                    .PRINT  #WRIMSG ;REPORT DISK WRITE ERROR
   000344  012700            MOV     #WRIMSG,%0
           001736'
   000350  104351            EMT     ^0351
   000352  132767    ERRWR:  BITB    #WRERR,ERROR    ;BUFFER OVERRUN?
           000010
           000617
   000360  001403            BEQ     ERRAD           ;NO, TRY ANOTHER
   000362                    .PRINT  #ERRMSG ;BUFFER OVERRUN, OUTPUT
   000362  012700            MOV     #ERRMSG,%0
           001521'
   000366  104351            EMT     ^0351
                                                      ;TOO SLOW
   000370  132767    ERRAD:  BITB    #ADERR,ERROR    ;A/D OVERRUN?
           000004
           000601
   000376  001403            BEQ     ERRSY           ;NO, TRY ANOTHER
   000400                    .PRINT  #ADCMSG ;A/D ERROR, SLOW
   000400  012700            MOV     #ADCMSG,%0
           001624'
   000404  104351            EMT     ^0351
                                                      ;INTERRUPT SERVICE
```

Figure C–1: Sample Application Program (Cont.)

```
   000406  132767  ERRSY:  BITB    #SYERR,ERROR     ;.SYNCH ERROR?
           000002
           000563
   000414  001403          BEQ     EXITP            ;NO, GO EXIT
   000416                  .PRINT  #SYNMSG ;.SYNCH ERROR, BUFFER
   000416  012700          MOV     #SYNMSG,%0
           001557'
   000422  104351          EMT     ^O351
                                                    ;FILLED TOO SOON

   000424          EXITP:  .PRINT  #EXTMSG ;REPORT EXITING PROGRAM
   000424  012700          MOV     #EXTMSG,%0
           001414'
   000430  104351          EMT     ^O351
   000432                  .EXIT                    ;EXIT TO RT-11
   000432  104350          EMT     ^O350

                   ;       FATAL ERRORS HERE...

   000434          ENTERR: .PRINT  #ENTMSG ;.ENTER ERROR, FATAL!
   000434  012700          MOV     #ENTMSG,%0
           001677'
   000440  104351          EMT     ^O351
   000442                  .EXIT                    ;EXIT IMMEDIATELY
   000442  104350          EMT     ^O350
   000444          PROERR: .PRINT  #PROMSG ;.PROTECT ERROR, FATAL!
   000444  012700          MOV     #PROMSG,%0
           001460'
   000450  104351          EMT     ^O351
   000452                  .EXIT                    ;EXIT IMMEDIATELY
   000452  104350          EMT     ^O350

                           .SBTTL INTERRUPT SERVICE ROUTINE

                   ;       A/D INTERRUPT ROUTINE

   000454          ADINT:  .INTEN  ADCPRI           ;ALERT RT-11 AND DROP
   000454  004577          JSR     5,@^O54
           000054
   000460  000040          .WORD   ^C<ADCPRI*32.>&224.
                                                    ;PRIORITY TO THAT OF HDWE
   000462  005737          TST     @#ADSTAT         ;INTERRUPT SERVICE TOO
           170400
                                                    ;LATE?
   000466  100003          BPL     NOADER           ;BRANCH IF OK...
   000470  152767          BISB    #ADERR,ERROR     ;SET A/D ERROR BIT,
           000004
           000501
                                                    ;BUT CONTINUE
   000476  013777  NOADER: MOV     @#ADDATA,@BUFPTR ;STORE SAMPLE IN BUFFER
           170402
           000464
   000504  004767          CALL    LEDDIS           ;DISPLAY A/D VALUE IN
           000272
                                                    ;LED DISPLAY
   000510  062767          ADD     #2,BUFPTR        ;INCREMENT BUFFER POINTER
           000002
           000452
   000516  005367          DEC     BUFCTR           ;DECREMENT COUNT...THRU
           000442
                                                    ;WITH BUFFER?
   000522  001422          BEQ     BUFFUL           ;BRANCH IF YES
                                                    ;(BUFFER FULL)
   000524  000207          RTS     PC               ;OTHERWISE, JUST RETURN
                                                    ;FROM INTERRUPT

                   ;       THIS CODE IS REACHED IF .SYNCH FAILS
                   ;       (2ND BUFFER FILLED TOO FAST !?!)

   000526  152767  SYNERR: BISB    #SYERR,ERROR     ;SET .SYNCH ERROR BIT
           000002
           000443
```

Figure C-1: Sample Application Program (Cont.)

```
                                                        ;THEN STOP INTERRUPTS!
   000534  005037           CLR     @#CLKREG            ;STOP THE CLOCK...
           170404
   000540  005037           CLR     @#ADSTAT            ;AND THE A/D...
           170400
   000544  012767           MOV     #1,BUFCTR           ;TRY .SYNCH AGAIN ON
           000001
           000412
                                                        ;NEXT INTERRUPT
   000552  012737           MOV     #ADVAL,@#ADSTAT     ;START UP THE A/D AGAIN...
           000140
           170400
   000560  012737           MOV     #CLKVAL,@#CLKREG    ;AND THE CLOCK...
           000417
           170404
   000566  000207           RTS     PC                  ;RETURN AND HOPE FOR
                                                        ;BETTER!

                   ;        THIS CODE PROCESSES THE BUFFER.  IT FIRST ADJUSTS
                   ;        THE POINTERS AT PRIORITY OF THE LPS HARDWARE,
                   ;        THEN ISSUES A .SYNCH AND WRITES THE BUFFER TO
                   ;        MASS STORAGE. (THIS CODE RUNS AS AN INTERRUPT
                   ;        COMPLETION ROUTINE AT PRIORITY LEVEL 0).

   000570  012767  BUFFUL:  MOV     #BUFSIZ,BUFCTR      ;RESET THE BUFFER COUNTER
           004000
           000366
   000576  156767           BISB    WFLAG,ERROR         ;SET A 'WRITE-IN-PROGRESS'
           000374
           000373
                                                        ;FLAG
                                                        ;(WILL BE DOWN UNLESS
                                                        ;WRITE IS UNFINISHED)
   000604  016767           MOV     NXTPTR,BUFPTR       ;FLIP-FLOP THE BUFFER
           000362
           000356
                                                        ;POINTERS
   000612  016767           MOV     LSTPTR,NXTPTR       ;THIS FOR NEXT TIME...
           000356
           000352
   000620  016767           MOV     BUFPTR,LSTPTR       ;THIS FOR TIME AFTER
           000344
           000346
                                                        ;NEXT.

                            .SBTTL INTERRUPT COMPLETION ROUTINE

                   ;        NOW IT IS SAFE TO ALLOW INTERRUPTS...

   000626                   .SYNCH  #ASYN               ;ALLOW PROG REQ & QUEUE
   000626  012704  MOV      #ASYN,%4
           001132'
   000632  013705           MOV     @#^O54,%5
           000054
   000636  004575           JSR     5,@^O324(5.)
           000324
                                                        ;COMPLETION RTNE
   000642  000731           BR      SYNERR              ;ERROR RETURN ... SYNCH
                                                        ;FAILED - GO PROCESS
   000644  105767           TSTB    ERROR               ;NORMAL RETURN... ANY
           000327
                                                        ;KIND OF ERROR LATELY?
   000650  001044           BNE     ERRET               ;YES, LEAVE IMMEDIATELY!
   000652  152767           BISB    #WRERR,WFLAG        ;SET FLAG WHILE WE'RE
           000010
           000316
                                                        ;WRITING TO DISK!
   000660                   .WAIT   #0                  ;MAKE CERTAIN LAST .WRITE
   000660  005000           CLR     %0
   000662  104374           EMT     ^O374
                                                        ;FINISHED OK
   000664  103442           BCS     WRIERR              ;BRANCH IF IT DIDN'T
```

Figure C-1: Sample Application Program (Cont.)

```
70 000666                          .WRITE  #AREA,#0,NXTPTR,#BUFSIZ,BLOCK ;OUTPUT
   000666  012700                  MOV     #AREA,%0
           001150'
   000672  012710                  MOV     #0+<9.*^O400>,(0)
           004400
   000676  016760                  MOV     BLOCK,2.(0)
           000260
           000002
   000704  016760                  MOV     NXTPTR,4.(0)
           000262
           000004
   000712  012760                  MOV     #BUFSIZ,6.(0)
           004000
           000006
   000720  012760                  MOV     #1,8.(0)
           000001
           000010
   000726  104375                  EMT     ^O375
71                                                         ;BUFFER TO DISK
72 000730  103420                  BCS     WRIERR          ;BRANCH IF ERROR
73 000732                          .PRINT  #BUFWR          ;LET USER KNOW WE'RE
   000732  012700                  MOV     #BUFWR,%0
           001365'
   000736  104351                  EMT     ^O351
74                                                         ;ALIVE & WELL...
75 000740  142767                  BICB    #WRERR,WFLAG    ;OUTPUT DONE, FLAG
           000010
           000230
76                                                         ;BACK DOWN
77 000746  062767                  ADD     #BLKBUF,BLOCK   ;UPDATE BLOCK POINTER...
           000010
           000206
78 000754  005367                  DEC     BUFNO           ;DONE WITH A BUFFER, ARE
           000206
79                                                         ;WE THRU WITH SCAN?
80 000760  001003                  BNE     COMRET          ;NO...JUST RETURN IF MORE
81                                                         ;BUFFERS IN SCAN
82 000762                  ERRET:  .RSUM                   ;ERRORS, OR DONE ...
   000762  012700                  MOV     #2.*^O400,%0
           001000
   000766  104374                  EMT     ^O374
83                                                         ;AWAKEN MAINLINE
84 000770  000207          COMRET: RTS     PC              ;RETURN FROM INTERRUPT
85                                                         ;VIA RMON

 1
 2                         ;       EXIT HERE ON WRITE ERROR
 3
 4 000772  152767          WRIERR: BISB    #PRERR,ERROR    ;FLAG A .WRITE ERROR...
           000001
           000177
 5 001000  000770                  BR      ERRET           ;AND TAKE ERROR RETURN
 6
 7                         ;       DISPLAY DATA IN LPS L.E.D. DISPLAY...
 8
 9 001002  012767          LEDDIS: MOV     #5,LEDCT        ;SET # OF DIGITS TO
           000005
           000174
10                                                         ;DISPLAY
11 001010  005067                  CLR     LEDTMP          ;CLEAR THE WORK SPACE
           000164
12 001014  017767                  MOV     @BUFPTR,ADTEMP  ;GET THE CURRENT
           000150
           000160
13                                                         ;A/D VALUE
14 001022  116767          LEDNX:  MOVB    ADTEMP,LEDTMP   ;GET WHAT'S LEFT OF
           000154
           000150
15                                                         ;A/D VALUE
16 001030  142767                  BICB    #370,LEDTMP     ;STRIP OFF ALL BUT
           000370
           000142
```

Figure C-1: Sample Application Program (Cont.)

```
                                                    ;LOWEST OCTAL DIGIT
001036  016737          MOV     LEDTMP,@#LEDREG ;PUT INTO L.E.D. REGISTER
        000136
        170402
001044  006267          ASR     ADTEMP          ;SHIFT 'REMAINDER'
        000132
                                                ;RIGHT 3 TIMES
001050  006267          ASR     ADTEMP          ;TO RIGHT JUSTIFY
        000126
001054  006267          ASR     ADTEMP          ;THE NEXT DIGIT...
        000122
001060  105267          INCB    LEDTMP+1        ;TELL L.E.D. REGISTER
        000115
                                                ;WHAT NEXT DIGIT IS
001064  105367          DECB    LEDCT           ;DECREMENT DIGIT COUNTER
        000114
                                                ;...ARE WE THRU?
001070  001354          BNE     LEDNX           ;LOOP IF NOT
001072  000207          RETURN                  ;WE'RE DONE
                                                ;...RETURN TO CALLER

                        .SBTTL PROGRAMMED REQUEST AREAS

                ;       THESE AREAS AND DATA BLOCKS ARE USED BY THE
                ;       VARIOUS PROGRAMMED REQUESTS...

001074  170400  ADDEV:  .WORD   ADSTAT          ;.DEVICE LIST - A/D
                                                ;STATUS REGISTER
001076  000000          .WORD   0               ;LOAD WITH 0 (STOP A/D)
001100  000000          .WORD   0               ;STOP THE LIST OF
                                                ;THINGS TO STOP!

001102          JBBLK:                          ;ARG BLOCK FOR .GTJB
001102          GJOBNO: .BLKW   12.             ;FILLED WITH JOB DATA
                                                ;(12 WDS FOR V4)
001132  000000  ASYN:   .WORD   0               ;ARG BLOCK FOR THE .SYNCH
001134  000000  SJOBNO: .WORD   0               ;FILLED AFTER THE
                                                ;.GTJB CALL
001136  000000          .WORD   0
001140  000000          .WORD   0
001142  000000          .WORD   0,-1,0          ;REQUIRED VALUES FOR
001144  177777
001146  000000
                                                ;RT-11 !!!

001150          AREA:   .BLKW   5               ;EMT ARG BLOCK FOR
                                                ;VARIOUS PROG REQUESTS

                        .SBTTL TEMPORARY STORAGE AND BUFFERS

                ;       MISCELLANEOUS STUFF...

001162  000000  BLOCK:  .WORD   0               ;BLOCK POINTER FOR
                                                ;DISK WRITES...
001164  000000  BUFCTR: .WORD   0               ;SAMPLES LEFT TO PUT
                                                ;IN BUFFER...
001166  000000  BUFNO:  .WORD   0               ;# BUFFERS LEFT IN SCAN...
001170  000000  BUFPTR: .WORD   0               ;POINTER WITHIN BUFFER
                                                ;BEING FILLED...
001172  000000  NXTPTR: .WORD   0               ;POINTER TO BUFFER
                                                ;BEING WRITTEN...
001174  000000  LSTPTR: .WORD   0               ;POINTER TO OTHER BUFFER...
001176     000  WFLAG:  .BYTE   0               ;WRITE FLAG ... =1 IF
                                                ;.WRITE IN PROGRESS
001177     000  ERROR:  .BYTE   0               ;ERROR STATUS WORD
                                                ;(BIT MASK)
001200  000000  LEDTMP: .WORD   0               ;L.E.D. DISPLAY WORKING
                                                ;SPACE
001202  000000  ADTEMP: .WORD   0               ; (DITTO)
001204     000  LEDCT:  .BYTE   0               ;# OF L.E.D. DIGITS TO
                                                ;DISPLAY (COUNTER)
```

Figure C-1: Sample Application Program (Cont.)

```
                      ;       MESSAGE TEXT...

                      .NLIST  BIN
   001205    BGNMSG:  .ASCIZ %*** A/D SAMPLING DEMO (LPS HARDWARE) ***%
   001256    SMESG:   .ASCII /SAMPLING STARTS, MAINLINE SUSPENDS/<12><15>
   001322             .ASCIZ /<THIS DEMO TAKES ABOUT 15 MINUTES>/
   001365    BUFWR:   .ASCIZ /BUFFER XFERRED TO DISK/
   001414    EXTMSG:  .ASCIZ %*** A/D SAMPLING DEMO COMPLETED ***%
   001460    PROMSG:  .ASCIZ /? .PROTECT ERROR - DEMO ABORTED!/
   001521    ERRMSG:  .ASCIZ /? BUFFER OVERRUN, SLOW OUTPUT/
   001557    SYNMSG:  .ASCIZ /? .SYNCH ERROR, SLOW SYNCH SERVICING/
   001624    ADCMSG:  .ASCIZ %? A/D TIMING ERROR, SLOW INTERRUPT SERVICE%
   001677    ENTMSG:  .ASCIZ /? .ENTER ERROR - DEMO ABORTED!/
   001736    WRIMSG:  .ASCIZ /? .WRITE ERROR - DEMO ABORTED!/
                      .EVEN
                              .LIST   BIN
   001776             BUFA:   .BLKW   BUFSIZ          ;BUFFER 'A'
   011776             BUFB:   .BLKW   BUFSIZ          ;BUFFER 'B'
             000022'          .END    LPSST           ;END OF SOURCE CODE

SYMBOL TABLE

ADCMSG  001624R         BUFWR   001365R         LEDTMP  001200R
ADCPRI= 000006          CLKREG= 170404          LPSST   000022RG
ADDATA= 170402          CLKVAL= 000417          LSTPTR  001174R
ADDEV   001074R         COMRET  000770R         NOADER  000476R
ADERR = 000004          DEVBLK  000000R         NXTPTR  001172R
ADINT   000454R         EAREA   000010R         PRERR = 000001
ADSTAT= 170400          ENTERR  000434R         PROERR  000444R
ADTEMP  001202R         ENTMSG  001677R         PROMSG  001460R
ADVAL = 000140          ERRAD   000370R         SCAN    000160R
ADVEC = 000360          ERRET   000762R         SJOBNO  001134R
AREA    001150R         ERRMSG  001521R         SMESG   001256R
ASYN    001132R         ERROR   001177R         SYERR = 000002
BGNMSG  001205R         ERRPRO  000334R         SYNERR  000526R
BLKBUF= 000010          ERRSY   000406R         SYNMSG  001557R
BLOCK   001162R         ERRWR   000352R         TOTBUF= 000004
BPREG = 170406          EXITP   000424R         USRPTR= 000046
BPVAL = 177772          EXTMSG  001414R         USRSWP  000022R
BUFA    001776R         GJOBNO  001102R         WFLAG   001176R
BUFB    011776R         JBBLK   001102R         WRERR = 000010
BUFCTR  001164R         JSW   = 000044          WRIERR  000772R
BUFFUL  000570R         LEDCT   001204R         WRIMSG  001736R
BUFNO   001166R         LEDDIS  001002R         ...V1 = 000003
BUFPTR  001170R         LEDNX   001022R         ...V2 = 000027
BUFSIZ= 004000          LEDREG= 170402

. ABS.  000000     000
        021776     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  10496 WORDS  ( 41 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  56 PAGES
,V4:ADDISK/L:MEB/L:TTM=V4:ADDISK
```

INDEX

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Chicago	312–640–5612 8:15 A.M. to 5:00 P.M. CT	Digital Equipment Corporation Accessories & Supplies Center 1050 East Remington Road Schaumburg, IL 60195
San Francisco	408–734–4915 8:15 A.M. to 5:00 P.M. PT	Digital Equipment Corporation Accessories & Supplies Center 632 Caribbean Drive Sunnyvale, CA 94086
Alaska, Hawaii	603–884–6660 8:30 A.M. to 6:00 P.M. ET *or* 408–734–4915 8:15 A.M. to 5:00 P.M. PT	
New Hampshire	603–884–6660 8:30 A.M. to 6:00 P.M. ET	Digital Equipment Corporation Accessories & Supplies Center P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A., Puerto Rico*	1–800–258–1710 8:30 A.M. to 6:00 P.M. ET	

*Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (call 809–754–7575)

From	Call	Write
Canada		
British Columbia	1–800–267–6146 8:00 A.M. to 5:00 P.M. ET	Digital Equipment of Canada Ltd 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Ottawa–Hull	613–234–7726 8:00 A.M. to 5:00 P.M. ET	
Elsewhere	112–800–267–6146 8:00 A.M. to 5:00 P.M. ET	
Elsewhere		Digital Equipment Corporation A&SG Business Manager*

*c/o DIGITAL's local subsidiary or approved distributor

RT–11
Software Support Manual
AA–H379B–TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

Did you find errors in this manual? If so, specify the error and the page number.

____

Please indicate the type of user/reader that you most nearly represent.

- __ Assembly language programmer
- __ Higher-level language programmer
- __ Occasional programmer (experienced)
- __ User with little programming experience
- __ Student programmer
- __ Other (please specify) ____________________

Name ____________________ Date ____________________

Organization ____________________ Telephone ____________________

Street ____________________

City ____________________ State __________ Zip Code or Country __________

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — — — —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG/ML PUBLICATIONS, MLO5–5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754

— — Do Not Tear — Fold Here —

Cut Along Dotted Line