RT-11
Advanced Programmer's Guide

Order No. AA-5280B-TC

November 1978

This manual is a reference document for advanced RT-11 users, including FORTRAN-IV users and MACRO-11 assembly language programmers.

RT-11
Advanced
Programmer's Guide

Order No. AA-5280B-TC

SUPERSESSION/UPDATE INFORMATION: This manual supersedes DEC-11-ORAPA-A-D. This manual includes Update Notice No. 1 (AD-5280B-T1), Update Notice No. 2 (AD-5280B-T2), and Update Notice No. 3 (AD-5280B-T3).

OPERATING SYSTEM AND VERSION: RT-11 V03B

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, October 1977
Revised: March 1978
July 1978
September 1978
November 1978

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CONTENTS (Cont.)

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

Page

FIGURES (Cont.)

TABLES

PREFACE

The Advanced Programmer's Guide is intended as a reference document primarily for advanced RT-11 users (including FORTRAN users) and MACRO-11 assembly language programmers. Although there are no absolute prerequisites for reading and understanding the contents of this manual, it is recommended that the reader be familiar with RT-11 operating procedures, PDP-11 system architecture, PDP-11 machine language, MACRO-11 assembly language and if appropriate, another higher level language such as FORTRAN IV.

The Advanced Programmer's Guide consists of the following four chapters and three appendices:

Chapter 1, I/O Programming Conventions - This chapter presents information on RT-11 supported I/O devices, associated device handlers and the various monitor services offered by the RT-11 operating system.

Chapter 2, Programmed Requests - This chapter describes all of the RT-11 programmed requests and provides information on how to use them to develop user-written programs. Program examples are also included to facilitate the explanations.

Chapter 3, Extended Memory - This chapter deals exclusively with the RT-11 concept of memory extension. The memory extension concepts and all memory extension programmed requests are explained in this chapter. An example program utilizing all memory extension programmed requests is included to assist users in developing their own programs to use this new feature.

Chapter 4, System Subroutine Library - This chapter describes all of the RT-11 FORTRAN-callable subroutines. This chapter also contains examples of the calls and most of the subroutines.

Appendix A, Display File Handler - This appendix describes the graphics support for the RT-11 operating system. Program examples are included to assist users in developing their own display program.

Appendix B, System Macro Library - This appendix is a listing of the RT-11 System Macro Library (SYSMAC), which provides the expansions for all RT-11 macro instructions.

Appendix C, Additional I/O Information - This appendix provides software support information for RT-11 programmers.

CHAPTER 1

I/O PROGRAMMING CONVENTIONS

This chapter introduces the MACRO-11 assembly language programmer to the basic concepts and features of device handlers and interrupt service routine for the RT-11 operating system. This system includes three compatible monitors and a variety of programming development tools and system utilities. The monitors and their designations are as follows:

SJ - Single-Job
FB - Foreground/Background
XM - Extended Memory

The SJ monitor is a single user, single job system restricted to 28K words of memory. The FB monitor is a single user, two job system also restricted to 28K words of memory. PDP-11/03 systems that include the MSV11-DD memory board with a special jumper can access 30K words of memory under SJ and FB. The XM monitor is an extension of the FB monitor that supports up to 124K words of physical memory. Operational XM monitors are not distributed on the RT-11 kit. A SYSGEN must be performed to create these monitors and their device handlers. See the <u>RT-11 System Generation Manual</u> for details.

In addition to the monitors already discussed, the SYSGEN program allows the user to create a custom monitor, containing those features required in a particular application. Such a custom monitor can have more or fewer features and can be larger or smaller than the standard monitor (see the <u>RT-11 System Generation Manual</u> for details).

Single-job operation supports only one program in memory at any time; execution of the program continues until either it is completed or it is physically interrupted by the user at the console.

In a foreground/background environment (under either the FB or XM monitor), two independent programs can reside in memory. The foreground program is given priority and executes until it relinquishes control to the background program; the background program executes until control is again required by the foreground program. This sharing of system resources greatly increases the efficiency of processor usage.

RT-11 is fast, reliable, and easy to use. It incorporates a sophisticated set of programming tools for the applications or end-user programmer. These tools and techniques are discussed in subsequent sections.

1.1 MONITOR SOFTWARE COMPONENTS

The main RT-11 monitor software components are:

Resident Monitor (RMON)

Keyboard Monitor (KMON)

User Service Routine (USR) and Command String Interpreter (CSI)

Device Handlers

1.1.1 Resident Monitor (RMON)

The resident monitor is the permanently memory-resident part of RT-11. The programmed requests for most services of RT-11 are handled by RMON. RMON also contains the console terminal support (TT.SYS is not resident in SJ), error processor, system device handler, EMT processor, and system tables.

1.1.2 Keyboard Monitor (KMON)

The keyboard monitor provides communication between the user at the console and the RT-11 system. Keyboard monitor commands allow the user to assign logical names to devices, run programs, load device handlers, invoke indirect command files, and control foreground/background operations. A dot at the left margin of the console terminal page indicates that the keyboard monitor is in memory and is waiting for a user command. KMON is 7400 octal (or 3840 decimal) words long in RT-11 V03B distributed BL, SJ, and FB monitors.

1.1.3 User Service Routine (USR)

The user service routine provides support for the RT-11 file structure and handles some of the programmed requests for RT-11. It loads device handlers, opens files for read or write operations, deletes and renames files, and creates new files. The Command String Interpreter is part of the USR and can be accessed by any program to process a command string. In XM, the USR is permanently resident.

1.1.4 Device Handlers

Device handlers for the RT-11 system perform the actual transfer of data to and from peripheral devices. New handlers can be added to the system as files on the system device and can be interfaced to the system easily by using the keyboard monitor INSTALL command (see Chapter 4 of RT-11 System User's Guide).

1.2 GENERAL MEMORY LAYOUT

The diagrams in Figure 1-1 show how components of the RT-11 system are arranged in memory.

Diagram A illustrates a single-job system just after it was bootstrapped. Location 54 in the system communication area contains the value x, which represents the bottom address of RMONSJ.

Diagram B shows the same single-job system with a background job executing. KMON is not resident in memory while the job is running. If the user job needs the memory space, it can swap over the USR.

Diagram C shows a foreground/background system. Two handlers were made resident by the LOAD command. They reside below RMONFB and above the USR. There is a background job running, so KMON is not shown in memory. If the background job needs the memory space, it can swap over the USR.

Diagram D illustrates the same foreground/background system. There is a foreground job running. There is no background job, so KMON is in memory.

Diagram E shows the same foreground/background system. Both the foreground and the background jobs are in memory. The background job can swap the USR at its default location just below the foreground job. The foreground job must allocate space within its own program area in order to swap in the USR.

Diagram F shows an extended memory system. There are two loaded device handlers, and both a foreground and a background job are in memory. Note that the USR is always resident.

Diagram G illustrates some characteristics of RT-11's memory allocation scheme. The third device handler in the diagram was loaded after the foreground job was started. If the foreground job were stopped and unloaded, the space it occupied would be placed in the free memory list. If the user needed to load another handler, it would reside in that free space if it could fit. If it did not fit, it would reside below the third handler. The USR and KMON slide down in memory to accommodate such new additions.

The memory area directly above the USR contains indirect file information. This section is always located just above the USR, and moves up or down in memory along with the USR. If, for example, the third handler were unloaded, the USR and the KMON would slide up in memory, and reside just below the foreground job.

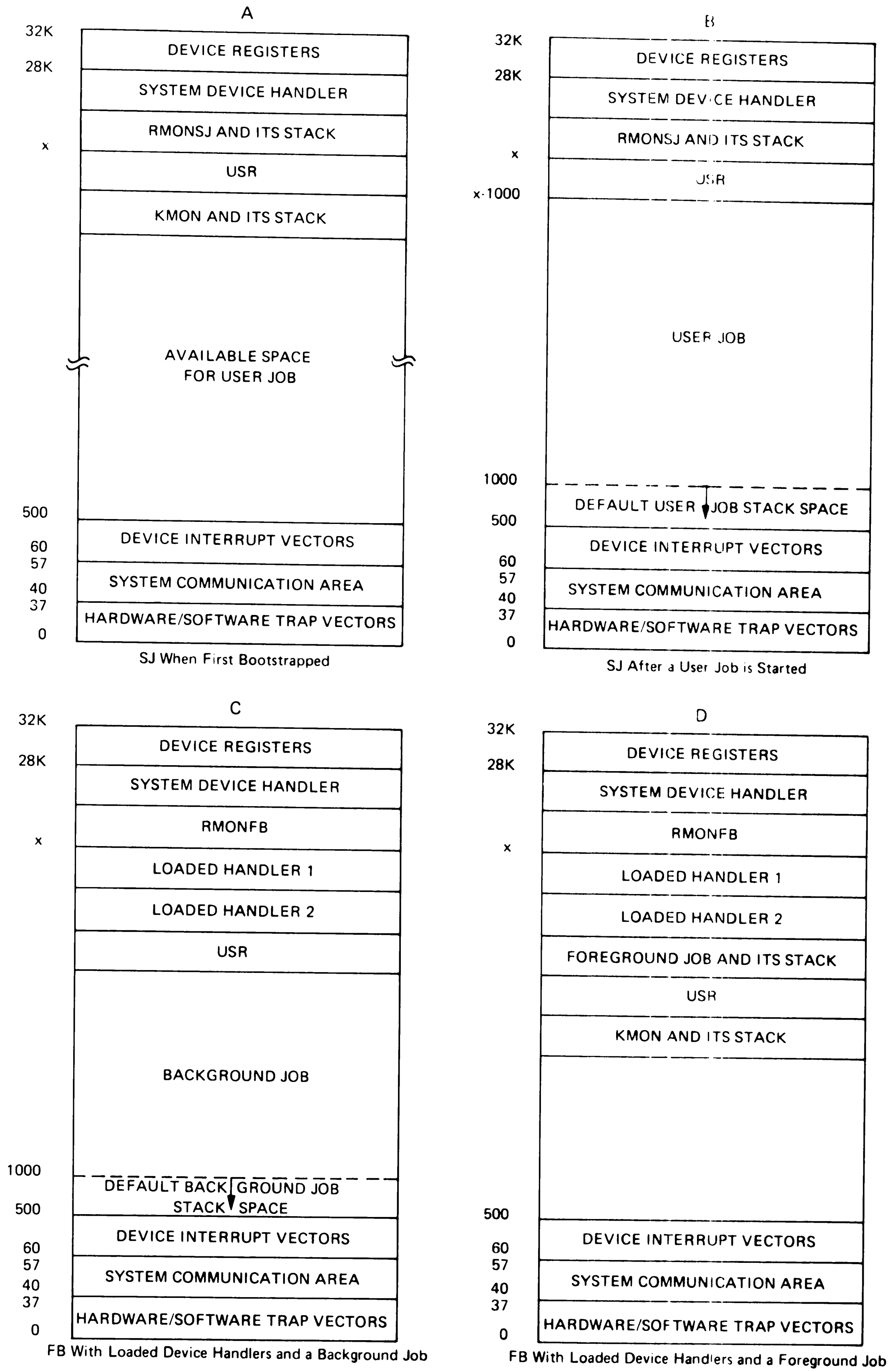


Figure 1-1 RT-11 Memory Layout

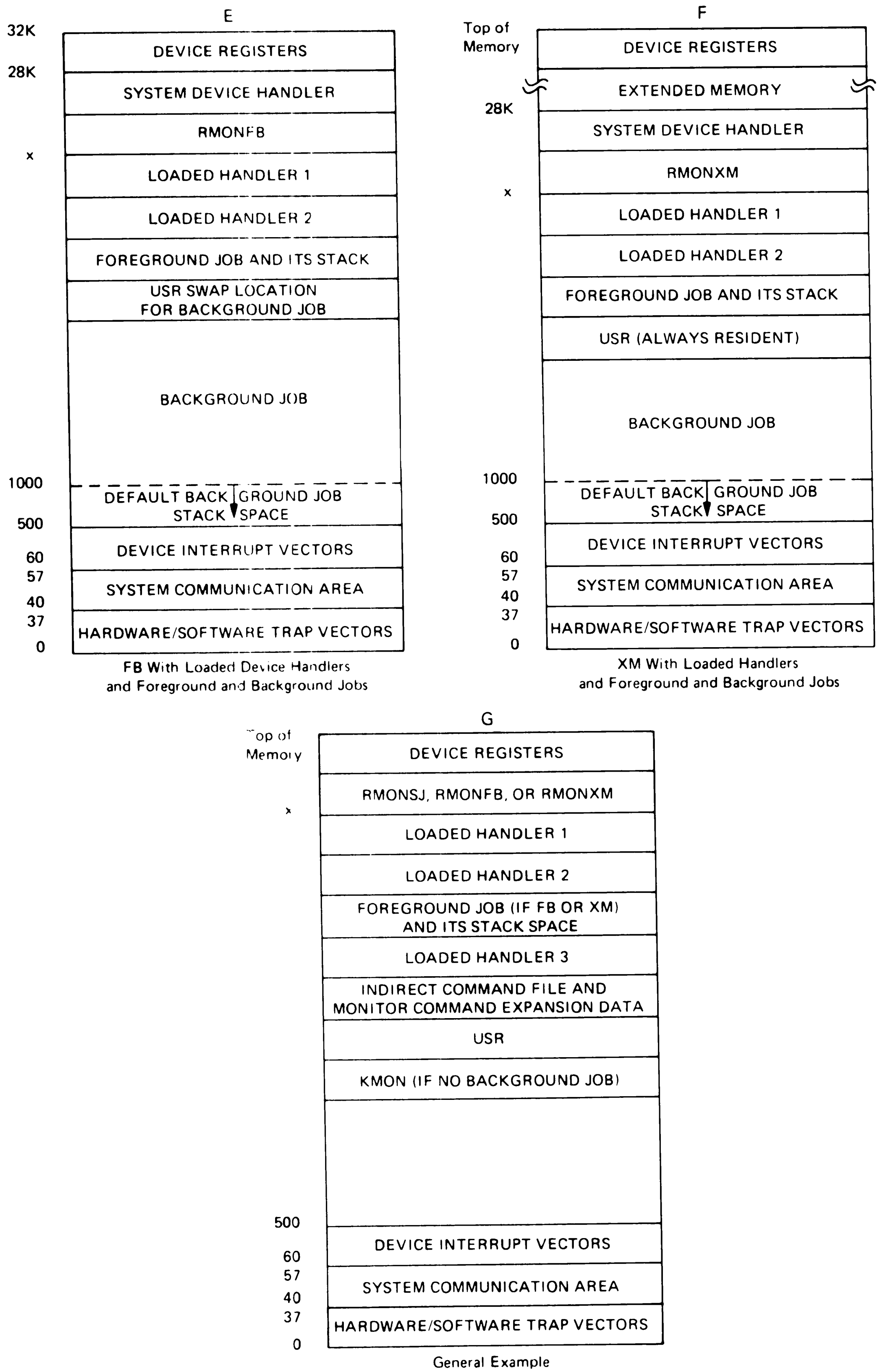


Figure 1-1 (Cont.) RT-11 Memory Layout

In addition to FRUN, which loads foreground jobs, other monitor commands can alter the memory map; these are R, RUN, GET, LOAD, UNLOAD, GT ON, GT OFF, and indirect command files invoked by "@". The LOAD command causes device handlers to be resident until an UNLOAD command is performed. The UNLOAD command removes handlers that have been loaded. The GT ON and GT OFF commands cause terminal service to utilize the VT11 or VS60 display hardware. RT-11 maintains a free memory list to manage memory. Memory space is always reclaimed if possible by moving KMON/USR up. If it cannot be reclaimed, it is placed in the free memory list.

1.3 WRITING USER INTERRUPT SERVICE ROUTINES

Certain programming conventions must be observed in RT-11 when writing user interrupt service routines. All device handlers follow these conventions. The procedures described in this section are necessary and must be followed to prevent system failures when jobs are running under RT-11.

1.3.1 Setting Up Interrupt Vectors

Devices for which no RT-11 handler exists must be serviced by the user program. For example, no LPS11 device handler exists; to use an LPS11, the user must incorporate the interrupt service routine within the program or write the device handler himself. It is the responsibility of the program to set up the vector for devices such as this. The recommended procedure is not to simply move the service routine address and 340 into the desired vector; rather, it is to precede the operation with a .PROTECT macro call. The .PROTECT ensures that neither the other job nor the monitor already has control of that device (FB and XM only). If the .PROTECT is successful, the vector can be initialized.

1.3.2 Interrupt Priorities

The status word for each interrupt vector should be set such that when an interrupt occurs, the processor takes it at level 7. Thus, a device that has its vectors at 70 and 72 has location 70 set to its service routine; location 72 contains 340. The 340 causes the service routine to be entered with the processor set to inhibit any further device interrupts.

1.3.3 Interrupt Service Routine

If conventions are followed, the processor priority will be 7 when an interrupt occurs. The first task of the interrupt service routine is to declare that an interrupt has occurred and to lower the processor priority to the correct value. This can be done by using the .INTEN macro call. The call is:

```
     .INTEN   priority
or
     .INTEN   priority,pic
```

The .INTEN call is explained in Chapter 2, Programmed Requests. On return from the .INTEN call, the processor priority is set properly;

registers 4 and 5 have been saved and can be used without the necessity of saving them again. All other registers must be saved and restored by the program if they are used.

For example, a user device interrupts at processor priority 5:

```
                DEVPRI=5

        DEVINT: .INTEN DEVPRI        ;NOTE, NOT #DEVPRI
                 .
                 .
                 .
                RTS PC
```

If the contents of the processor status word, loaded from the interrupt vector, are significant to the interrupt service routine (such as the condition bits), the PS should be moved to a memory location (not the stack) before issuing the .INTEN. The interrupt service routine uses the monitor stack and should avoid excessive use of stack space.

1.3.4 Return From Interrupt Service

When an interrupt is serviced, instead of issuing an RTI to return from the interrupt, the routine must exit with an RTS PC. This RTS PC returns control to the monitor (assuming that .INTEN has been executed), which then restores registers 4 and 5, and executes the RTI.

1.3.5 Issuing Programmed Requests at the Interrupt Level

Programmed requests from interrupt routines must be preceded by a .SYNCH call. This ensures that the proper job is running when the programmed request is issued. The .SYNCH call assumes that nothing is pushed onto the stack by the user program between the .INTEN call and the .SYNCH call. On successful completion of a .SYNCH, R0 and R1 have been saved and are free to be used. R4 and R5 are no longer free, and should be saved and restored if they are to be used. Programmed requests that require USR action must not be called from within interrupt routines.

1.3.6 User Interrupt Service Routines with the XM Monitor

There are three restrictions to using user interrupt service routines with the XM monitor. See Section 3.6.1 of this manual for specific details.

1.4 DEVICE HANDLERS

This section deals with the device handlers that are part of the RT-11 operating system. Any device dependent information or general information required by the user is contained here. No mention of a handler implies that no special conditions must be met to use that device (all disks, except diskette, RL01, and RK06/07 are in this category, and therefore are not covered here).

1.4.1 Differences Between V2 and V3 Device Handlers

The RT-11 device handler format changed slightly from version 2C to version V03. (There are no changes from version 3 to version 3B.) Most of these changes were brought about by the addition of a system generation process and many new handler options in V03. Changes are implemented through a new set of handler macros, which make conversion easier.

The new handler options being offered in version 3 and later releases include: error logging, I/O time-out, extended memory support, multi-vectored device support and fork level processing. All but fork processing are options that are determined at SYSGEN time. The monitor and the set of handlers must have matching options, so a common option definition file must be used to assemble all the components (drivers and monitors) of the system.

In addition, RT-11 version 2C and version 3 non-NPR device handlers follow different conventions for signalling the end of file condition. In version 2C, a non-NPR device handler sets the EOF bit in the channel status word as soon as it detects an end of file condition on the device (for example, no more paper in the paper tape reader). It can set the EOF bit even if the program's buffer is only partly full. Thus, the program may find the EOF bit on after a transfer that returns some usable data. Programs written for version 2C check the EOF bit after using the last data read.

In contrast, a version 3 non-NPR device handler does not set the EOF bit in the channel status word if the handler returns any usable data to the program. When such a handler detects an end of file condition on the device, it checks to see whether any data has been loaded in the program's buffer. If the buffer is not empty, the handler remembers the end of file condition but does not set the EOF bit. Instead, it fills the rest of the program's buffer with zeros and returns. The next time the handler is entered, it finds the remembered end of file condition, sets the EOF bit, and returns an empty buffer. Programs written for version 3 check the EOF bit as soon as the read is complete; they assume that the buffer is empty if the bit is on.

NOTE

Device handlers distributed with RT-11, Version 1, will not work properly with Version 2. Version 2 device handlers require changes to utilize all features of the version 3 release. Any user-written device handlers should be rewritten to comply with the Version 3 conditions. Instructions for interfacing new handlers to RT-11 are provided in the following portions of Section 1.4 of this manual.

1.4.2 The Parts of a Handler

Every RT-11 format handler has the following seven parts: the preamble, SET options, header, I/O initiation code, asynchronous trap processing code, I/O completion code, and terminator. The following sections describe the format of each of these parts. An example program of a device handler is included at the end of this section. In the following text, "dd" represents the two-character physical device name.

1. Preamble

   The preamble typically contains the trap and device register definitions and global declarations. In version V03 several new items are required in the handler preamble:

   a. An .MCALL statement is needed for the set of driver macros used in the handler.

      .MCALL .DRBEG,.DRAST,.DREND,.DRFIN

   b. The device size (former contents of $DVSIZ table) and the device status word (contents of $STAT table) must be defined in the preamble, using the mnemonics ddDSIZ and ddSTS. These values are assembled into the handler .ASECT (block 0 of the SYS file) and are extracted from the handler file when needed by the .DSTATUS request.

   c. The default values of handler system generation options can be included in the preamble section. They are not

essential if a system definition file is always included when assembling the handler. Otherwise, assembly errors can occur.

The default definitions currently include:

```
.IIF NDF MMG$T,MMG$T=0     ;NO 18-BIT I/O
.IIF NDF ERL$G,ERL$G=0     ;NO ERROR LOGGING
.IIF NDF TIM$IT,TIM$IT=0   ;NO TIME-OUT
```

d. The .QELDF macro can be invoked to symbolically define all queue element offsets for the specified set of system generation options. .QELDF must be invoked after the system generation options have been defined. See Section 1.4.4.5 for the queue element offset symbolics.

2. SET Options

The option list starts at 400 in the handler .ASECT and is terminated by a zero word. Devices that can be used as the system device can have SET options when they are assembled and linked for use as non-system devices.

The system generation procedure permits the separate assembly of the system device. The SET options should be enclosed in conditionals, being assembled only if the symbol $SYSDV is undefined. The options are not assembled into a system device and the SET command is ineffective. The monitor must be patched to change an option in the system device. Section 1.4.3 describes how to add a SET option to a handler.

3. Header

The header contains standard data in fixed locations used by the monitor when it is interfacing with the handler. The header has two forms; one for a single vector device and one for a multiple vector device.

a. Single-vector handlers

The device handler header is generated by the macro .DRBEG. This macro has the following form:

```
.DRBEG name,vec,dsiz,dstat
```

where:

name — is the two-letter device name.

vec — is the device vector.

dsiz — is the number of 256-word blocks of storage on the volume (0 if non-directory structured); returned to user by .DSTATUS request.

dstat — is the device status word (not to be confused with hardware CSR); returned to user by .DSTATUS request.

This macro generates the handler .ASECT and .PSECT. It also generates any necessary globals, labels and the queue header. The load point of the handler is given the symbolic name ddSTRT. The queue header words have the names ddLQE and ddCQE.

For example: .DRBEG dd,ddVEC,ddDSIZ,ddSTS

.DRBEG RK,220,RKDSIZ,RKSTS

b. Multi-vector handlers

The monitor can load device handlers having more than one vector. This feature facilitates the use of multi-controller devices. In a driver with multiple vectors, the word normally containing the interrupt vector contains an offset to a table of vector triplets. The difference in meaning of the word is flagged by setting bit 15. The first word of the multi-vector handler header is as follows:

```
.WORD <table-.>/2-1+100000
```

where:

table   is a table of vector triplets of the form:

```
VECTOR
TRAP ADDRESS-.
PS
```

The table is terminated with a zero word.

The .DRBEG macro is similar to the single vector version with the addition of a final argument, vtbl.

```
.DRBEG name,vec,dsiz,dstat,vtbl
```

where:

vtbl   is the name of a table of vector triplets in a handler requiring multiple vectors.

For example: .DRBEG PC,PCVEC,PCDSIZ,PCSTS,PTBL

DX, DY, and PC are devices that use this feature.

4. I/O Initiation Section

This section is entered in system state (with context switching inhibited) by the queue manager. All registers are available for use. The queue element to be processed is pointed to by ddCQE. The I/O initiation section must return with a RTS PC.

5. Asynchronous Trap Entry Points

The asynchronous trap entry points consist of the interrupt entry and abort entry. The AST entry point branch table is created by a macro called .DRAST. This macro has the form:

```
.DRAST name,pri[,abo]
```

where:

name   is the two-letter device name.

pri   is the priority at which the interrupt service is to execute.

abo — is the optional abort entry code symbolic label (if not specified, an RTS PC is generated).

The .DRAST macro generates the AST branch table and an .INTEN call for the interrupt service routine. The interrupt routine has the symbolic name ddINT, which is declared global by the macro if the device is to be a system device.

For example:

```
.DRAST RK,5

.DRAST DT,6,DTSTOP
```

In a multi-vector handler, the abort entry point is assumed to precede the interrupt entry point having the label ddINT, where dd is the two-letter device name declared initially in the .DRBEG macro.

6. I/O Completion

A macro called .DRFIN is provided for completing an I/O transfer and returning the queue element. The macro call is:

```
.DRFIN name
```

where name is the two-letter device name.

This macro points R4 to the handler queue head and jumps to the monitor I/O completion routine. Its expansion is identical to the current procedure and it is provided as a shorthand method of completing a transfer. It also serves to isolate system dependencies from the handler code.

For example:

```
.DRFIN RK
```

expands to:

```
MOV     PC,R4
ADD     #RKCQE-.,R4
MOV     @#54,R5
JMP     @270(R5)
```

7. **Handler Termination**

A macro is provided to terminate the device handler code. When invoked, the macro generates a table of pointers to monitor routines (interrupt entry, error logging, etc.), and computes the size of the handler load module for use by .FETCH. The macro call is:

```
.DREND name
```

where name is a two-letter device name.

For example:

```
.DREND RK
```

1.4.3 Adding a SET Option

The keyboard monitor SET command permits certain device handler parameters to be changed from the keyboard. For example, the width of the line printer on a system can be SET with a command such as:

```
SET LP WIDTH=80
```

This is an example of a SET command that requires a numeric argument. Another type of SET command is used to indicate the presence or absence of a particular function. An example of this is a SET command to specify whether an initial form feed should be generated by the LP handler:

```
SET LP FORM               (generate initial form feed)

SET LP NOFORM             (suppress initial form feed)
```

In this case, the FORM option can be negated by appending the NO prefix.

The SET command is entirely driven by tables contained in the device handler itself. Making additions to the list of SET options for a device is easy, requiring changes only to the handler, and not to the monitor. This section describes the method of creating or extending the list of SET options for a handler. The SET command is described in Chapter 4 of the RT-11 System User's Guide.

Device handlers have a file name in the form xx.SYS, where xx is the two-letter device name (for example, LP.SYS). Handler files are linked in memory image format at a base address of 1000, in which a portion of block 0 of the file is used for system parameters. The rest of the block is unused, and block 0 is never FETCHed into memory. The SET command uses the area in block 0 of a handler from 400 to 776 (octal) as the SET command parameter table. The first argument of a SET command must always be the device name; (LP in the previous example command lines). SET looks for a file named xx.SYS (in this case LP.SYS) and reads the first two blocks into the USR buffer area. The first block contains the SET parameter table, and the second block contains handler code to be modified. When the modification is made, the two blocks are written out to the handler file, effectively changing the handler. The SET parameter table consists of a sequence of four-word entries. The table is terminated with a zero word; if there are no options available, location 400 must be zero. Each table entry has the form:

```
.WORD   value
.RAD50  /option/           (two words of Radix-50)
.BYTE   <routine-400>/2
.BYTE   mode
```

where:

value — is a parameter passed to the routine in register 3.

option — is the name of the SET option; for example, WIDTH or FORM.

routine — is the name of a routine following the SET table that does the actual handler modification.

mode — indicates the type of SET parameter:

a. Numeric argument - byte value of 100
b. NO prefix valid - byte value of 200

The SET command scans the table until it finds an option name matching the input argument (stripped of any NO prefix). For the first example command string, the WIDTH entry would be found. The information in this table entry tells the SET processor that O.WIDTH is the routine to call, that the prefix NO is illegal and that a numeric argument is required. Routine O.WIDTH uses the numeric argument passed to it to modify the column count constant in the handler. The value passed to it in R3 from the table is the minimum width and is used for error checking.

The following conventions should be observed when adding SET options to a handler:

1. The SET parameter tables must be located in block 0 of the handler file and should start at location 400. This is done by using an .ASECT 400.

2. Each table entry is four words long, as described previously. The option name may be up to six Radix-50 characters long, and must be left-justified and filled with spaces if necessary. The table terminates with a zero.

3. The routine that does the modification must follow the SET table in block 0. It is called as a subroutine and terminates with an RTS PC instruction. If the NO prefix was present and valid, the routine is entered at entry point +4. An error is returned by setting the C bit before exit. If a numeric argument is required, it is converted from decimal to octal and passed in R0. The first word of the option table entry is passed in R3.

4. The code in the handler that is modified must be in block 1 of the handler file; that is, in the first 256 words of the handler.

5. Since an .ASECT 400 was used to start the SET table, the handler must start with an .ASECT 1000.

6. The SET option should not be used with system device handlers, since the .ASECT will destroy the bootstrap and cause the system to malfunction.

1.4.4 Monitor Services for Device Handlers

The RT-11 monitor provides a set of services for device handlers. These services are located in the resident monitor and can be shared by all device handlers to minimize overall system size and simplify the development and conversion of handlers. The services consist of interrupt entry processing, fork list processing, error logging, request time-out, and extended memory support. The interrupt entry processing and the fork list processing are permanent monitor features. The rest can be included or excluded at SYSGEN time. The following sections discuss the extent of each service and describe when it should be used.

1.4.4.1 Use of .FORK Process - RT-11 provides handlers with the capability of executing code as a serialized, zero-priority system process. This process, called a fork process, is similar to the service provided in other PDP-11 operating systems. A handler can request a fork process while at interrupt level (that is, after the

.INTEN request). The stack must be clean before the .FORK request is issued. That is, the stack must be in the same state when the .FORK request is issued as it was after the .INTEN request was processed. Anything pushed onto the stack after the .INTEN request must be popped off the stack before the .FORK request is issued. Control returns to the line following the .FORK request when the fork request is granted. See Figure 1-2 for a diagram of RT-11's priority structure.

The .FORK request causes the interrupt to be dismissed and adds the driver's request to a first-in/first-out (FIFO) list. The fork queue manager is activated after the last interrupt is dismissed but before the scheduler is called. Drivers are called serially in FIFO order, at priority level 0 and system state (that is, monitor stack, context switching inhibited). Registers R4 and R5 are preserved through the .FORK request, and in addition, registers R0-R3 are available for use at fork level.

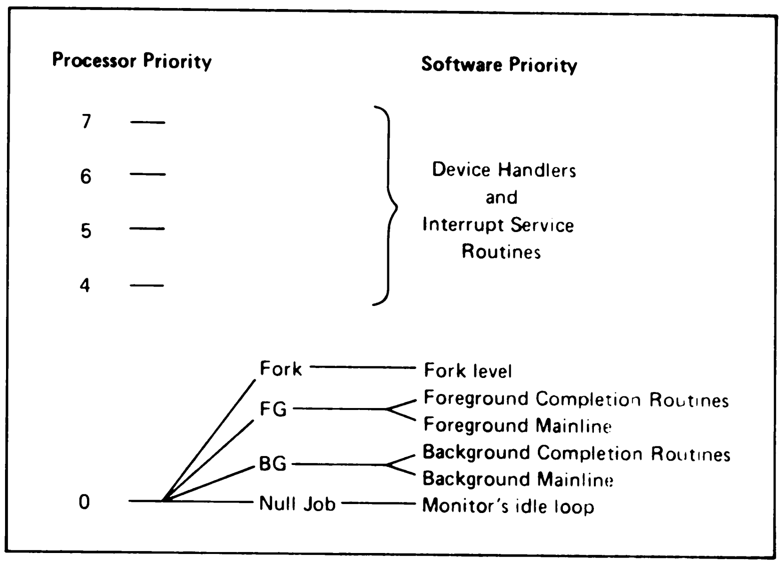


Figure 1-2 RT-11 Priority Structure

The handler must provide a four-word fork queue element that is used to preserve R4, R5 and the return PC while in the fork queue. The first word of the fork queue element is the link word and must be zero when the .FORK request is issued. A non-zero link implies the queue element is in use. However, the monitor does not check this case. This implies that the interrupt service code should check the link word before issuing the .FORK if the code could possibly be used in a re-entrant way.

The .FORK request has the form:

```
.FORK frkblk
```

where:

frkblk is the name of fork list element.

For example:

```
.FORK ddFBLK
```

where:

ddFBLK is defined as

```
ddFBLK: .WORD 0,0,0,0
```

The .FORK request has several applications in a real-time systems environment. It permits lengthy but non-critical interrupt processing to be postponed until all other interrupts are dismissed. Its use in the card reader and line printer drivers solves some of the latency problems encountered in remote batch and DECNET applications.

For example, the card reader driver internally buffers 80 columns of card 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 version 2C, this process took place at priority level six, 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 dropping priority to zero since the card reader can have interrupted a lower priority device. Lowering priority causes re-entrancy problems in the other device drivers. Using a .SYNCH does not always solve the problem. The SJ monitor only simulates a .SYNCH and drops priority to zero, which produces the same re-entrancy problems. The FB monitor must perform a context switch since .SYNCH returns to the caller in user context, running on the user stack. This is a lengthy process and does not occur at all if there is a compute bound foreground job.

The .FORK request is the optimum solution to the problem. It returns at priority zero, but only when all other interrupts have been dismissed and before control is returned to the interrupted user program.

Actual fork support is not provided in SJ unless timer support is generated in the monitor. Instead, the .FORK is simulated to the extent that registers R0-R3 are saved before the driver is called back. Beyond that, no serialization of interrupts is provided.

1.4.4.2 **Use of .SYNCH** - The .SYNCH request is provided to allow device drivers and user interrupt service routines to issue programmed requests. When issued, the .SYNCH request dismisses the interrupt and queues the .SYNCH block provided on the I/O completion queue (in FB and XM monitors). The job is flagged as having an I/O completion routine pending, which causes the scheduler to switch in the job.

This procedure is necessary since programmed requests must be issued in job context, and interrupts occur asynchronously. The .SYNCH request forces a context switch so the code following the .SYNCH runs in job context. In the SJ monitor the .SYNCH request simulates the register manipulations of the FB .SYNCH processor and then returns immediately to the caller at priority level 0. This occurs because the SJ monitor has a single job context and does not use an I/O completion queue. This is the reason the .SYNCH request cannot be used to simulate the functions of the .FORK request in SJ systems.

The .SYNCH request can be issued either after an .INTEN request or after a .FORK request. The handler must not have pushed anything on the stack when the .SYNCH is issued.

The XM monitor must also change the mapping mode when calling I/O completion routines. Regular I/O completion routines are run in user context and user mapping. The .SYNCH routines are run in user context, but the XM monitor requires all interrupt service routines (both user and system handler) to run in kernel mode. Thus, under the XM monitor, the .SYNCH request does not change mapping mode from kernel to user mode, but runs the .SYNCH routine in user context and kernel mapping.

1.4.4.3 **Multi-Vector Support** - A feature is provided to load device handlers having more than one vector. Previously the handler initialization code was required to set up the extra vectors. This feature makes it easier to support multi-vector devices.

The presence of multi-vector support is transparent to single-vector handlers.

The handler header normally has the form:

```
Vector
Word Offset to Interrupt Routine
PS
End of Queue Pointer
Head of Queue Pointer
```

In a handler with multiple vectors, the word containing the interrupt vector contains an offset to a table of vector triplets. The difference in meaning of this word is flagged by setting bit 15. The first word of the handler header contains:

```
.WORD <table-.>/2-1+100000
```

where table is a table of vector triplets of the form:

```
VECTOR
TRAP ADDRESS-.
PS
```

The table is terminated with a zero word. For example, a handler to handle both input and output for a PC11 High Speed Paper Tape Punch/Reader would have a header, generated by .DRBEG, of the form:

```
        .WORD   <PTBL-.>/2-1+100000     ;OFFSET TO TABLE OF VECTORS
        .WORD   PRINT-.                 ;OFFSET TO FIRST INTERRUPT
        .WORD   340                     ;DUMMY PRIORITY
        .WORD   0
        .WORD   0
```

where PTBL has the form:

```
PTBL:   .WORD   70                      ;READER VECTOR
        .WORD   PRINT-.                 ;READER TRAP ROUTINE OFFSET
        .WORD   340
        .WORD   74                      ;PUNCH VECTOR
        .WORD   PPINT-.                 ;PUNCH TRAP ROUTINE OFFSET
        .WORD   340
        .WORD   0                       ;END OF TABLE
```

Note that only the status bits in the PS word specified are actually loaded. The priority is always forced to 7. When a single vector is loaded, the .FETCH code completely ignores the PS word specified, setting the value 340 into the vector PS word.

The macro .DRBEG contains an optional fifth parameter that points to the table.

```
.DRBEG name,vec,dsiz,dstat,vtbl
```

where:

vtbl is the name of a table of vector triplets in a driver requiring multiple vectors.

For example:

```
.DRBEG PC,PCVEC,PCDSIZ,PCSTS,PTBL
```

1.4.4.4 **Error Logging** - Error logging is an option provided to enhance system reliability. Its effective use requires that appropriate device handlers report on their activity so that a log of system I/O activity can be collected and analyzed. Both successful and unsuccessful transfers are logged. Section 1.6 describes error logging in detail. Section 1.6.3.3 describes how to call the error logger from a user-written device handler.

1.4.4.5 **Extended Memory Support for Handlers** - RT-11 supports systems with 128K words of memory. All device handlers, both NPR (non-processor request) and programmed transfer, support extended memory. RT-11 has a set of subroutines that are available to all drivers. There are three routines that move a byte to or from the user buffer or move a word to the user buffer for programmed transfer devices. Another routine converts the buffer address information supplied in the queue element into an 18-bit physical address for NPR devices.

The queue element size for unmapped systems is seven words. However, the queue element size is ten words in the mapped (XM) monitor. The .QELDF macro supplies the queue element offset symbolics and queue element byte size for the appropriate implementation (mapped or unmapped), provided the symbol MMG$T is correctly defined before .QELDF is invoked.

The queue element format in the XM monitor is essentially an extension of the unmapped format. The queue element in the XM monitor requires three additional words. One additional word is required to pass the user buffer address to the handler. The other two words are unused and provided for future expansion without another change in I/O queue element size. The queue element has the following format in the XM monitor:

SYMBOLIC	BYTE OFFSET	CONTENTS
Q.LINK	0	Link to next element
Q.CSW	2	Pointer to channel status word
Q.BLKN	4	Block number
Q.FUNC	6	Special function byte
Q.JNUM	7	Job number
Q.UNIT	7	Unit number
Q.BUFF	10	Displacement to user buffer
Q.WCNT	12	Word count
Q.COMP	14	Completion routine address
Q.PAR	16	Page address register 1 bias to map user buffer (XM only)

The monitor routines that support extended memory are called through pointers in the handler. These pointers are reserved and labelled by the .DREND macro. The monitor fills the pointers with correct absolute addresses at fetch time.

The following are the call sequences and register conditions for invoking the extended memory handler support routines in the XM monitor:

1. Convert Mapped Address to Physical Address

   The monitor routine $MPPHY (Convert Mapped Address to Physical Address) is available to NPR device handlers. It converts the virtual buffer address supplied in the queue element into an 18-bit physical address that is returned on the stack.

Call:	JSR PC,@$MPPTR	
Inputs:	R5	Contains pointer to Q.BUFF in queue element.
Outputs:	2(SP)	Second word on stack contains high order two bits of physical address in bit positions 4 and 5.
	(SP)	First word on stack contains low order 16 bits of physical address.
	R5	Contains pointer to Q.WCNT in queue element.

2. Move Byte to User Buffer

   The routine $PUTBYT in the resident monitor is available to programmed transfer device handlers to transfer a byte passed on the stack to the user buffer. The buffer address in Q.BUFF in the queue element is updated and mapping register overflow is detected and adjusted. The byte count is not modified.

Call: JSR PC,@$PTBYT

Inputs: (SP) First word on stack contains byte of data to be transferred.

R4 Contains pointer to Q.BLKN in queue element.

Outputs: Byte is removed from stack.
Buffer pointer is updated.
R4 is unmodified.

3. Move Byte From User Buffer

The routine $GETBYT is the complement of $PUTBYT. A byte is extracted from the user buffer and returned on the stack. The buffer pointer is updated, but the byte count is not modified.

Call: JSR PC,@$GTBYT

Inputs: R4 Contains pointer to Q.BLKN in current queue element.

Outputs: (SP) First word on stack contains byte of data from user buffer.

Buffer address (Q.BUFF) is updated.
R4 is unmodified.

4. Move a Word to User Buffer

The $PUTWRD routine is available through the $PTWRD pointer and moves a word supplied on the stack to the user buffer. Its anticipated uses are in handlers for analog devices and to return status information.

Call: JSR PC,@$PTWRD

Inputs: (SP) First word on stack contains word of data to move.

R4 Contains pointer to Q.BLKN in queue element.

Outputs: Word of data is removed from stack.
Q.BUFF is updated.
R4 is unmodified.

5. The .DREND macro generates a fifth pointer, $RLPTR, which points to the monitor routine $RELOC. This routine is reserved for use by DIGITAL software only.

1.4.4.6 **Device Time-out Support** - A SYSGEN option adds device time-out support to the monitor. This option permits device handlers to do the equivalent of a mark time without doing a .SYNCH request. Data transfers can be timed, and the driver can take action if the transfers do not complete in the expected time interval.

This feature is not used by any of the RT-11 device handlers. However, it is used by the multi-terminal monitor when the multi-terminal time-out option or remote DZ11 lines are selected during SYSGEN. In these two cases, the device time-out support is automatically included in the monitor during SYSGEN. The device time-out option is also required for DECNET applications. The user must specifically request it in the SYSGEN dialogue when he builds a monitor for a DECNET application.

Two macros can be used only within a device handler. The macros, .TIMIO and .CTIMIO, permit the scheduling and cancelling of a mark time request. They can be issued from the entry point of the handler, from interrupt level, or from a time-out completion routine. The macros are contained in the system macro library, SYSMAC.SML.

To schedule a mark time from a handler:

```
.TIMIO tbk,hi,lo
```

where tbk is the address of a seven-word timer block containing the following:

Word	Contents
0	hi order time
2	lo order time
4	link to next queue element; 0 if none
6	owner's job number
10	owner's sequence number
12	-1 if system timer element -3 if .TWAIT element in XM
14	address of completion routine; zeroed by the monitor when the routine is called to indicate that the timer block is available for reuse.

The .TIMIO request schedules a completion routine to run after the specified number of clock ticks have occurred. The completion routine runs in user context (kernel mapping), associated with the job specified in the timer block. Registers R0 and R1 are available for use. When the completion routine is entered, R0 contains the sequence number of the request that timed out.

To cancel a mark time from a handler:

```
.CTIMIO tbk
```

where tbk is the address of the seven-word timer block used in the .TIMIO request being cancelled.

If the timer request has already timed out and been placed in the completion queue, the .CTIMIO fails, since a timer request cannot be cancelled after being placed in the completion queue. Failure to cancel the queue element is indicated by the C bit set on return from the .CTIMIO request.

1.4.5 Installing and Removing Handlers

The installation and removal of device handlers from the system is done from the keyboard monitor. Two keyboard monitor commands, INSTALL and REMOVE, make the temporary installation of a handler very easy; no patching procedures are required.

The INSTALL command has the following form:

```
.INSTALL dd
```

where dd is the two-letter device (and file) name.

The INSTALL command searches the system device for a file named dd.SYS (or ddX.SYS for XM), extracts the device status word from the handler, and updates the $STAT, $PNAME and $DVREC tables in the resident monitor. The device can now be used without rebooting the monitor.

NOTE

> INSTALL is effective only on the monitor in memory. It does not permanently modify the monitor file on the system device. To permanently install a handler, the system must be patched. This requires patching the Radix-50 name into $PNAME and the device status word into $STAT. Another way is to include the INSTALL command in the startup indirect command file (STARTx.COM) that is executed on every boot. (Note that startup indirect command files are optional.) The monitor file can also be re-SYSGENED.

1.4.6 Converting Handlers to V03 Format

A V02 format device handler requires some conversion to operate under a V03 or later monitor. The conversion effort ranges from a short patch to a complete re-edit, depending on how many new features the user desires. Special device handlers require some extra effort to support the new error reporting capability of the special device interface. This conversion can be implemented in the following ways.

1.4.6.1 Patching a V02 Format Handler - A version V02 driver can be patched to operate under a V03 or later monitor, provided the monitor generated does not support extended memory, error logging or device I/O time-out. Four locations in block 0 of the handler file must be patched to contain handler information essential to the operation of the new .FETCH mechanism.

The four locations contain the handler size, device status word, the device block size (that is, number of 256-word blocks on the volume), and the SYSGEN options compatible with this handler. All handlers have pointers to $INTEN and $FORK and optional pointers to support routines for the SYSGEN options at the end of the handlers, which are initialized when the handler is .FETCHed. Since V02 handlers have only the $INTEN pointer, an extra word (two bytes) must be added to the actual handler size when patching. The other two locations contain the data normally present in the $STAT and $DVSIZ tables (the $DVSIZ table is eliminated in V03 and later releases of RT-11).

Location	Contents
52	Handler size in bytes (plus 2 for $FORK pointer)
54	Device size in number of 256-word blocks
56	Device status word, as contained in $STAT table.
60	SYSGEN options, must be 0

For example, to patch the V02C MT.SYS handler to function under the V03 monitor:

```
.R PATCH

FILE NAME--
*MT.SYS <RET>
*52/    0       4300 <LF>
54/     0       0 <LF>
56/     0       12011 <LF>
60/     0       0 <RET>
*E
```

NOTE

This patch does not work with V03 or later monitors having error logging, extended memory or device time-out support.

1.4.6.2 **Source Edit Conversion of Handlers** - A V02 format, non-system handler can be converted to function with the V03 or later monitors (without .FORK, error logging or extended memory support) by applying a minimal set of edits to the device source. The two essential changes are the addition of the four words described in the first method to the handler .ASECT, and the addition of a dummy .FORK pointer to the end of the handler.

The faster method is to directly edit in the .ASECT and extra word. The better method is to replace the handler header with the .DRBEG macro and insert the .DREND macro at the end of the handler. No problems will be encountered if standard RT-11 naming conventions were used in writing the handler. Neither of these methods takes full advantage of the new features of RT-11.

NOTE

To convert a version 2C device handler to version 3, change the version 2C device handler so that it sets the EOF bit in the channel status word in the proper sequence. (See Section 1.4.1.) If this change is not made, the last block of data may be lost during a data transfer.

a. (Fast Method)

Step 1: Define the device handler size, block size and status word.

For example:

```
RKDSIZ = 0
RKSTS = 20003
```

The driver size is usually defined at the end of the handler using the convention:

```
RKHSIZ = .-RKSTRT
```

Step 2: Install the handler .ASECT.

For example:

```
.ASECT
.=52
      .WORD RKHSIZ
      .WORD RKDSIZ
      .WORD RKSTS
      .WORD 0
```

Add a .CSECT after the .ASECT if one is not already in the existing handler code.

Step 3: Add a dummy $FKPTR to the end of the handler.

For example:

```
$INPTR:   .WORD 0
RKHSIZ = .-RKSTRT
```

becomes

```
$INPTR:   .WORD 0
$FKPTR:   .WORD 0
RKHSIZ = .-RKSTRT
```

b. (Best Method)

Perform steps 1, 2, 3, 4 and 7 of the full conversion method.

1.4.6.3 **Full Conversion of Device Handlers** - To take advantage of the new features, the handler must be modified. Inserting the .DRBEG, .DRAST, .DRFIN and .DREND macros makes conversion to V03 format easier, but it does not supply the functional conversion necessary to support error logging or extended memory. Difficulty of functional conversion varies with the complexity of the device and its handler.

To make the full conversion of a device handler, perform the following:

1. Insert an .MCALL containing the handler macros that are to be used in converting the handlers.

   For example:

   ```
   .MCALL .DRBEG,.DRAST
   .MCALL .DRFIN,.DREND,.QELDF
   .QELDF
   ```

2. Insert the default system build options:

   For example:

   ```
   .IIF NDF MMG$T,MMG$T=0
   .IIF NDF ERL$G,ERL$G=0
   .IIF NDF TIM$IT,TIM$IT=0
   ```

3. Define the device block size and status words using the proper mnemonics.

   For example:
   ```
   RKDSIZ = 0
   RKSTS = 20003
   ```

4. Replace the handler header with the .DRBEG macro.

   For example:
   ```
   RKSTRT: .WORD    200
           .WORD    RKINT-.
           .WORD    340
   RKSYS:
   RKLQE:  .WORD    0
   RKCQE:  .WORD    0
   ```

is replaced by the macro:

```
.DRBEG RK,200,RKDSIZ,RKSTS
```

5. Replace the interrupt entry and abort entry points with the .DRAST macro (optional, but recommended).

   For example: replace the code:

```
        BR      RKDONE          ;ABORT ENTRY POINT
RKINT:  JSR     R5,@$INPTR      ;INTERRUPT ENTRY POINT
        .WORD   ^C<PR5>&340
```

   with the macro:

```
        .DRAST RK,5,RKDONE
```

6. Replace the I/O completion code with the .DRFIN macro (optional, but recommended).

   For example:

   replace the code:

```
        MOV     PC,R4
        ADD     RKCQE-.,R4
        MOV     @#54,R5
        JMP     @270(R5)
```

   with the macro call:

```
        .DRFIN RK
```

7. Replace the $INPTR location at the end of the handler with the .DREND macro.

   For example: replace:

```
$INPTR: .WORD   0
RKHSIZ = .-RKSTRT
```

   with:

```
        .DREND  RK
```

8. The handler can now be assembled and tested. Assembly errors can occur if RT-11 naming conventions were not followed (for example, if the queue pointers were not originally named RKLQE and RKCQE, the start of the CSECT was not named RKSTRT, and the interrupt entry point was not named RKINT). The handler should now function correctly under the SJ and FB monitors, provided that the monitors have not been SYSGENed to include any other handler features like error logging and device time-out.

9. Extended memory conversion can now be done, if desired.

   a. NPR (Non-Processor Request) Devices

      Assumptions: R5 is used to point to the queue element.

      Procedure: The buffer address supplied in the queue element in a mapped monitor is really in two parts. Q.BUFF contains the buffer displacement in the virtual address space defined by Q.PAR. This must be converted to an 18-bit physical address, which is done by a call through $MPPTR. Two words are returned on the stack, containing the low order 16 bits and high order two bits.

For example:

```
        RKCS = nnnnn2           ;CONTROL AND STATUS
                                ;REGISTER
        RKWC = nnnnn4           ;WORD COUNT REGISTER
        RKBA = nnnnn6           ;UNIBUS ADDRESS REGISTER
            .
            .
            .
        MOV     #103,R3         ;ASSUME A WRITE
        MOV     #RKBA,R4        ;R4 -> BUFFER ADDRESS REG
        MOV     (R5)+,(R4)      ;MOVE BUFFER ADDRESS
        MOV     (R5)+,-(R4)     ;MOVE WORD COUNT
```

is replaced with the conditional code:

```
        RKCS = nnnnn2           ;CONTROL AND STATUS
                                ;REGISTER
        RKWC = nnnnn4           ;WORD COUNT REGISTER
        RKBA = nnnnn6           ;UNIBUS ADDRESS REGISTER
            .
            .
            .
.IF EQ MMG$T
.IFTF
        MOV     #103,R3         ;ASSUME A WRITE
        MOV     #RKBA,R4        ;R4 -> BUFFER ADDRESS REG
.IFT                            ;IF UNMAPPED
        MOV     (R5)+,@R4       ;MOVE BUFFER ADDRESS
                                ;TO RKBA
.IFF                            ;IF MAPPED
        JSR     PC,@$MPPTR      ;CONVERT TO 18 BITS
        MOV     (SP)+,@R4       ;MOVE LOW 16 BITS TO RKBA
.IFTF                           ;IN ANY CASE,
        MOV     (R5)+,-(R4)     ;MOVE WORD COUNT TO RKWC
    .
    .
    .
.IFF                            ;IF MAPPED
        BIS     (SP)+,R3        ;SET IN HI ORDER
                                ;ADDRESS BITS
.IFTF                           ;IN ANY CASE
  6$:   MOV     R3,-(R4)        ;START THE OPERATION
        RTS     PC              ;AWAIT INTERRUPT
.ENDC
```

For NPR devices which may be interfaced to a mass bus controller, the address extension bits must be placed in bits 8 and 9 of the control and status register rather than bits 4 and 5. For these devices (such as RJS03/04) the code above must be modified to shift the bits into place.

```
.IFF                            ;IF MAPPED
        JSR     PC,@$MPPTR      ;CONVERT TO 18 BITS
        MOV     (SP)+,@R4       ;MOVE LOW 16 BITS
        ASL     (SP)            ;SHIFT HI BITS INTO PLACE
        ASL     (SP)            ;
        ASL     (SP)            ;
        ASL     (SP)            ;
        BIS     (SP)+,R3        ;SET IN HI ORDER BITS
```

b. Programmed Transfer Devices

Assumptions: R4 points to Q.BLKN in the queue element.

Procedure: Programmed transfer devices must directly move the data to or from the user buffer. This is usually done a byte or word per interrupt, but sometimes a complete buffer is moved, as in the CR handler.

To move data the handler must save the contents of the kernel mapping register* (page address register 1), move Q.PAR to kernel page address register 1, and then move one byte or word indirectly off the contents of Q.BUFF. If more than 4K-32 words of data can be moved, the Q.BUFF address must be checked for overflow each time it is updated, since a page address register can map only 4K words of memory. A simple approach is to use one of the monitor routines provided.

For example, the original handler contains the code:

```
BYTCNT = 6          ;OFFSET TO BYTE COUNT
BUFF = 4            ;OFFSET TO BUFFER ADDRESS
    .
    .
    .
MOV       PPCQE,R4                   ;R4 -> Q.BLKN
    .
    .
    .
MOVB      BUFF(R4),@#PPB             ;MOVE A CHARACTER
INC       BUFF(R4)                   ;UPDATE BUFFER ADDRESS
INC       BYTCNT(R4)                 ;BUMP BYTE COUNT
BEQ       PPDONE                     ;IF EQ DONE
```

which becomes the conditionalized code:

```
        BYTCNT = 6                   ;OFFSET TO BYTE COUNT
        BUFF = 4                     ;OFFSET TO BUFFER ADDRESS
            .
            .
            .
        MOV       PPCQE,R4           ;R4 -> Q.BLKN
            .
            .
            .
.IF EQ MMG$T                         ;IF UNMAPPED
        MOVB      BUFF(R4),@#PPB     ;MOVE CHARACTER
        INC       BUFF(R4)           ;UPDATE BUFFER ADDRESS
.IFF                                 ;IF MAPPED
        JSR       PC,@$GTBYT         ;GET A CHARACTER
        MOVB      (SP)+,@#PPB        ;PUT IT OUT.
.IFTF                                ;IN EITHER CASE,
        INC       BYTCNT(R4)         ;BUMP BYTE COUNT
        BEQ       PPDONE             ;IF EQ DONE
.ENDC
```

There are cases where the monitor subroutines cannot be used. In those cases, the remapping of the kernel mapping register (page address register 1) must be done within the handler code.

* For an explanation of mapping registers, refer to Chapter 3.

The call to $GTBYT is equivalent to the following in-line code sequence:

```
        KISAR1 = 172342                       ;KERNEL PAR1
        MOV    @#KISAR1,-(SP)                 ;SAVE PAR1
        MOV    Q.PAR-Q.BLKN(R4),@#KISAR1      ;MAP TO USER BUFFER
        MOVB   @Q.BUFF-Q.BLKN(R4),@#PPB       ;MOVE NEXT BYTE
        MOV    (SP)+,@#KISAR1                 ;RESTORE PAR1
        INC    Q.BUFF-Q.BLKN(R4)              ;UPDATE BUFFER ADDRESS
        BIT    #40000,Q.BUFF-Q.BLKN(R4)       ;OVERFLOWS 4K LIMIT?
        BEQ    1$                             ;IF EQ, NO
        SUB    #20000,Q.BUFF-Q.BLKN(R4)       ;ADJUST DISPLACEMENT
        ADD    #200,Q.PAR-Q.BLKN(R4)          ;AND PAR1 BIAS
1$:
```

1.4.7 Device Handler Program Skeleton Outline

The following code illustrates a device handler outline. In the example the designation SK is used as the device name.

```
.TITLE  SK V03.01

; SK DEVICE HANDLER

.IDENT  /V03.01/

.SBTTL  PREAMBLE SECTION

.MCALL  .QELDF, .DRBEG, .DRAST, .DRFIN, .DREND, .FORK

; SYSGEN DEFAULT DEFINITIONS:
.IIF NDF MMG$T, MMG$T = 0
.IIF NDF ERL$G, ERL$G = 0
.IIF NDF TIM$IT, TIM$IT = 0

; DEVICE UNIBUS ADDRESSES:
.IIF NDF SK$VEC, SK$VEC = 200             ;SK VECTOR
.IIF NDF SK$CSR, SK$CSR = 177514          ;SK CONTROL STATUS REGISTER
        SKBR    = SK$CSR+2                ;SK BUFFER REGISTER

        HDERR   = 1                       ;HARD ERROR ON CHANNEL

; DEVICE STATUS INFORMATION:
SKDSIZ  = 0                               ;DEVICE BLOCK SIZE
SKSTS   = 20003                           ;DEVICE STATUS WORD

; DEFINITION OF Q ELEMENT SYMBOLICS:
.QELDF
WCNT    = Q.WCNT - Q.BLKN
BUFF    = Q.BUFF - Q.BLKN


.SBTTL  SET OPTIONS

.ASECT

. = 400

        NOP
        .RAD50  /RANDOM/
        .WORD   <O.RNDM-400>/2+100000

        .WORD   0                 ;END OF LIST
```

```
O.RNDM: MOV     (PC)+,R3           ;GET NEW INSTRUCTION TO STORE
         MOVB   SP,RO              ;CHANGE INST FOR SET OPTION
        MOV     R3,SKOPT           ;STORE IT IN HANDLER BODY
        RTS     PC                 ;DONE WITH SET OPTION CHANGE


.SBTTL  HEADER SECTION

        .DRBEG  SK,SK$VEC,SKDSIZ,SKSTS

; ENTRY POINT FORM QUEUE MANAGER

        MOV     SKCQE,R4           ;R4 -> CURRENT QUEUE ELEMENT
        ASL     WCNT(R4)           ;MAKE WORD COUNT A BYTE COUNT
        BCC     SKERR              ;A READ REQUEST IS ILLEGAL
        BEQ     SKDONE             ;A SEEK COMPLETES IMMEDIATELY
RET:    BIS     #100,@#SK$CSR      ;ENABLE INTERRUPTS
        RTS     PC                 ;EXIT AND WAIT FOR ONE


.SBTTL  INTERRUPT TRAP PROCESSING

        .DRAST  SK,4,SKDONE

; INTERRUPT SERVICE:

.IF EQ MMG$T
.IFTF
        MOV     SKCQE,R4           ;;;R4 -> CURRENT QUEUE ELEMENT
        TST     @#SK$CSR           ;;;ERROR?
        BMI     RET                ;;;YES IF MI, HANG UNTIL CORRECT
        TSTB    @#SK$CSR           ;;;IS DEVICE READY?
        BPL     RET                ;;;NO IF PL, EXIT AND WAIT
        CLR     @#SK$CSR           ;;;YES, DISABLE INTERRUPTS

; PROCESS REMAINING CODE AT FORK LEVEL

        .FORK   SKFBLK             ;REQUEST FORK PROCESS
SKNEXT: TSTB    @#SK$CSR           ;READY FOR ANOTHR CHARACTER?
        BPL     RET                ;BR IF NOT READY
        TST     WCNT(R4)           ;ANY LEFT TO PRINT?
        BEQ     SKDONE             ;NO IF EQ, XFER IS DONE
.IFT
        MOVB    @BUFF(R4),R5       ;GET A CHARACTER
        INC     BUFF(R4)           ;BUMP BUFFER POINTER
.IFF
        JSR     PC,@$GTBYT         ;GET A CHARACTER
        MOV     (SP)+,R5           ; INTO R5
.IFTF
        INC     WCNT(R4)           ;BUMP CHARACTER COUNT
        MOV     #177770,R5         ;7 BIT ASCII
SKOPT:  NOP                        ;RANDOM OPTION BY SET COMMAND
        MOVB    R5,@#SKBR          ;PUT IT OUT TO DEVICE
        BR      SKNEXT             ;TRY FOR ANOTHER
.ENDC
```

```
.SBTTL  I/O COMPLETION SECTION

SKERR:  BIS     #HDERR,@Q.CSW-Q.BLKN(R4)
                                ;SET ERROR BIT IN CHANNEL
SKDONE: BIC     #100,@#SK$CSR   ;DISABLE INTERRUPTS
        .DRFIN  SK              ;GO TO I/O COMPLETION

SKFBLK: .WORD   0,0,0,0         ;FORK QUEUE ELEMENT

        .DREND  SK

.END
```

1.4.8 Programming for Specific Devices

This section discusses specific devices that have operating and/or programming techniques and features unique or different from most peripheral devices. Included in this category are the following:

1. Magtape - TM11-Type Controllers (TMA11/TS03,TM11/TU10,TMB11)
   TJU16-Type Controllers (TJU16/TM02/TU16,TJE16/TM03/TE16,TU45).

2. Cassette - TA11

3. Diskette - RX11/RXV11 RX01; RX211/RXV21 RX02

4. Disk - RK611 RK06,RK07; RL11/RLV11 RL01

In addition to these devices, mention is also made of some other devices and other device characteristics.

1.4.8.1 **Magnetic Tape Handlers (MM,MT)** - The magtape device has a file structure that is different from other RT-11 devices. The magtape device handler is capable of supporting a file structure compatible with ANSI magnetic tape labels and tape format. This allows the user full access to the controller without being totally familiar with the device.

NOTE

> It should be noted that RT-11 magtape file structure support is only compatible among systems that support DEC and ANSI standards for magtape labels and tape format. Hence, DOS formatted magtape cannot be read or written.

The handler consists of two versions. One version is the hardware handler (MMHD.SYS,MTHD.SYS), which is designed to accept hardware requests only. This type of handler is useful in I/O operations where no file structure exists. Any file-structure request to the hardware handler results in a monitor directory I/O error. The user accesses the hardware handler with a non-file-structured .LOOKUP (see Chapter 2 for details), special function .SPFUN, .READx/.WRITx*, and .CLOSE requests. The hardware handler contains code to accomplish basic

---

* The term .READx/.WRITx refers to the following group of programmed requests: .READ, .READC, .READW, .WRITE, .WRITC, WRITW.

input/output functions on physical blocks, tape positioning, error recovery and other hardware functions. The other version of the magtape device handler combines the hardware handler with a file-structure module to produce MM.SYS and MT.SYS. The file-structure module provides the handler with the capability to accept file-structure requests. It is designed so that it can be used with any hardware handler. The magtape handler supports up to eight drives and one controller, and operates under all RT-11 monitors. The file-structure version is desirable in most circumstances and is the only one that works with system utilities. The hardware handler is for users with special requirements. Both file-structure and hardware handlers are delivered on the system disk distribution media. The file-structure handler is distributed supporting drives 0 and 1. More drives can be supported as a SYSGEN option. The file-structure handler is the standard version (MT.SYS or MM.SYS) and the hardware handler must be renamed to be used, as shown below:

```
.REMOVE MT                    !Remove from device table

.RENAME/SYS MT.SYS MTFS.SYS !Save file-structure handler

.RENAME/SYS MTHD.SYS MT.SYS !Create new magtape handler
```

File-Structure Handler Functions

The file-structure handler searches through sequence numbers. The file-structure handler performs file searches using the file sequence number (FSN) to determine the tape's current position relative to where the tape has to go to be at the desired file. When the handler receives a sequence number, it compares it to the known position according to the following algorithm:

1. When the file sequence number for the file desired is greater than the current position, the tape simply searches in a forward direction.

   For example:

	Current Position	**File Desired**
	FSN=1	FSN=2

   Tape moves forward from its position at the tape mark after file #1 to the tape mark at the start of file #2.

2. When the file sequence number for the file desired is less than the current position of the tape by greater than two and/or less than five files from the beginning of tape (BOT), the tape is rewound and searching begins in the forward direction. Otherwise, the tape is searched in the backward direction. This procedure utilizes the optimum seek time for file searching on magtape.

   For example:

	Current Position	**File Desired**
Casel:	FSN=2	FSN=1

   The tape drive leaves its position at the tape mark for file #2, and rewinds to the beginning of tape; it then moves forward to the tape mark at the start of file #1.

Case2:	FSN=9	FSN=7

   The tape drive rewinds to the beginning of tape and searches the tape in the forward direction.

3. When the file sequence number for the file desired is the same as the current position or one file away from the current position, the tape is searched in the backward direction.

   For example:

	Current Position	File Desired
Case 1:	FSN=6	FSN=6

   The tape drive leaves its position at the tape mark at the end of file #6, and backspaces to the tape mark following file #5.

Case 2:	FSN=5	FSN=4

   The tape drive leaves its position at the tape mark at the end of file #5, and backspaces to the tape mark following file #3.

If the user .UNLOADs or .RELEASEs the handler, the file position is lost for the file-structure handler. Hence, in this situation the tape moves in a backward direction until it locates the beginning of tape or a label from which the tape's position can be determined.

The file-structure handler searches through file names. The routine to match file names uses an algorithm that enables recognition of file names and file types written by other DIGITAL systems. The method for doing this applies in the algorithm discussed below to the file identifier field, which translates the contents to a recognizable file name. This file name is matched to a file name translated into a Radix-50 format.

The format is:

    filnam.typ

where

filnam
: is a legal RT-11 file name left justified into a six character field and padded with spaces, if necessary.

typ
: is a file type left justified into a three-character field.

The algorithm used is compatible with the DIGITAL standard. It allows tapes written under RT-11 V02C and earlier versions to be read by V03 and later versions and matched (these tapes don't have a dot to separate the file name from the file type). RT-11 format tapes are detected by the presence of "RT11" in character positions 64-67 of the HDR1 label.

The algorithm is as follows:

1. Clear the character count (CC).

2. Look at the first character in the file name; if it is a dot then do the following:

   a. Mark a dot found.
   b. When CC < 6 then insert spaces and increment the CC until CC = 6.
   c. When CC > 6 then delete characters and decrement the CC until CC = 6.

3. When CC = 6 and if "RT11" is found in character positions 64-67 of the system code field, then 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 a dot was not found, back up four characters and insert ".DAT" for the file type.

8. Now perform a character by character comparison between the file name being looked for and the file name that was just translated from the file identifier field in the HDR1 label. When they match exactly, then the file name is found.

1. .ENTER Request - The .ENTER requests an HDR1 label (file header label) and tape mark to be written on tape and leaves the tape positioned after the tape mark. The .ENTER 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 a .SPFUN request is performed on that channel. Normally, files opened with an .ENTER do not have .SPFUN requests performed on them. An exception to this rule is the case where a non-standard block size is to be written (a block size that is not 512 bytes long). To write a non-standard block, the file must be opened with an .ENTER request; then an .SPFUN write request must be performed. The file must be closed with a .CLOSE request after the operation is complete. If a file search is to be performed, the file is opened with a .LOOKUP request. The .ENTER request has the following form:

   .ENTER area,chan,dblk,,seqnum

Table 1-1
Sequence Number Values for .ENTER Requests

Seqnum argument	File name	Action Taken	Position
>0	not null	Position at file sequence number and do a .ENTER	Found: tape is ready to write Not Found: tape is at logical end of tape (LEOT). LEOT is an end-of-file 1 label followed by two tape marks. LEOT is different from the physical end of tape.

(continued on next page)

Table 1-1 (Cont.)
Sequence Number Values for .ENTER Requests

Seqnum argument	File name	Action Taken	Position
0	not null	Rewind tape and search tape for file name. If found then give error. If not found then enter the file	Found: tape is positioned before file Not Found: tape is positioned ready to write
-1	not null	position tape at logical end of tape and enter file	tape is positioned ready
-2	not null	Rewind tape and serach tape for file name. Enter file at found file or logical end of tape, whichever comes first.	tape is positioned ready to write
0	null	do a non-file-structured .LOOKUP	tape is rewound

The .ENTER request returns the following errors.

Byte 52 Code	Explanation
0	Channel in use
1	Device full. Issued if physical end of tape (EOT) detected while writing HDR1. Tape is positioned after first tape mark following the last end-of-file 1 label on the tape.
2	Device already in use. Issued if magtape already has a file open.
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 of -3 through -32,767 was detected. A null file name was passed to enter.

The .ENTER request issues a directory hard error if errors occur while entering the file.

2. .LOOKUP Requests - 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. The .LOOKUP request has the following forms:

```
.LOOKUP area,chan,dblk,seqnum
```

Table 1-2
Sequence Number Values for .LOOKUP Requests

Seqnum argument	File name	Action Taken	Position
-1	null	do a non-file-structured .LOOKUP	Tape is not moved.
>0	null	do a file-structured .LOOKUP on the file sequence number	If operation succeeds, tape is ready to read 1st data block. If the file sequence number is not found, tape is at logical end of tape.
0	not null	rewind to the beginning of tape, then use file name to do a file-structured .LOOKUP	If found, tape is ready to read 1st data block. If file name not found, tape is at logical end of tape.
-1	not null	don't rewind; just do a file-structured .LOOKUP for a file name	If found, tape is ready to read 1st data block. If not found, tape is at logical end of tape.
>0	not null	position at file sequence number and do a file-structured .LOOKUP. If file name does not match file name given, give error.	If found, tape is ready to read 1st data block. If not found, tape is at logical end of tape.

NOTE

If a channel is opened with a non-file-structured .LOOKUP (file name null and file sequence number=0 or -1), .READx requests use an implied word count equal to the physical block size on the tape and .WRITx requests use the word count to determine the block size on the tape. This convention is used instead of using 512 as a default block size and doing blocking/deblocking. This request is almost identical to a .SPFUN read or write which does not report any errors (blk=0). Also note that the error and status block must not be overlaid by the USR.

The .LOOKUP request returns the following errors.

Byte 52 Code	Explanation
0	Channel in use
1	File not found. Tape is positioned after the first tape mark following the last end of file on the tape.
2	Device in use. Issued if the magtape has a file already open.
5	Illegal argument error. A seqnum argument in the range of -2 through -32,767 was detected. A .LOOKUP to the hardware handler must have a positive seqnum.

This request issues the directory hard error in the same manner as the .ENTER request discussed previously.

NOTE

The term .READx/.WRITx refers to the following group of programmed requests: .READ, .READC, .READW, .WRITE, .WRITC, .WRITW.

3. .READx Requests - The .READx request reads 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's (see Section 8 under Hardware Handler Functions). If a request is issued that is less than 512 bytes, then the correct number of bytes is read. If a request is greater than 512 bytes, the handler performs the request with multiple 512 byte requests (or less for the last request if the number of bytes does not equal an exact multiple of 512). The .READx is valid in a file opened with a .LOOKUP request. It is also valid in a file opened with a .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 request is issued to a file opened by a .ENTER request, the tape is not positioned after the last block written. This could cause loss of information if the user issued 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 rules for block numbers are as follows:

   a. .READx - When a .LOOKUP is used (to search file) with this request, the tape drive tries to position the tape at the indicated block number. When it cannot, a 0 (end of file code) error is issued, and the tape is positioned after the last block on the file.

b. .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.

This request has the form:

```
.READx area,chan,buf,wcnt,blk[,crtn]
```

The .READx request returns the following errors.

Byte 52 Code	Explanation
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.

4. .WRITx Requests - The .WRITx request writes data to magtape in blocks of 512 bytes. If a request is issued that is less than 512 bytes, the tape drive forces the writing of 512 bytes from the given buffer address. If a request is issued that is greater than 512 bytes, then the handler performs multiple 512 bytes per block requests.

The .WRITx request is only valid in a file opened with a .ENTER or a non-file-structured .LOOKUP. The .WRITx request has the following form:

```
.WRITx area,chan,buf,wcnt,blk[,crtn]
```

The .WRITx request returns the following errors.

Byte 52 Code	Explanation
0	End of tape (means that the data was not written but the previous block is valid and the file can be .CLOSEd). Also issued if the block number is too large.
1	Hard error occurred on channel
2	Channel not open

It should be noted that no operation other than a write operation can be performed beyond the last block written on tape (see Figure 1-3). Note that the head is positioned in a gap between operations.

a. In example 1, blocks A, B and C are written on the tape. Now the head is positioned in the gap immediately following block C. Any forward operation of the tape drive except write commands (that is, write, erase gap and write, or write tape mask) yields undefined results due to hardware restrictions.

b. In example 2, the head is shown positioned at beginning of tape after a rewind operation. Now 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 case 1 above are applicable.

c. In example 4, a rewind operation was performed followed by a write. New data (block D) replaced the old data (block A) data and now the head is positioned in the gap immediately following block D. Since block D is now the last block written on tape (in the current time frame), blocks B and C cannot be read and this data cannot be recovered. As in previous examples, the magtape handler can only accept write requests at this point.

5. .DELETE and .RENAME Requests - The .DELETE and .RENAME requests are illegal operations on magtape, and any attempt to execute them results in an illegal operation code (2) being returned in byte 52.

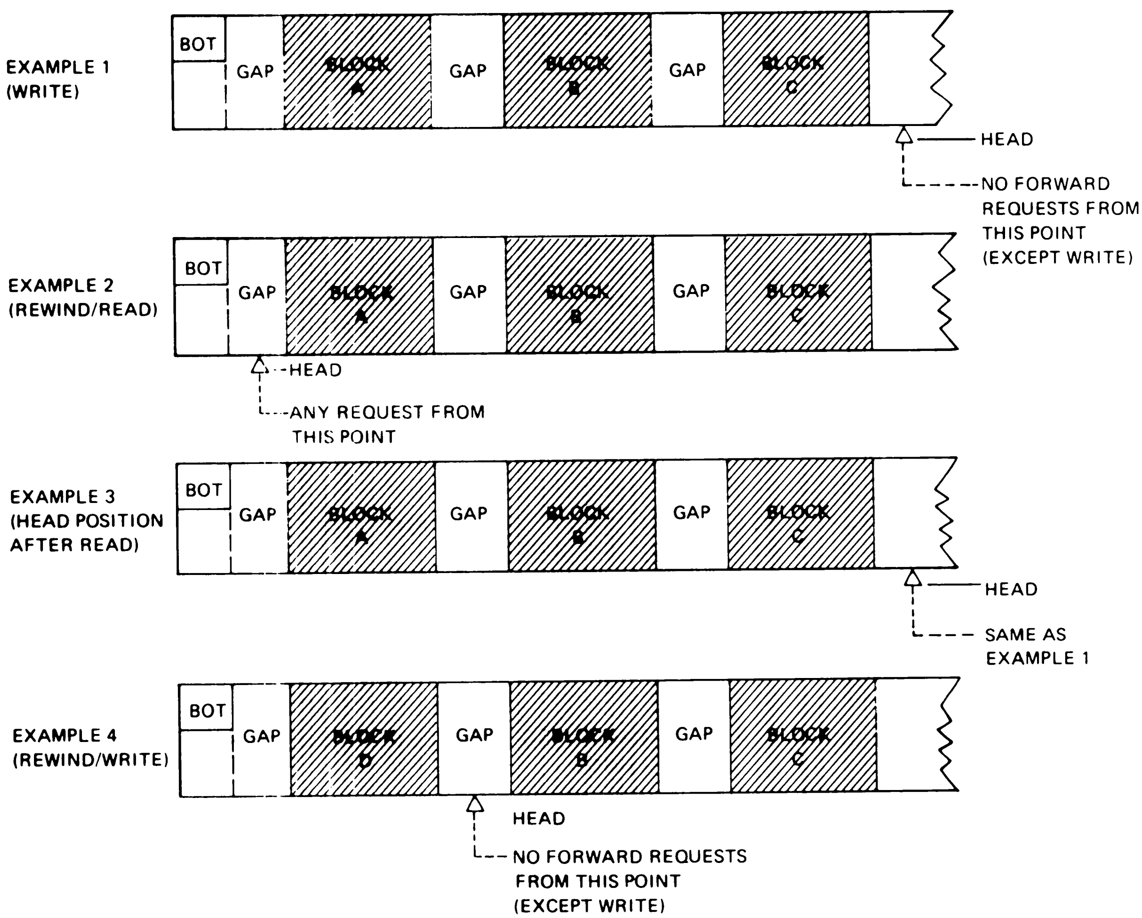


Figure 1-3 Examples of Operations Performed After the Last Block Written on Tape

6. .CLOSE Requests - The .CLOSE request operates in the following three ways:

a. When a file is opened with a .ENTER request, the file is closed by writing a tape mark, an end-of-file 1 label and then three more tape marks. In this operation, the tape drive is left positioned just before the second tape mark at logical end of tape.

b. When a file is opened with a file-structured .LOOKUP, the tape is positioned after the tape mark following the end-of-file 1 label for that file.

c. 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 form:

```
.CLOSE chan
```

This request issues a directory hard error if a malfunction is detected. The error can be recovered with the .SERR request.

7. Asynchronous Directory Operations Request - The asynchronous directory operation request performs directory operations without the USR. This request can be used for long tape searches without tying up the USR. It is provided for users of multi-user 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 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 has been issued to assign a channel to the magtape handler. Indeterminate 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 form:

```
.SPFUN  area,chan,-20.,buf,,blk
```

where:

-20. (decimal) is the code for the synchronous directory request.

buf is the address of a seven-word block in the following format:

Word	Meaning
0 through 2	Radix-50 representation of the file name.
3	Code which is one of the following: LOOKUP=3 ENTER=4
4	Sequence number value. See the corresponding sections for .LOOKUP or .ENTER for complete information on the interpretation of this value.
5,6	Reserved

The blk argument 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 running in the extended memory monitor.

Example:

```
                                 .TITLE ASYNCHRONOUS DIRECTORY OPERATION REQUEST EXAMPLE X01.01

                                 .MCALL  .LOOKUP,.SPFUN,.CLOSE,.PRINT,.EXIT

                                 ;DEFINITIONS

  000000

         177754                  ASYREQ  =       -20.    ;ASYNCHRONOUS REQUEST CODE
         000003                  LOOKUP  =       3       ;LOOKUP CODE FOR ASYNCHRONOUS REQUEST
         000004                  ENTER   =       4       ;ENTER CODE FOR ASYNCHRONOUS REQUEST
         000000                  CHAN    =       0       ;USE CHANNEL 0
         000001                  FNF     =       1       ;FILE NOT FOUND ERROR
         000000                  FSN     =       0       ;USE 0 FOR FILE SEQUENCE NUMBER

                                 ;MAGTAPE HANDLER IS ASSUMED TO BE LOADED.

  000000                         START:  .LOOKUP #AREA,#CHAN,#NFSBLK     ;OPEN A CHANNEL FOR THE NEXT REQUEST
  000024 103433                          BCS     LOOKER                  ;ERROR OCCURRED
  000026                                 .SPFUN  #AREA,#CHAN,#ASYREQ,#COMBLK,,#ERRBLK    ;DO A LOOKUP
  000074 103012                          BCC     FOUND                   ;NO ERRORS MEANS FILE WAS FOUND

  000076 022767 000001 000104            CMP     #FNF,ERRBLK     ;FILE NOT FOUND ERROR?
  000104 001411                          BEQ     NOTFIND         ;YES
  000106 012700 000320'                  MOV     #ASYERR,R0      ;NO
  000112 000410                          BR      CLOSE

  000114 012700 000220'          LOOKER: MOV     #LOOERR,R0      ;NFS .LOOKUP ERROR
  000120 000405                          BR      CLOSE

  000122 012700 000264'          FOUND:  MOV     #OK,R0          ;FILE FOUND MESSAGE
  000126 000402                          BR      CLOSE

  000130 012700 000300'          NOTFND: MOV     #NOK,R0         ;FILE NOT FOUND MESSAGE

  000134                         CLOSE:  .PRINT                  ;PRINT MESSAGE POINTED TO BY R0
  000136                                 .CLOSE  #CHAN
  000144                                 .EXIT

                                 ;DATA AREA

  000146                         AREA:   .BLKW   6               ;EMT ARGUMENT AREA
  000162 052140                  NFSBLK: .RAD50  /MT/            ;USE THIS TO OPEN MAGTAPE NON FILE STRUCTURED
  000164 000000 000000 000000            .WORD   0,0,0
  000172 023364 053665 100370 COMBLK:    .RAD50  /FILNAMTYP/     ;THIS IS THE FILE NAME WE'RE LOOKING FOR
  000200 000003                          .WORD   LOOKUP          ;THIS IS THE ASYNCHRONOUS OPERATION CODE FOR LOOKUP
  000202 000000                          .WORD   FSN             ;THIS IS THE FILE SEQUENCE NUMBER FOR THE LOOKUP
  000204 000000 000000                   .WORD   0,0             ;RESERVED (MUST BE ZERO)
  000210 000000 000000 000000 ERRBLK:    .WORD   0,0,0,0         ;PICK UP ERRORS HERE
  000216 000000

                                 ;MESSAGE AREA

  000220    116    117    116 LOOERR:    .ASCIZ  'NON FILE STRUCTURED .LOOKUP FAILED.'
  000264    106    111    114 OK:        .ASCIZ  'FILE FOUND.'
  000300    106    111    114 NOK:       .ASCIZ  'FILE NOT FOUND.'
  000320    101    123    131 ASYERR:    .ASCIZ  'ASYNCHRONOUS REQUEST ERROR.'

         000000'                         .END    START
```

Hardware Handler Functions

The hardware handler functions can be used with or without the file structure module.

1. Issuing hardware handler calls in a magtape file

   The magtape handler is designed to perform two distinct types of access. One type of access is file oriented and it attempts to make the magtape act like a disk; in other words, to make the magtape device be as device independent as possible. The other type of access allows access to the hardware commands such as read, write, space, etc., but the user doesn't have to know whether the magtape is a TM11 or TJU16.

   When accessing magtape using file oriented commands, the handler keeps track of the file sequence number where the tape is positioned. Tape movement during file searches can be optimized.

   When accessing data in a magtape file using the .READx/.WRITx requests, the magtape handler 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 .ENTERed files.

The two access methods described above can be combined; that is, it is possible to use hardware handler tape movement commands on a magtape file. However, doing so causes the following to happen.

a. 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 .ENTERed file case, writes the tape no matter what commands have been issued since the file was opened. Also note that the tape will no longer be an ANSI compatible tape. When the file is .CLOSEd, the magtape handler can't write out the size of the file because the file size is lost to the handler. It writes out a zero in its place. The file sequence number field will be correct.

b. The only exception to the above rule is when the user wishes to open the tape as file structured and write data blocks that are not the standard 512 (decimal) byte size that RT-11 magtape .WRITx commands use. The magtape handler keeps track of the number of blocks written and the end-of-file 1 label are correct as long as no commands other than the .SPFUN write command are used. Otherwise, the block size will be lost.

c. It is recommended that the user issue .SPFUN commands to a magtape file only for the case described in b. above.

2. Exception Reporting - Those .SPFUN's that are accepted by the hardware handler report end of file and hard error conditions through byte 52 in the system communication area. Additionally, they use the argument normally used for a block number as a pointer to a four-word error and status block 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 (carry bit not set). The block is defined as follows:

   Words 1 and 2 are qualifying information.

   Words 3 and 4 are reserved, and must be set to 0.

   a. Qualifying information returned for the end of file condition is as follows:

	Code (Octal)	Condition
Word 1:	1	Tape at end of file only (tape mark detected)
	2	Tape at end of tape only (no tape mark detected)
	3	Tape at end of tape and end of file (tape mark detected)
	4	Tape at beginning of tape (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.

End of tape, tape mark and beginning of tape are returned as an end of file by the hardware handler.

b. Qualifying information returned for the hard error condition is as follows:

	Code (Octal)	Condition
Word 1:	0	No additional information
	1	Tape drive not available
	2	Tape position lost. When this error occurs, the tape should be rewound or backspaced to a known position.
	3	Nonexistent memory accessed
	4	Tape write-locked.
	5	Last block read had more information. The MM handler will return the number of words not read in the second word.
	6	Short block was read (the differences between the number of bytes (not words) requested and the number of bytes read is returned in the second word).

c. The hardware handler issues a hard error if it receives any request other than .LOOKUP (non-file-structured), .CLOSE, or any .SPFUN request not defined for the hardware handler.

d. When running under the XM monitor the blk area for error reporting must be mapped at all times.

3. Read/Write Physical Blocks of Any Size - The hardwarehandler reads and writes blocks of any size. Requests for reading and writing a variable number of words are implemented with two .SPFUN codes.

   a. The .SPFUN request to read a variable number of words in a block has the following form:

```
.SPFUN area,chan,#370,buf,wcnt,blk[,crtn]
```

where: 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 is to be entered after the request is executed.

This request returns the following errors. Additional qualifying information for these errors is returned in the first two words of the blk argument block.

Byte 52 error	Qualifying information
EOF (end of file) Value=0	Tape is at end of file only (tape mark detected) if bit 0 is set. Tape is at end of tape only (no tape mark detected) if bit 1 is set. Tape is at end of tape and end of file (tape mark detected) if bits 0,1 are set.
Hard Error (Value=1)	No additional information (Code=0)
	Tape drive is not available (Code=1)
	Tape position lost (Code=2)
	Nonexistent memory accessed (Code=3)
	Short block was read. The difference between the number of words requested and the number of words read is returned in the second word of blk (Code=6).
	The last block read had more information. For the TJU16 the number of words not read is returned in the second word of blk (Code=5).

b. The .SPFUN request to write a variable number of words to a block has the following form:

.SPFUN area,chan,#371,buf,wcnt,blk[,crtn]

where: 371 is the function code (decimal) for a write operation.

This request returns the following errors. Additional qualifying information for these errors is returned in the first two words of the blk argument block.

Byte 52 Error	Qualifying Information
EOF (end of file) (Value=0)	Tape is at end of tape only if bit 1 is set.
Hard error (Value=1)	No additional information (Code=0) Tape drive not available (Code=1) Tape position lost (Code=2) Nonexistent memory accessed (Code=3) Tape is write locked (Code=4)

NOTE

The TJU16 tape drive can return a hard error if a write request with a word count less than 7 is attempted.

4. Space Forward/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. There are two spacing requests, which have the following forms:

   a. Space forward by block

      .SPFUN area,chan,#376,,wcnt,blk[,crtn]

      where: 376 is the function code for forward space operation.

      wcnt is the number of blocks to space past (cannot exceed 65,534).

      crtn is a completion routine to be entered when the operation is complete.

   This request returns the following errors. Additional qualifying information for these errors is returned in the first two words of the blk argument block.

Byte 52 error	Qualifying information
EOF (end of file)	Tape is at end of file only (tape mark detected) if bit 0 is set Tape is at end of tape only (no tape mark detected) if bit 1 is set Tape is at end of tape and end of file (tape mark detected) if bits 0,1 are set

   The second word in blk contains the number of blocks requested to be spaced (wcnt) minus the number of blocks spaced if a tape mark is detected. Otherwise its value is not defined.

Hard error	No additional information (Code=0) Tape drive not available (Code=1) Tape position lost (Code=2)

NOTE

Due to hardware restrictions it is recommended that no forward space commands be issued if the reel is positioned past the end of tape marker.

   b. Space backward by block:

      .SPFUN area,chan,#375,,wcnt,blk[,crtn]

      where: 375 is the function code for a backspace operation.

This request returns the following errors and additional qualifying information is returned in the first two words of the blk argument block.

Byte 52 error	Qualifying information
EOF (end of file)	Tape is at end of file (tape mark detected) if bit 0 is set Tape is at end of tape (no tape mark detected) if bit 1 is set Tape is at end of tape and end of file (tape mark detected) if bit 0,1 are set Tape is at beginning of tape (no tape mark detected) if bit 2 is set

The second word in blk contains the number of blocks requested to be spaced (wcnt) minus the number of blocks actually spaced (including the tape mark) if a tape mark is detected. Otherwise, its value is not defined.

Hard error	No additional information (Code=0) Tape drive not available (Code=1) Tape position lost (Code=2)

5. Rewind - The handler accepts a rewind command, and rewinds the tape drive to the beginning of tape. The handler cannot accept other requests until the rewind operation is complete, but other handlers can be active during tape rewind. The rewind request has the following format:

```
.SPFUN area,chan,#373,,,blk[,crtn]
```

where: 373 is the function code for the rewind operation.

crtn is a completion routine to be entered when the operation is complete.

This request returns the following error, and additional qualifying information is returned in the blk argument block.

Byte 52 error	Qualifying information
Hard error	No additional information (Code=0) Tape drive not available (Code=1)

6. Rewind and Go Off Line - This request is the same as rewind except that it takes the tape drive off-line, and then rewinds to the beginning of tape. 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]
```

where: 372 is the function code for the rewind and go off-line operation.

crtn

This request returns the same error codes and qualifying information as the rewind request.

7. Write With Extended Gap - This request allows writing on tapes with bad spots. This request is identical to the write request except that the function code for write with extended gap operation is 374.

   The errors for this request are identical to those for the write request.

8. Write 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]

   where: 377 is the function code for the write tape mark operation.

   This request returns the following errors: Additional qualifying information is returned in the first two words of the blk argument block.

Byte 52 error	Qualifying information
EOF (end of file)	End of tape is detected if bit 1 is set.
Hard error	No additional information (Code=0) Tape drive not available (Code=1) Tape position lost (Code=2) Tape is write locked (Code=4)

9. Error Recovery Algorithm - Any errors detected during spacing operations cause the recovery attempt to be aborted and a hard (position) error is reported.

   a. Read Error Recovery - The hardware handler performs the following algorithm if a read parity error is detected.

      1. Backspaces over the block and rereads. When unsuccessful it is repeated until five read commands have failed.

      2. Backspaces five blocks, spaces forward four blocks, then reads the record.

      3. This entire sequence (steps 1 and 2) is repeated eight times or until the block is read successfully.

   b. Write Error Recovery - The hardware handler performs the following algorithm upon detection of a read after write 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 successful, the block could be marginal and cause problems at a later time.

      3. If the read after write still fails, the entire sequence (steps 1 and 2) are repeated. When 25 feet of erased tape have been written, a hard error is given.

10. Non-File-Structured .LOOKUP Request - The hardware handler accepts a non-file-structured .LOOKUP request. This function 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 performed. This request has the following form:

```
.LOOKUP area,chan,dblk,seqnum
```

where: seqnum is an argument that specifies whether the tape is to be rewound or not. When this argument is 0, the tape is rewound. When this argument is -1, the tape is not rewound.

This request returns the following 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	Illegal argument detected. The file name was not 0 or the seqnum had an argument that was not 0 or -1.

11. .CLOSE Request - The hardware handler accepts the .CLOSE request and causes the handler to mark the drive as available. This request has the following form:

```
.CLOSE chan
```

12. SET Commands - The hardware handler accepts SET commands to set the track number, density and parity of the tape drive. These commands are fully described in Chapter 4 of the <u>RT-11 System User's Guide</u>.

13. Non-File-Structured .WRITx Request - The hardware handler accepts .WRITx requests that write a variable number of words to a block on tape. The block number field is ignored. This request has the following form:

```
.WRITx area,chan,buf,wcnt[,,crtn]
```

This request returns the following errors. Note that no additional qualifying information is available.

Byte 52 error	Meaning
EOF (end of file) (Value=0)	The end of tape marker has been sensed.
Hard error (Value=1)	This can mean any of the error conditions listed for the file-structured write request.

14. Non-File-Structured .READx Request - This request reads a variable number of words from a block on tape. It ignores the end of tape marker and only reports end of file when a tape mark is read. The block number field is ignored. The request has the following form:

```
.READx area,chan,buf,wcnt[,,crtn]
```

This request returns the following errors. Note that there is no additional qualifying information available.

Byte 52 error	Meaning
EOF (end of file) (Value=0)	Only reported if a tape mark is read. The end of tape marker will not cause end of file.
Hard error (Value=1)	This can mean any of the error conditions listed for the file-structured read request.

Writing Tapes On Other PDP-11 Operating Systems To Be Read By RT-11

RT-11 can read files written on other computer systems that support the DIGITAL standard (ANSI) for labels. Below are a few examples of how to write ANSI tapes on some common DIGITAL PDP11 operating systems. Keep in mind that there are other factors involved besides just the label and format compatibility. These include density, parity and number of tracks written on the tape.

Writing Tapes on RSTS/E

RSTS/E supports two types of magtape formats, DOS-11 and ANSI. In the following examples, dd represents the magtape handler name, either MM or MT. In order to ensure that an ANSI file structure is written, be sure to issue the following command:

ASSIGN ddn:.ANSI	(Allocates the device to the job and ensures that an ANSI file structure is used)
RUN $PIP ddn:/ZE/VID:xxxxxx	(PIP is used to initialize the tape; xxxxxx is the volume ID)
Really zero ddn:? Yes	(PIP prompts before initializing the tape)
PIP ddn:=FARQUA.MAC,VEG.TEC	(PIP is used to copy files to the tape
DEASSIGN ddn:	(Deallocates the device)

Writing Tapes on RSX-11/M

RSX-11/M needs the following commands to access a magtape.

ALL ddn:	(Allocates a drive)
INIT ddn:RT11	(Initializes the tape and gives the name "RT11" as the volume identifier)
MOU ddn:RT11	(Mounts the tape volume)
PIP ddn:=[13,10]F11PRE.MAC,ALLOC.MAC	(Copies files to the tape)
DMO ddn:RT11	(Dismounts the tape volume)
DEA ddn:	(Deassigns the drive)

Writing Tapes on RSX-11/D and IAS

INIT ddn:RT11	(Initializes the tape and gives the name "RT11" as the volume identifier)
MOU ddn:RT11	(Mounts a tape volume)

```
(For RSX-11/D use the PIP program to write files to the tape)
(For IAS use the COPY command)
DMO ddn:RT11      (Dismounts the tape volume)
```

The above examples are intended only as examples. For more complete information on the above systems consult the appropriate documentation.

The contents of files written under the RSX-11 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 imbedded in the text) whereas the other systems just mentioned use a different type of character storage. The user is urged to pay special attention to the contents of the files he wishes to transfer.

When writing files to be read under RT-11, the only block size the RT-11 PIP program reads is 512(decimal) characters/block. However, the RT-11 DIR program produces a directory for any compatible tape.

1.4.8.2 **Cassette Tape Handler (CT)** - The CT handler can operate in two 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 used when accessing the device through any of the RT-11 system programs. In software mode, the handler automatically attends to file headers and uses a fixed record length of 64 words to transfer data.

Hardware mode allows the user to read or write any format desired, using any record size. In this mode, the word count is taken as the physical record size.

When the handlers are initially loaded by either the .FETCH programmed request or the LOAD command, only software functions are permitted. To switch from software to hardware mode, either a rewind or a non-file-structured .LOOKUP must be performed. (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:

.ENTER	- Open new file for output
.LOOKUP	- Open existing file for input and/or output
.DELETE	- Delete an existing file on the specified device.
.CLOSE	- Close a file that was opened with .ENTER or .LOOKUP
.READ/.WRITE	- Perform data transfer requests

In .ENTER, .LOOKUP, and .DELETE an optional file count parameter can be specified. Its meaning is as follows:

Count Argument	Meaning
=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 switch 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.

As an example, consider:

```
        .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 end-of-tape occurs in that space, the search is stopped, and the tape is positioned either at the end of tape (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 the end of tape is encountered first, an error is generated.

As another example:

```
    .LOOKUP #AREA,#0,#PTR,#-5
```

This performs a rewind, and then uses a file count of five in the same way the previous example does.

Handler Functions - The cassette handler performs the following functions:

1. .LOOKUP Request

   If the file name (or the first word of the file name) is null, 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. .LOOKUP uses the optional file count as illustrated above. Only software functions are allowed.

2. .DELETE Request

   .DELETE eliminates a file of the designated name from the device. .DELETE also uses the file count argument, and can thus do a delete of a numbered file as well as a delete by name. When a file is deleted, an unused space is created there. However, it is not possible to reclaim that space, as it is when the device is random access. The unused spot remains until the volume is re-initialized and rewritten. If a file name is not present, a non-file-structured .DELETE is performed and the tape is zeroed.

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 enter a file by name or by number. If enter by name is done, the handler deletes any files of the same name.

If enter by number is done, 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 .ENTERs, as it is possible to enter a file in the middle of existing files and thus destroy any 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 sees while passing down the tape. If an .ENTER is done with a count greater than 0, no rewind is performed before the file is entered. 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 to the tape's file structure, they should be used with care on a file-structured tape.

4. .CLOSE Requests

   .CLOSE terminates operations to a file on cassette and resets the handler to allow more .LOOKUPs, .ENTERs, or .DELETEs. If a .CLOSE is not performed on an entered file, the end of tape 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.

5. .READ/.WRITE Requests

   .READ and .WRITE requests are unique in that they can be done either in hardware or software mode. In software mode (file opened with .LOOKUP or .ENTER), records are written in a fixed size (64 words). The word count specified in the operation is translated to the correct number of records. On a .READ, the user buffer is filled with zeroes if the word count exceeds the amount of data available.

   Following is a discussion of how the various parameters for .READ/.WRITE are used.

   a. Block Number

      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.

b. Word Count

If the word count is 0, the following conditions are possible:

If the block number is non-zero, the operation is actually 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.

The user can then do 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.

Following is a description of the allowed hardware mode functions for the handler, as well as examples of how to call them. In general, special functions are called by using the .SPFUN request; examples of usage follow each function. The special functions require a channel number as an argument. The channel must initially be opened with a non-file-structured .LOOKUP, which places the handler in hardware mode.

The general form of the .SPFUN request is:

```
.SPFUN  area,chan,func,buf,wcnt,blk,crtn
```

where:

func is the function code to be performed.

The request format is:

```
R0 area:  32 chan

          blk
          buf
          wcnt
          func 377
          crtn
```

Cassette Special Functions

1. Rewind (Code = 373) - This request rewinds the tape to 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 done. Unless a completion routine is specified, control does not return to the user until the rewind completes. This request has the following form:

```
.SPFUN   area,#0,#373,#0,#0,#0,crtn
```

where: crtn is a completion routine to be entered when the operation is complete. The other arguments are not required.

2. Last File (Code = 377) - This request rewinds the cassette and positions it immediately before the sentinel file (logical end-of-tape). The request form is the same as for rewind except that code 377 is used.

   ```
   .SPFUN  area,#0,#377,#0,#0,#0[,crtn]
   ```

3. Last Block (Code = 376) - This request rewinds one record.

   ```
   .SPFUN  area,#0,#376,#0,#0,#0[,crtn]
   ```

4. Next File (Code = 375) - This request spaces the cassette forward to the next file.

   ```
   .SPFUN  area,#0,#375,#0,#0,#0[,crtn]
   ```

5. Next Block (Code = 374) - This request spaces the cassette forward by one record.

   ```
   .SPFUN  area,#0,#374,#0,#0,#0[,crtn]
   ```

6. Write File Gap (Code = 372) - This request terminates a file written by the user program when in hardware mode.

   Sample Macro Call:

   ```
   .SPFUN  area,#0,#372,#0,#0,#0
   ```

   This writes a file gap synchronously, while:

   ```
   .SPFUN  area,#0,#372,#0,#0,#0,#1
   ```

   or

   ```
   .SPFUN  area,#0,#372,#0,#0,#0,crtn
   ```

   performs asynchronous write file gap operations.

Cassette End-of-File Detection - Since cassette is a sequential device, the handler for this device cannot know in advance 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 file. User programs can use the following procedures to determine if the handler has encountered end of file in either software or hardware mode.

In software mode, if end of file is encountered during a read and some data is read the cassette handler will zero fill the rest of the buffer and return to the caller. The next read attempted on that channel returns with the carry bit set and with the error byte (absolute location 52) set to indicate an attempt to read past end-of-file.

In hardware mode, the cassette handler does not report end of file as it does in software mode. The best way that user 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.

Example:

```
        TACS=177500                         ;TA11 CONTROL AND STATUS REGISTER
        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 PL - NO
        BIT     #TAEOF,@#TACS               ;YES - WAS IT END OF FILE?
        BNE     EOF                         ;IF NE - YES
            .
            .
            .
EOF:                                        ;CASSETTE END OF FILE ENCOUNTERED
            .
            .
```

If desired, both the EOF and EOT bits could be checked:

```
        BIT     #MTSEOF+MTSEOT,@#MTS ;MT EOF OR EOT?
```

or

```
        BIT     #TAEOF+TAEOT,@#TACS ;CT EOF OR EOT?
```

1.4.8.3 **Diskette Handlers (DX,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 normally 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,code,buf,wcnt,blk,crtn
```

where:

code — is the function to be performed:

- 377 Read physical sector
- 376 Write physical sector
- 375 Write physical sector with deleted data mark
- 374 unused
- 373 (DY only) determine device size, in 256-word blocks, of a particular volume

buf — for functions 377, 376, 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.

for function 373:
buf 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 (decimal) is returned. For double density diskettes, 988 (decimal) is returned.)

wcnt — for functions 377, 376, 375:
is the absolute track number, 0 through 76, to be read or written.

for function 373:
wcnt is unused and should be set to 1.

blk — for functions 377, 376, 375:
is the absolute sector number, 0 through 26, to be read or written.

for function 373:
blk is unused and should be set to C.

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.

Sample Macro Call:

```
.SPFUN  #RDLIST,#1,#377,#BUFF,#0,#7,#0
                                ;PERFORM A
                                ;SYNCHRONOUS SECTOR READ
                                ;FROM TRACK 0, SECTOR 7
                                ;INTO THE 65-WORD AREA BUFF
```

Each DX and DY handler can support two controllers, and each controller supports two drives. For example, if the RX01 handler is SYSGENed 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. There is no overlapped I/O to the handler.

1.4.8.4 **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 1-3) or as column images controlled by the SET command. In ASCII mode (SET CR NOIMAGE), invalid punch combinations are decoded as the error character 134(octal)--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 end-of-file card is encountered (12-11-0-1-6-7-8-9 punch in column 1; the rest of the card is arbitrary). On end-of-file, the buffer is filled with zeroes and the request terminates successfully; the next input request from the card reader gives an end-of-file 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. End-of-file 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/NO

HANG option. No special action is required to use the card reader handler with the CM 11 mark sense card reader. The program should be aware of the fact that mark sense cards can contain less 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.

Table 1-3
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	107	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)

(continued on next page)

Table 1-3 (Cont.)
DEC 026/DEC 029 Card Code Conversions

Zone	Digit	Octal	Character	Name
(DEC 029)	6	075	=	equal sign
(DEC 026)		047	'	single quote
(DEC 029)	7	042	"	double quote
(DEC 026)		134	\	backslash
9				
	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

(continued on next page)

Table 1-3 (Cont.)
DEC 026/DEC 029 Card Code Conversions

Zone	Digit	Octal	Character	Name
11-8				
	none	121	Q	upper-case Q
(DEC 029)	2	135	]	close square bracket
(DEC 026)		072	:	colon
	3	044	$	currency symbol
	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

(continued on next page)

Table 1-3 (Cont.)
DEC 026/DEC 029 Card Code Conversions

Zone	Digit	Octal	Character	Name
0-9-8				
	5	005	CTRL/E	ENQ
	6	006	CTRL/F	ACK
	7	007	CTRL/G	BEL
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

1.4.8.5 **High-Speed Paper Tape Reader/Punch** (PC) - RT-11 provides support of the PR11 High Speed Reader and the PC11 High Speed Reader/Punch through the PC handler. The PC handler distributed with the system supports both the paper tape reader and punch. A handler supporting only the paper tape reader can be created during SYSGEN. The PC handler does not print an ^ on the terminal when it is entered for input the first time, as did the PR handler for earlier releases 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 being done 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 to disk or DECtape, and then perform the operation on the transferred 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 end-of-file bit in byte 52 is set and the C bit is set on return from the I/O completion.

1.4.8.6 **Console Terminal Handler (TT)** - The console terminal can be used as a peripheral device by using the TT handler. Note that:

1. An ^ 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, struck while output is directed to TT, causes an entire output buffer (all characters currently queued) to be ignored.

4. A single CTRL/C struck 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. The user 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 does a .TTYIN, typed characters are passed unpredictably to the .TTYIN and TT. Therefore, this practice should be avoided.

8. If a job sends data to TT for output and then does a .TTYOUT or a .PRINT, 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.

9. The TT handler does not interface to terminals other than the assigned console terminal in a multi-terminal system.

1.4.8.7 **RK06/07 Disk Handler (DM)** - The RT-11 RK06/07 handler has some features that are not standard for most RT-11 handlers. Among these non-standard 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.)

These features are discussed further in the following sections.

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 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 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. The advantage of using bad block replacement is that the entire disk appears to be good. If .BAD files are used instead, the disk becomes fragmented around the bad blocks.

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.

2. .SPFUN Requests - The RK06/07 handler accepts the .SPFUN request with the following function codes:

   377 - for a read operation
   376 - for a write operation
   374 - for initializing the bad block replacement table in the handler.
   373 - for determining the size, in 256-word blocks, of a particular volume.

   The format of the .SPFUN request is the same as explained in Chapter 2 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	If the I/O operation is successful
100200	If a bad block is detected (BSE error)
100001	If an ECC error is corrected
100002	If an error recovered on retry
100004	If an error recovered through an offset retry
100010	If an error recovered after recalibration
1774xx	If an error did not recover

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.

1.4.8.8 Null Handler (NL) - The null handler can accept all read/write requests. On output operations this handler acts as a data sink. When NL is called, it returns immediately to the monitor indicating that the output is complete. The NL handler returns no errors and causes no interrupts. On input operations NL returns an end-of-file indication for all requests and no data is transferred. Hence, the contents of the input buffer are unchanged.

1.4.8.9 RL01 Disk Handler (DL) - The RL01 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 .SPFUN requests are as follows:

377 -	for a read operation
376 -	for a write operation
374 -	for initializing the bad block replacement table in the handler
373 -	for determining 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.

See the description of the .SPFUN programmed request in Chapter 2 for details on the special functions.

Bad block replacement for the RL01 is similar to the bad block support for the RK06 and RK07. However, the RL01 device generates 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 disk contains a table of the bad sectors that were discovered during manufacture of the disk. The ten blocks preceding this table (the last ten blocks in the second-to-last track) are set aside for bad block replacements. The maximum number of bad blocks, ten, is defined in the handler.

As with the RK06 and RK07, the user determines 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.

If the /REPLACE option is specified during initialization of an RL01 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 a zero word. The entries in the table are in order by ascending bad block number. A sample table is as follows:

Bad block	12	Word 0
Its replacement	10210	
	37	Word 2
	10211	
	553	Word 4
	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 is the number of RL01 units to be supported. The value defaults to two 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 a .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.

1.5 MULTI-TERMINAL SUPPORT

The multi-terminal device handler supports from one to sixteen terminals. It is a SYSGEN option for FB and XM monitors that is integrated into the resident monitor (RMON) and console terminal service.

The multi-terminal service provides eight programmed requests as follows (see Chapter 2 for additional details):

Sub-Code	Request	Operation
0	.MTSET	Set terminal characteristics
1	.MTGET	Get terminal characteristics
2	.MTIN	Input characters from terminal
3	.MTOUT	Output characters to terminal

Sub-Code	Request	Operation
4	.MTRCTO	Reset CTRL/O flag
5	.MTATCH	Attach a terminal
6	.MTDTCH	Detach a terminal
7	.MTPRN	Print a line

Errors are returned in the error byte, location 52, as follows:

Error Codes	Meanings
0	No character in buffer (MTIN). No room in buffer (MTOUT).
1	Illegal unit number. The job did not attach it.
2	Non-existent unit number.
3	Illegal request - sub-code out of range.
4	Attempt to attach or detach a unit that is already attached to another job.
5	Buffer or status block is outside legal addressing range (XM monitor only).

The number and types of interfaces must be declared at SYSGEN time, then logical unit numbers (lun) are assigned to identify the terminals. Lun's are assigned in the following order:

1. hardware console interface (a local DL11)
2. other local mode DL11's
3. remote DL11 interfaces
4. local DZ11 lines
5. remote DZ11 lines

A unit control block, which associates a lun with a specific interface, is set up for each terminal. Terminals are referenced by the logical unit numbers. For example, logical unit number 0 is the default console lun and is assigned to the hardware console interface. The .TTYIN, .TTYOUT, .PRINT, .CSIGEN, .CSISPC, .GTLIN requests, and all TT references use the console; no TT support is provided for terminals other than the console. Hence, an .MTIN or .MTOUT executed with 0 as the logical unit number is directed to the console terminal. However, the terminal that the system uses as the console can be changed by the SET command as follows (provided that the terminal is a local DL11):

```
SET TT CONSOL=n
```

where:

n is a decimal value from 0 to 15 that indicates the logical unit number of the terminal to be used as the new console.

For example, the following command assigns terminal number 3, which is a local DL11, to the system hardware console interface:

```
SET TT CONSOL=3
```

After this command is issued, .TTYIN, .TTYOUT, .PRINT, and any other requests directed to the console terminal will use terminal number 3.

The foreground and background jobs can either share a single console or they can have separate consoles. If a console is shared, only one job can attach it. Only the owner of the shared console can issue multi-terminal programmed requests to the terminal, but both jobs can issue .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT requests.

All other terminals must be attached by the job before they can be referenced and used. When the terminal becomes attached, it is dedicated to the job that issued the attach request except when the console must be shared by foreground and background jobs. The foreground job can have a separate console with a different lun assigned to it. This lun will be the default value for the .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT programmed requests. The assigning of the separate console is performed at load time by the FRUN option /T:lun. The separate console is not the primary system console and can only be considered an auxiliary console since KMON cannot communicate with it. This auxiliary foreground console must also be a local DL11 terminal interface and cannot be changed by the SET TT CONSOL command.

When a terminal is attached to a job, it remains attached until it is detached by a .MDTCH programmed request (see Chapter 2 for details), or until the job exits or is aborted. If the terminal is detached through a programmed request, the output in process at the terminal is allowed to finish before the terminal is detached. If the terminal is detached by aborting the job, the output is terminated and the terminal is detached immediately.

When a terminal is attached to a job, it has the following default characteristics:

80.	character column width
CRLF$	option enabled (generates LF after RET)
PAGE$	option enabled (XON/XOFF enabled)

These defaults can be changed by the .MTSET request.

An asynchronous terminal status (ATS) option is available and can be selected at SYSGEN time. This option provides the job with updated status of the terminal and modem. When the terminal is attached, the job can supply a status word that is updated as changes in the terminal status occur. The status bits and their meanings are as follows:

AS.CTC	100000	bit 15	Double CTRL/C struck
AS.INP	40000	bit 14	Input is available
AS.OUT	20000	bit 13	Output buffer empty
AS.CAR	200	bit 7	Carrier present (remote only)

The AS.CTC bit is set if a double CTRL/C is struck on any terminal except the job's console terminal. If a double CTRL/C is struck on the job's console terminal, the job is aborted unless an .SCCA request has been issued. In this case, bit 15 of the terminal status word is set. This bit must be reset by the job before further processing.

The AS.INP bit is set if input is available (a line of characters in normal mode or a single character in special mode). The bit is cleared when the characters are read.

The AS.OUT bit is set when the output ring buffer is empty (when the last character is printed). It is cleared when characters remain to be printed.

The AS.CAR bit is set when a remote line is answered. It is cleared when a remote line hangs up or drops a carrier.

All of the bits discussed in the previous section indicate significant events have occurred when they are set. These bits are set and cleared by the multi-terminal service, except AS.CTC, which must be cleared by the program when tested.

1.6 ERROR LOGGING

The error logging process keeps a statistical record of all I/O operations on devices that are supported by this feature. In addition to the statistics, the error logging process also detects and stores any errors that occur during the I/O operations. The following statistics for each supported device are recorded:

1. number of read successes
2. number of write successes
3. number of hard errors (unrecoverable errors)
4. number of soft errors (recoverable errors)

The following statistics for memory and cache are recorded:

1. number of memory parity errors
2. number of cache memory errors

In addition to the statistics listed above, the following information is retained if an error occurs in memory:

1. error sequence number
2. PC
3. PS
4. memory parity registers
5. cache error registers

The following information is retained if an error occurs on a supported peripheral device:

1. error sequence number
2. unit number
3. device ID (from $STAT)
4. queue element block number

5. queue element buffer address
6. queue element word count
7. device hardware registers
8. total retry count
9. retry countdown

1.6.1 The Error Logging Subsystem

The error logging process is implemented through an error logging subsystem consisting of four programs written in MACRO and FORTRAN (see Figure 1-4 and Table 1-4).

Table 1-4
Error Logging Subsystem Components

Program	Language	Function
Error Log Handler (EL)	MACRO-11	Reads and stores system errors and successful I/O operations for all supported devices.
Error Log Utility (ERRUTL)	MACRO-11	Creates a disk file (ERRTMP.SYS), writes out the data collected by the EL handler to the file ERRTMP.SYS, and queries for number of errors in EL.
Error log file Formatter (PSE)	MACRO-11	Formats the file produced by ERRUTL into a standard error log file named ERROR.DAT.
Error Summary/ Report Generator (SYE)	FORTRAN IV	Analyzes and writes out the contents of the standard error log file to a hard-copy or visual display device.

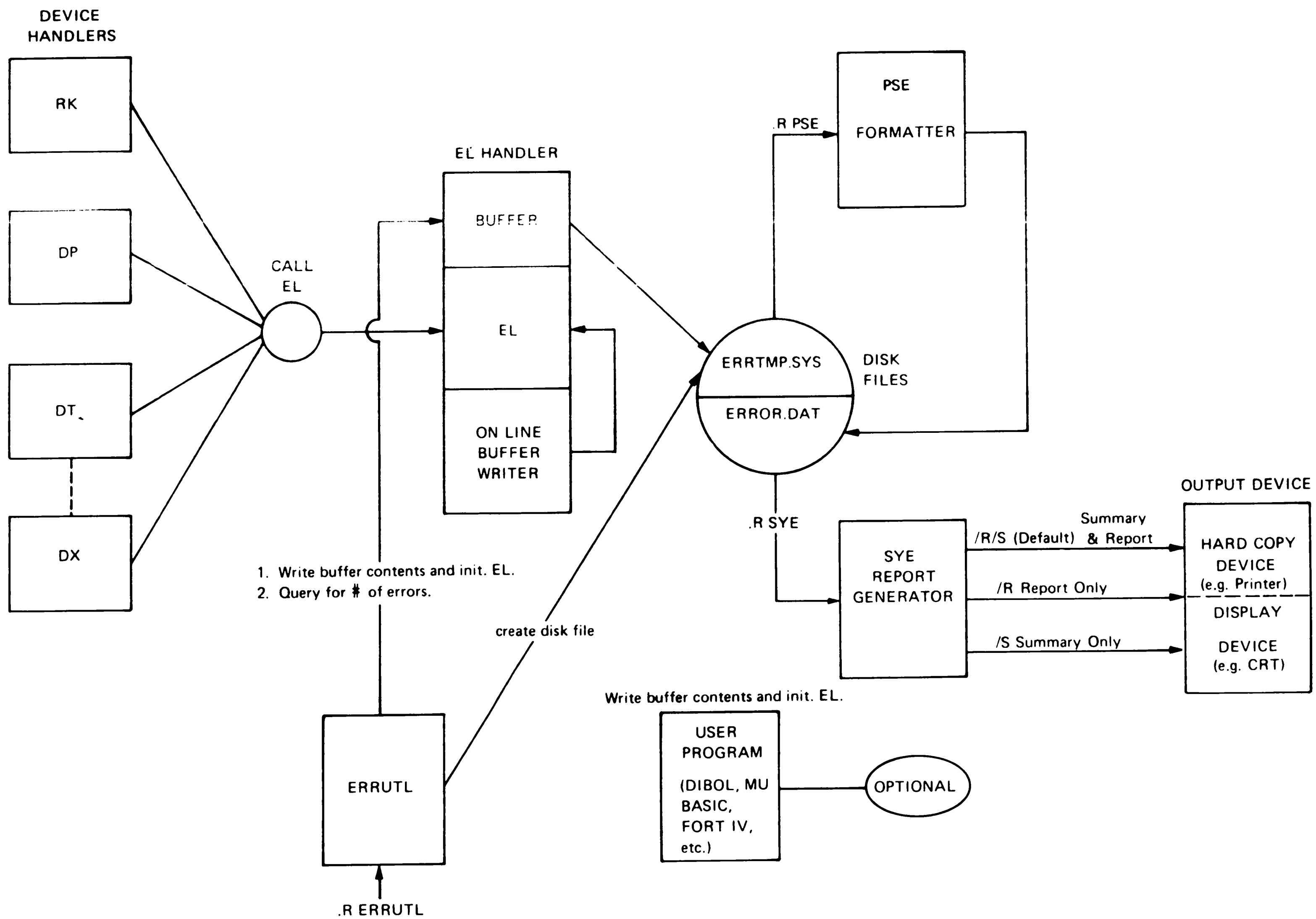


Figure 1-4 Error Logging Subsystem Functional Block Diagram

1.6.1.1 **The Error Log (EL) Handler** - The RT-11 EL handler is a MACRO-11 program that reads and stores errors and statistics and I/O operations. It consists of the following parts:

1. Information and pointer area
2. Buffer initializer
3. On-line memory-to-file routine
4. Statistics and error collector
5. Statistics buffer
6. Error log buffer

The functions for the various parts of the EL handler are discussed in the following section:

1. The information and pointer area consists of the following:

   a. An error buffer overflow counter containing the number of free words in the error log buffer.

   b. An offset pointer containing the byte offset to the statistics buffer from the EL load address.

   c. An offset pointer containing the byte offset to the error log buffer from the EL load address.

   d. The sequence number of the next error to be logged. If the value is equal to 1, it indicates that no error has been logged.

2. Buffer Initializer - This section of the EL handler is called to initialize the error log buffer. The error log buffer can be initialized in two ways:

   a. As a ring buffer to save the newest data.

   b. As a sequential buffer to save the oldest data.

3. On-Line Memory-To-File Routine - This part of the EL handler allows the user programs or system programs (such as a multi-user language system) to write the statistics buffer's and error log buffer's contents to the disk-resident error log file (ERRTMP.SYS). The program, however, must provide a channel and queue element to accomplish the write operation. In addition to this program-controlled method of writing the buffers' contents to the ERRTMP.SYS file, facilities are provided for accomplishing the same thing manually through a system utility program (ERRUTL).

4. Statistics and Error Collection - This section of the EL handler logs the read/write statistics and detects and stores the error information. The information is retained in two separate buffers until they are written to the disk-resident error log file (ERRTMP.SYS).

5. The statistics buffer stores information on I/O operations for all supported devices since the last time that the buffer was initialized. The information contained in this buffer is as follows:

   a. The number of successful read/write operations.

b. The number of hard and soft I/O errors. A soft error is defined as one that recovered or corrected itself. A hard error is defined as one that did not recover or correct itself and was reported back to the program.

c. The number of cache and memory parity errors.

6. The error buffer stores information on each hard or soft device error and parity or cache errors. The information contained in this buffer for a hard or soft device error is as follows:

   a. The error sequence number and error type

   b. The unit and device identification

   c. The device's block address

   d. The memory buffer address

   e. The word count

   f. The retry count

   g. The number and content of all pertinent hardware registers

   The information contained in the error buffer for a cache or memory parity error is as follows:

   a. The error sequence number and error type

   b. The PC and PS

   c. The memory parity registers or cache error registers

1.6.1.2 **The Error Utility Program (ERRUTL)** - ERRUTL is a utility program that performs the following operations:

1. Creates the system's error log file, ERRTMP.SYS.

2. Writes the error buffer and statistics buffer to the ERRTMP.SYS file.

3. Allows the operator to query the number of errors in the error buffer.

4. Initializes the EL handler after writing the buffer contents or after creating the ERRTMP.SYS file upon an operator's request.

Table 1-5 summarizes the commands that ERRUTL accepts.

1.6.1.3 **Data Format Converter (PSE)** - PSE is a system program that performs the following operations:

1. Determines if the ERROR.DAT file exists on the system disk.

2. Creates the ERROR.DAT file if it does not already exist.

3. Reads the ERRTMP.SYS file and converts the RT-11 specific records to equivalent records in a DIGITAL standard error logging format in the ERROR.DAT file.

1.6.1.4 **Report Generator (SYE)** - SYE is a system program that performs the following operations:

1. Formats the ERROR.DAT file data into an error report, or summary, or both.

2. Writes out the formatted data to a display, hard-copy, or other device.

1.6.2 Using the Error Logging Subsystem

The error logging subsystem is useful in providing a history of system performance that can be used to determine if specific devices are becoming unreliable. However, the use of this subsystem does present some restrictions to the overall RT-11 operating system. Among these restrictions is the additional overhead on all I/O transfers whether an error occurred or not. The additional overhead on I/O transfers will be noticed only on time-dependent processes. In addition to the increased amount of time, some memory space must be permanently given up due to the increased size of the monitor and the presence of the EL handler. Presently, the EL handler occupies a minimum of slightly less than 1K words of memory.

Error logging is not included in the distributed RT-11 monitors. A system generation must be performed to enable error logging. See the RT-11 System Generation Manual for details.

1.6.2.1 **Loading the EL Handler** - The first thing that must be done to use the error logging subsystem is to load the EL handler. The EL handler is loaded and unloaded by the standard RT-11 LOAD and UNLOAD commands.

To load the handler, type the following command in response to the monitor's prompt (.). All commands must be terminated by a carriage return.

```
.LOAD EL(RET)
```

It is desirable to have the EL handler loaded before other handlers are loaded. This practice allows other handlers to be loaded and then released, thus returning the memory space back to the system.

1.6.2.2 **Using ERRUTL** - After the EL handler is loaded, ERRUTL must be used to create the error log file (ERRTMP.SYS) on the designated device. To invoke the ERRUTL program, the user should type the following command in response to the keyboard monitor's prompt (.). All commands must be terminated by a carriage return.

```
.R ERRUTL(RET)
*
```

The asterisk indicates that ERRUTL is ready to accept a command. Commands to ERRUTL are of the form:

```
device-name:[/options]
```

where:

device-name
: is the RT-11 physical device code for the output device for the file ERRTMP.SYS. The format for the device code is:

    ddn

    where

    dd
    : is the two-character RT-11 device mnemonic.

    n
    : is the device unit number.

options
: represents one or more command options from Table 1-5.

Table 1-5
ERRUTL Options

Option	Meaning
/C[:s]	Creates ERRTMP.SYS. The argument, s, specifies the file size, in records. The default size is 20 records. One record accommodates the full contents of the EL handler's error and statistics buffers. ERRTMP.SYS need not be created more than once.
/N	Saves the most recent errors. When the buffer becomes full, the old data is replaced with the most recent errors. The default operation is to save the oldest errors. If the default is chosen, errors that occur after the buffer becomes full are lost. This default can be changed at SYSGEN time.
/Q	Queries the EL handler for the number of errors currently in the buffer.
/W	Writes the contents of the buffer to ERRTMP.SYS and empties the buffer.

The following example creates ERRTMP.SYS on device DM1:. It will contain 50 records of the most recent errors.

```
*DM1:/C:50./N (RET)
```

The user should type (CTRL C) to exit from ERRUTL.

NOTE

If ERRTMP.SYS is written by a program other than ERRUTL, the file ERRTMP.SYS must be created each time the system is bootstrapped.

Another function of the ERRUTL program is to query the EL handler for the number of errors currently stored in the error buffer. Once the amount of data stored in the EL handler is known, ERRUTL can be used to write the contents of the error and statistics buffers to ERRTMP.SYS.

The /Q option should be used to query the EL handler, as this example shows:

```
.R ERRUTL (RET)
*/Q (RET)
```

The current sequence number and the number of bytes remaining in the error buffer print on the console terminal. The error sequence numbers begin with 1 and end with the number of errors in a full buffer, plus 1. An error sequence number of 1 indicates that the buffer is empty.

The /W option is used to write the buffer contents to ERRTMP.SYS. Once ERRUTL is invoked, the format is as follows:

```
*device-name:/W
```

where:

device-name represents the device on which ERRTMP.SYS is stored.

The buffers in the EL handler are written to the specified device and re-initialized as a result of this procedure. The EL handler is returned to the state it was in when it was first loaded.

The following command sequence, for example, causes the error and statistics buffers to be written to DM0:ERRTMP.SYS.

```
*DM0:/W (RET)
```

1.6.2.3 **Converting the Error Log File to a FORTRAN Data File** - A system program (PSE) is used to convert the error log file (ERRTMP.SYS) to a FORTRAN IV-compatible ERROR.DAT permanent disk file. When the ERRTMP.SYS file is full, or when the user desires to get a listing of the data even if the file is not full, the PSE program can be used to read the ERRTMP.SYS file and convert it to FORTRAN-readable records. Then another system program (SYE, discussed in the next section) is run on the ERROR.DAT file to generate a hard-copy report or display. The ERROR.DAT file is much larger than the ERRTMP.SYS file. Thus, the ERROR.DAT file can accumulate several ERRTMP.SYS files to provide a history of processor errors.

To invoke the PSE program, the user should type the following command in response to the keyboard monitor's prompt (.). All commands must be terminated by a carriage return.

```
.R PSE (RET)
*
```

The asterisk indicates that PSE is ready to accept a command. Commands to PSE are of the form:

```
output-filespec[/option]=input-device:
```

where:

output-filespec — represents the device, file name, and file type for the FORTRAN file, in the format:

    ddn:filnam.typ[size]

where

dd	is the two-character RT-11 device mnemonic.
n	is the device unit number.
filnam	is the six-character file name.
typ	is the three-character file type.
size	is an optional argument that specifies the size in blocks of the output file. The default is 60 (decimal) blocks.

The default is DK:ERROR.DAT.

option — represents a command option from Table 1-6.

input-device — represents the input device. The default is DK:.

Table 1-6
PSE Options

Option	Meaning
/x[:size]	Causes PSE to change the size of the existing output file. If no size argument is specified, the existing output file is doubled. If a size is specified, the existing output file is increased by that number (decimal) of blocks.

When PSE is invoked, it must first determine the state of the ERROR.DAT file. If no current ERROR.DAT file exists, a new file must be created. Under this condition, the user is requested to input the number of blocks for the new file. If the ERROR.DAT file does exist, PSE examines the file to ensure that there is enough space for the records to be added.

The output file size can be changed at the program level in the following manner:

1. PSE determines if the output file size is large enough to hold the new input records.

   a. If the file size is sufficient, processing continues.

b. If the file size is insufficient, the following error message and prompts are printed out at the console:

```
?PSE-F-OUTPUT FILE TOO SMALL

MUST DELETE RECORDS FROM:  MM:DD:YY

OK TO DELETE:  Y OR N?
```

2. If N followed by a carriage return is entered, PSE prompts for further input, and no records are deleted.

3. If Y followed by a carriage return is entered, records from the specified days (month: day: year) are deleted from the file and processing continues.

The records to be deleted are displayed at the terminal, and PSE can be aborted if the operator wishes to retain the old records.

Once the file space has been examined, the formatting operation begins. As records are formatted and added to the ERROR.DAT file, they are deleted from the ERRTMP.SYS disk file. At the end of the operation ERRTMP.SYS is left in its original null state and is available to receive new data from the EL handler's buffers again. If no further error logging is required, the operator can completely delete the ERRTMP.SYS file.

The PSE program creates one error record for each memory or device error in the buffer. This record contains a header field, a register field and a program field.

In addition to the error record, PSE creates one statistics record for each unit in use during the time span encompassed by this buffer of error data. PSE also creates a statistics record for memory and cache systems. This record contains a header field and a statistics field. Each record contains an error sequence number starting with the next sequential error number after the last one in the buffer. If no error occurred for a device during the time span of the buffer, only the statistics record is generated for read/write operations.

1.6.2.4 **Generating the Error Report** - The last operation and the end objective of the error logging subsystem is to generate the error report. This function is accomplished by using a system program named SYE. SYE formats the ERROR.DAT file into an error report, an error summary, or both. After the data is formatted into the desired type of report, SYE writes out the file to a printer, visual display or other device.

NOTE

Before running the SYE program, the user must make certain that the system date and time are current by executing the DATE and TIME commands. If changes are necessary, enter the following commands in response to the monitor's prompt (.):

```
.DATE  dd-mmm-yy (RET)
.TIME  hh:mm:ss (RET)
```

To invoke the SYE program, the user should type the following command in response to the keyboard monitor's prompt (.). All commands must be terminated by a carriage return.

```
.R SYE (RET)
*
```

The asterisk indicates that SYE is ready to accept a command. Commands to SYE are of the form:

output-filespec=input-filespec[/options]

where:

output-filespec
: represents the device, file name, and file type that are destination for the error report. If no file specification is given, the default device is LP:. If a file name and file type are specified, the default device is DK:. The default file name and file type are ERROR.LST.

input-filespec
: represents the device, file name, and file type for the input file. The default is DK:ERROR.DAT.

/options
: represents the valid options for SYE. If no options are specified, SYE prints both an error report and an error summary. Table 1-7 lists the options for SYE.

Table 1-7
SYE Options

Option	Meaning
/R	Generates the error report.
/S	Generates the error summary.
default (when no option specified)	Generates both the error report and the error summary.

An error report is a listing of the error types for each supported device. The format of this report is very similar to the format of the error log file. The error summary is a tally or summation of all errors contained in the error log file. The output is selected by user-specified options included in the command string.

1.6.2.5 **Error Logging Example** - The following commented listing is a sample error log run. Following the commands are actual reports produced by SYE, including detailed descriptions of them.

RT-11SJ V03-02 ?KMON-F-Command file not found	The RT-11 system is bootstrapped.
.TIME 15:40:30	The date and time are entered.
.DATE 14-FEB-78	
.LOAD EL	The EL handler is made resident in memory.
.R ERRUTL *RK0:/C	The file ERRTMP.SYS is created on RK0: since it did not already exist. The error logger has been initialized and is ready to log errors when the user proceeds with regular system operation. If the user has not altered the application software to automatically dump the memory error buffer (see Section 1.6.3), ERRUTL must be queried to determine whether or not the buffer is full.
.R ERRUTL */Q	ERRUTL is queried for the number of errors in the memory error buffer.
SEQUENCE NUMBER=2 WORDS LEFT=243	
*RK0:/W	The memory error buffer is written to RK0:.
.R PSE *RK0:ERROR.DAT=RK0: *^C	PSE is invoked to convert the MACRO records to standard FORTRAN-IV records. (Note that FORTRAN-IV is not required to obtain error logging support).
.R SYE *LP:=/R/S ?SYE-I- 2. Pages *^C	SYE is invoked to format the ERROR.DAT file into an error report and a summary, and to output it to the line printer. SYE then prints an informational message specifying the number of pages printed.

The device error report is printed first.

```
A SYE V03-01 SYSTEM ERROR REPORT COMPILED AT 14-FEB-78 15:44:09    PAGE    1.

 ****************************************************************************
B DISK DEVICE ERROR
C LOGGED 14-FEB-78 15:41:31
D ENTRY NUMBER     1.
 ****************************************************************************

 UNIT IDENTIFICATION:
      E UNIT  PHYSICAL #            1
      F TYPE                      RX01

 SOFTWARE STATUS INFORMATION:
      G RETRIED                     8.
      H NON-RECOVERED

 DEVICE INFORMATION:
      REGISTERS:
           I RXCS                100140
           J RXDB                000000
           K RXES                000120
      ADDRESS AT ERROR:
           L BLOCK                             6.
      M TRANSFER SIZE IN BYTES:             1000
      N PHYSICAL BUFFER ADDRESS START:       640.
```

The report is interpreted as follows:

Line	Meaning
A	SYE version identification and report title; includes date and time report was generated.
B	Describes the type of error.
C	Date and time the error occurred.
D	Entry number for the particular error.
E	The unit number of the device in error.
F	The type of device in error.
G	The number of times that RT-11 retried the operation in error.
H	Indicates whether or not RT-11's error retry procedure corrected the error.
I through K	Show the device/controller registers at the time of the error. The first column lists the register mnemonics. The second column lists the contents of the registers. The register information on retry operations is not logged.
L	The logical block number at the start of the transfer.
M	The amount of data being transferred to the buffer of the program incurring the error.
N	The starting address of the buffer in the program incurring the error.

The device statistics report is printed next.

```
 ******************************************************************
A DEVICE STATISTICS
B LOGGED 14-FEB-78 15:41:31
C ENTRY NUMBER     2.
 ******************************************************************

 UNIT IDENTIFICATION:
      D UNIT  PHYSICAL #          0
      E TYPE                     RK03/RK05/RK05F

 DEVICE STATISTICS FOR THIS UNIT:
      F # SOFT ERRORS:                          0.
      G # HARD ERRORS:                          0.
      H # OF READ SUCCESSES:                   22.
      I # OF WRITE SUCCESSES:                   0.


 ******************************************************************
 DEVICE STATISTICS
 LOGGED 14-FEB-78 15:41:31
 ENTRY NUMBER     3.
 ******************************************************************

 UNIT IDENTIFICATION:
        UNIT  PHYSICAL #          1
        TYPE                     RX01

 DEVICE STATISTICS FOR THIS UNIT:
        # SOFT ERRORS:                          0.
        # HARD ERRORS:                          1.
        # OF READ SUCCESSES:                    0.
        # OF WRITE SUCCESSES:                   0.
```

The report is interpreted as follows:

Line	Meaning
A	Shows that device statistics follow.
B	Date and time the statistics were logged.
C	The entry number of the error in the error log.
D	The unit number for the device in error.
E	The type of device in error.
F	The number of soft errors that were logged for the device.
G	The number of hard errors that were logged for the device.
H	The number of successful reads that were performed without retries for the device.
I	The number of successful writes that were performed without retries for the device.

The summary report is printed last.

```
SYE V03-01 SYSTEM ERROR REPORT COMPILED AT 14-FEB-78 15:44:34    PAGE     2.

          SUMMARY REPORT

   A REPORT FILE ENVIRONMENT
           B INPUT FILE              DK :ERROR .DAT
           C OUTPUT FILE             LP :ERROR .LST
           D SWITCHES                /R/S
           E DATE OF FIRST ENTRY     14-FEB-78 15:41:31
           F DATE OF LAST ENTRY      14-FEB-78 15:41:31
   G ENTRIES PROCESSED                          3.
   H ENTRIES MISSING                            0.
   I UNKNOWN ENTRY TYPES ENCOUNTERED            0.
   J FIELD FORMAT ERRORS ENCOUNTERED            0.
   K UNKNOWN DEVICES ENCOUNTERED                0.
   L DEVICE STATISTICS PROCESSED                2.
   M MEMORY STATISTICS PROCESSED                0.
   N DEVICE ERRORS PROCESSED                    1.
   O PARITY ERRORS PROCESSED                    0.
           P -MEMORY                            0.
           Q -CACHE                             0.
           R -UNKNOWN                           0.

     SYSTEM ERROR REPORT SUMMARY

   S RK03/RK05/RK05F        UNIT #    0
   T SOFT                        0.
   U HARD                        0.
   V READ SUCCESSES             22.
   W WRITE SUCCESSES             0.

     SYSTEM ERROR REPORT SUMMARY

     RX01                   UNIT #    1
     SOFT                        0.
     HARD                        1.
     READ SUCCESSES              0.
     WRITE SUCCESSES             0.
```

The summary report is interpreted as follows:

Line	Meaning
A through F	Describe the SYE input and output files, and the date and time of the first and last entries in the input file.
G	The number of error entries formatted in the report.
H	The number of errors missed because the occurrence of another error prevented a previous one from being logged.
I	The number of unknown errors encountered by SYE. An unknown error is any entry that the current version of SYE cannot format. This situation can occur if an old version of SYE has been run.
J	The number of times that the input file encountered a data structure error (field format error). Such encounters can indicate that the wrong version of the pre-formatter PSE was used.
K	The number of entries that referred to a device not supported by SYE. Such an entry can be encountered if an application has implemented error logging on a device that SYE does not recognize.

where: wcnt =1 initializes the EL handler and configures the buffer to retain the newest errors.

=2 initializes the EL handler and configures the buffer to retain the oldest errors.

=3 returns four words of information in the buffer "buf".

The four words returned in buf for code 3 are as follows:

word 1 = the number of bytes remaining in the error buffer. If this word is equal to 0, no space remains. The buffer is full.

2 = the offset from the load point of the handler to the start of the statistics buffer. Note that a .DSTATUS returns the load address+6.

3 = the offset from the load point of the handler to the start of the error buffer.

4 = the sequence number of the next error. This value is reset to 1 when the EL handler is initialized.

1.6.3.3 **Calling the Error Logger from a Handler** - The error logger can be called from a user-written device handler. EL should be called on every successful transfer. If possible, the handler should distinguish non-recoverable errors (such as write-locked volume, unmounted volume, etc.) and not log them. The error logger should also be called on an initial error and on every retry for that error. Eventual success should be reported by a -1 in the high byte of R4; complete failure should be reported by a 0. The error logger keeps track of both soft (recoverable) and hard (non-recoverable) errors.

When the error logger is called from a handler, the call must be made from fork level. This is because the error logger is not re-entrant. Fork must be used to serialize access to the logger.

The call for the error logger is:

```
JSR    PC,@$ELPTR
```

where:

$ELPTR is a pointer to the error logger in the table of pointers constructed by the .DREND macro.

At the time of the call, the error logger requires the following information:

R2 must point to a buffer in the driver that is large enough to temporarily store all the device registers.

R3 the low byte must contain the number of device registers to log; the high byte must contain the maximum retry count.

R4 the high byte must contain the device identification code (extracted from the low byte of the device status word); the low byte must contain a success code:

- -1 for a successful transfer
- 0 for a transfer that has failed completely (the retry count is exhausted)
- n a non-zero retry count for a transfer that failed but is being tried again

R5 must point to the third word of the queue element.

After the error logger is called, R0 through R3 are restored; R4 and R5 are destroyed.

1.6.4 Building the EL Handler

The EL handler is an option that must be SYSGENed into the system along with the fork processor to have a functioning error logging capability. In addition to these software components, the system must contain a disk and at least 16K (words) of memory.

The EL handler contains the following conditional assembly parameters, which are set through a SYSGEN or contained in SYCND.MAC.

1. ERL$B = the error buffer size in 256-word blocks. The default value is 1.

2. ERL$U = the number of specific device units that can be logged. The maximum number is 35 and the default value is 10.

3. ERL$W = the buffer configuration. If set to 1, the newest errors are kept. If not, the default value of 0 is assumed, and the oldest errors are kept. The buffer configuration can be changed by the ERRUTL program when the ERRTMP.SYS file is created.

4. ERL$A = the on-line memory to file routine. If set to 1, it is included. The default value is 0 indicating that the on-line memory to file routine is not included.

The EL handler is assembled and linked as follows:

```
.R MACRO
*EL=SYCND,EL
*^C
.R LINK
*EL.SYS=EL
*^C
.
```

CHAPTER 2

PROGRAMMED REQUESTS

A number of services at the machine language level that the monitor regularly provides to system programs are also available to user-written programs. These include services for file manipulation and command interpretation, and facilities for input and output operations. User programs call these monitor services by means of "programmed requests", which are assembler macro calls written into the user program and interpreted by the monitor at program execution time.

2.0 PROGRAMMED REQUESTS WITH EARLY VERSIONS OF RT-11

Programmed requests were implemented differently in each major release of RT-11. The following sections outline the changes that were made to the programmed requests.

2.0.1 Version 1 Programmed Requests

The earliest programmed requests, such as .READ and .WRITE, were provided with the first release of RT-11. They were designed for a single user, single job environment. As such, they differ significantly from Version 2 and Version 3 programmed requests. Arguments for Version 1 requests were pushed on the stack instead of being stored in an argument list as they are now. The channel number was limited to the range 0 through 17; more channels can be allocated in later versions. Finally, no arguments could be omitted in the macro call.

Programs written for use under Version 1 assemble and execute properly under Version 3 when the ..V1.. macro call is used (see Section 2.3.1.1). The ..V1.. macro call causes all Version 1 programmed requests to expand exactly as they did in Version 1. Version 2 and Version 3 programmed requests expand as they should for Version 2 and Version 3, respectively. However, it is to the user's advantage to convert Version 1 programs so they use the current format for programmed requests. See Section 2.5 for instructions on converting Version 1 macro calls to the current format.

2.0.2 Version 2 Programmed Requests

The second major release of RT-11 brought with it some new programmed requests and a different way of handling arguments for both the new and the pre-existing requests. The new programmed requests reflected

RT-11's ability to run a foreground job as well as a background job; they provided means to suspend and resume the foreground job, and to share messages and data between the two jobs.

The major difference between Version 1 and Version 2 programmed requests is that in Version 2, arguments for the macro calls are stored in an argument list instead of on the stack. Another substantial difference is that arguments can be omitted from the macro calls in Version 2. If the area argument is omitted, the macro assumes that R0 points to a valid argument block. If any of the optional arguments are not present, the macro places a zero in the argument list for the corresponding argument. Version 1 programmed requests were modified to incorporate these changes, and the ..V1.. macro was provided so that Version 1 programs could execute properly under Version 2 without further modification.

Programs written for use under Version 2 assemble and execute properly under Version 3 when the ..V2.. macro call is used (see Section 2.3.1.1). The ..V2.. macro call causes all pre-Version 3 programmed requests to expand in Version 2 format. Version 3 programmed requests, if any, always expand in Version 3 format.

2.0.3 Version 3 (or later) Programmed Requests

The programmed requests for Version 3 provide means for user programs to access regions in extended memory and to use more than one terminal. The chief difference between Version 3 and Version 2 programmed requests is the way in which omitted arguments are handled. In Version 3, blank fields in the macro calls do not cause zeros to be entered into the argument block. In fact, the corresponding argument block entry for the missing field is left untouched.

This change can have a significant impact on user programs. If an argument block within a program is to be used many times for similar calls, a programmer can save instructions by setting up the argument block entries only once (at assembly or run time) and then leaving the corresponding fields blank in the mmacro call.

However, users should keep in mind the fact that zeros are not substituted for missing fields. Programs that make this assumption operate incorrectly and exhibit a wide range of symptoms that can be hard to diagnose. Therefore, the necessary instructions must be written to fill the argument block, if a programmed request is issued with fields left blank in the argument list.

Programmed requests from previous versions were modified to incorporate this change, and the ..V2.. macro was provided so that Version 2 programs could execute properly under Version 3 without further modification.

The macro definitions are included in the file SYSMAC.MAC; Appendix B provides a listing of SYSMAC.MAC.

The FORTRAN programmer should note that the system subroutine library gives him some of the same capability (through FORTRAN) to use the programmed requests that are available to the assembly language programmer and described in this chapter. FORTRAN users should first read this chapter and then read Chapter 4.

2.1 FORMAT OF A PROGRAMMED REQUEST

The basis of a programmed request is the EMT (emulator trap) instruction, used to communicate information to the monitor. When an EMT is executed, control is passed to the monitor, which extracts appropriate information from the EMT instruction and executes the function required. The low-order byte of the EMT instruction contains a code that is interpreted as follows:

Low-Order Byte of EMT	Meaning
377	Reserved; RT-11 ignores this EMT and returns control to the user program immediately.
376	Used internally by the RT-11 monitor; this EMT code should never be used by user programs.
375	Programmed request with several arguments: R0 must point to a list of arguments that designates the specific function.
374	Programmed request with one argument: R0 contains a function code in the high-order byte and a channel number (see Section 2.2.1) or code in the low-order byte.
360-373	Used internally by the RT-11 monitor; these EMT codes should never be used by user programs.
340-357	Programmed request with arguments on the stack and/or in R0.
0-337	Version 1 programmed request. These EMTs use arguments both on the stack and in R0. They are supported for binary compatability with Version 1 programs.

A programmed request consists of a macro call followed, if necessary, by one or more arguments. All programmed requests start with a period (.) to distinguish them from user defined symbols and macros. Arguments supplied to a macro call must be legal assembler expressions since arguments are used as source fields in instructions (such as MOV) when the macros are expanded at assembly time. The following two formats are accepted by the monitor.

```
Format 1:  .PRGREQ arg1,arg2,...argn

Format 2:  .PRGREQ area,arg1,arg2,...argn
```

Format 1 contains the argument list arg1 through argn; no argument list pointer is required. Macros of this form generate either an EMT 374 or one of the EMTs 340-357. Certain arguments for this form can be omitted.

In format 2, area is a pointer to the argument block that contains the arguments arg1 through argn. This form always causes an EMT 375 to be generated. Blank fields are permitted; however, if the area argument is empty, the macro assumes that R0 points to a valid argument block (see Section 2.2.3). If any of the fields arg1 to argn are empty, the corresponding entries in the argument list are left untouched. Thus,

```
        .PRGREQ area,a1,a2
```

points R0 to the argument block at area and fills in the first and second arguments, while:

```
.PRGREQ area
```

points R0 to the block, and fills in the first word (request code) but does not fill in any other arguments.

The call:

```
.PRGREQ ,a1
```

assumes R0 points to the argument block and fills in the a1 argument, but leaves the a2 argument alone. The call:

```
.PRGREQ
```

generates only an EMT 375 and assumes that both R0 and the block to which it points are properly set up.

The arguments to RT-11 programmed request macros all serve as the source field of an instruction that moves a value into the argument block or R0. For example:

```
.PRGREQ  CHAR
```

expands into:

```
MOV   CHAR,R0
EMT   374
```

Care should be taken to make certain that the arguments specified are legal source fields and that the address accurately represents the value desired. If the value is a constant or symbolic constant, the immediate addressing mode [#] should be used; if the value is in a register, the register mnemonic [Rn] should be used; if the value is indirectly addressed, the appropriate register convention is necessary [@Rn]; and if the value is in memory, the label of the location whose value is the argument is used.

Following are some examples of both correct and incorrect macro calls. Consider the general request:

```
.PRGREQ area,arg1,...argn
```

A more common way of writing a request of this form is:

```
.PRGREQ #area,#arg1,...#argn
```

In this format, the address of area is put into register 0. Area is the tag that indicates the beginning of the argument block. For example:

```
        .PRGREQ #AREA,#4
              .
              .
              .
AREA:   .WORD 0,0,0
```

When a direct numerical argument is required, the # causes the correct value to be put into the argument block. For example:

```
.PRGREQ #area,#4
```

is correct, while:

```
.PRGREQ #area,4
```

is not. This form interprets the 4 as meaning "move the contents of location 4 into the argument block." Instead, the number 4 itself should be moved into the block.

If the request is written as:

```
.PRGREQ area,#4
```

it is interpreted as "use the contents of location area as the list pointer", when the address of area is actually desired. This expansion could be used with the following form:

```
          .PRGREQ LIST1,#4
             .
             .
             .
LIST1:    .WORD AREA
AREA:     .WORD 0,0,0
```

In this case, the content of location LIST1 is the address of the argument list. Similarly, this form is correct:

```
          .PRGREQ LIST1,NUMBER
                .
                .
                .
LIST1:    .WORD AREA
NUMBER:   .WORD 4
```

In this case, the contents of the locations LIST1 and NUMBER are the argument list pointer and data value, respectively.

NOTE

All registers except R0 are preserved across a programmed request. (In certain cases, R0 contains information passed back by the monitor; however, unless the description of a request indicates that a specific value is returned in R0, the contents of R0 are unpredictable upon return from the request). With the exception of calls to the Command String Interpreter (CSI), the position of the stack pointer is also preserved across a programmed request.

2.2 SYSTEM CONCEPTS

Some basic operational characteristics and concepts of RT-11 are described in the following sections.

2.2.1 Channel Number (chan)

A channel number is a logical identifier for a file or "set of data" used by the RT-11 monitor. It can have a value in the range 0 to 377 (octal)--0 to 255 (decimal). In RT-11, a channel is the logical connection between a channel number and all information that must be maintained between data transfers, such as device and file name. When a file is opened on a particular device, a channel number is assigned to that file. To refer to an open file, it is only necessary to refer to the appropriate channel number for that file.

2.2.2 Device Block (dblk)

A device block is a four-word block of Radix-50 information that specifies a physical device, file name and file type for an RT-11 programmed request. For example, a device block representing the file FILE.TYP on device DK: could be written as:

```
.RAD50  /DK /
.RAD50  /FIL/
.RAD50  /E  /
.RAD50  /TYP/
```

The first word contains the device name, the second and third words contain the file name, and the fourth contains the file type. Device, name, and file type must each be left-justified in the appropriate field. This string could also be written as:

```
.RAD50  /DK FILE  TYP/
```

Note that spaces must be used to fill out each field. Note also that the colon and period separators do not appear in the actual Radix-50 string. They are used only by the Command String Interpreter to delimit the various fields.

2.2.3 EMT Argument Blocks

Programmed requests that call the monitor via EMT 375 use R0 as a pointer to an argument list. In general, this argument block appears as follows when the EMT instruction is executed:

address	contents	
R0→area:	function code	channel number
[R0+2]	argument 1	
[R0+4]	argument 2	
	⋮	
[R0+(n*2)]	argument n	

R0 points to location x. The even (low-order) byte of location x contains the channel number named in the macro call. If no channel number is required, the byte is set to 0. The odd (high-order) byte of x is a code specifying the function to be performed. Locations x+2, x+4, etc., contain arguments to be interpreted. These are described in detail under each request.

Requests that use EMT 374 set up R0 with the channel number in the even byte and the function code in the odd byte. They require no other arguments.

2.2.4 Important Memory Areas

The memory areas for vector addresses, the resident monitor and certain system communication information are particularly important for RT-11's operation. Some addresses in these areas can be used by user programs, but others must not be used under any circumstances.

2.2.4.1 Vector Addresses (0-37 octal, 60-477 octal)

- Certain areas of memory between 0 and 477 are reserved for use by RT-11. The monitor does not load these locations from the memory image file when it initiates a program. (The monitor RUN command does not load these words, for example.) However, no hardware memory protection is supplied. Therefore programs should never alter the contents of these areas. If they are destroyed by a program, the system must be re-bootstrapped or the program must restore them.

Locations	Contents
0,2	Monitor restart. Executes the .EXIT request and returns control to program. Modifying these locations while using the XM monitor always causes a system crash.
4,6	Time out or bus error 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.
30,32	EMT trap vector.
34,36	TRAP instruction vector (in an FB or XM environment this area is loaded by R, RUN, GET and FRUN.
40-51	RT-11 system communication area (this area is loaded by R, RUN and GET).
52-57	RT-11 system communication area (see Section 2.2.4.3, below).
60,62	Console Terminal input interrupt vector.
64,66	Console Terminal output interrupt vector.
100,102	KW11L vector.
104,106	KW11P vector.
160,162	RL01 Disk vector.
200,202	LP11/LS11/LPV11 Line printer vector.
204,206	RF11,RS03/4 vector.
210,212	RK611/RK06, RK07 Disk pack vector.

Locations	Contents
214,216	TC11 vector.
220,222	RK11/RKV11 RK05 Disk vector.
224,226	TJU16,TM11,TS03 Magnetic tape vector.
250,252	KT11 Memory management fault vector.
254,256	RP04/11 Disk pack vector.
260,262	TA11 Cassette vector.
264,266	RX11/RXV11 RX01,RX211/RX2V1 RX02 Diskette vector.
320,322 324,326 330,332	VT11/VS60 Graphics terminal vectors.

2.2.4.2 **Resident Monitor** - Chapter 1 describes the placement of monitor components when the SJ monitor, the FB monitor or the XM monitor is brought into memory; the approximate size of each monitor component and the size of the area available for handlers and user programs is included.

2.2.4.3 **System Communication Area** - RT-11 uses bytes 40-57 to hold information about the program currently executing, as well as certain information used only by the monitor. A description of these bytes follows:

Bytes	Meaning and Use
40,41	Start address of job. When a file is linked to create an RT-11 memory image, this word is set to the starting address of the job. When a foreground program is executed, the FRUN processor relocates this word to contain the actual starting address of the program.
42,43	Initial value of the stack pointer. If it is not set by the user program in an .ASECT, it defaults to 1000 or the top of the .ASECT in the background, whichever is larger. If a foreground program does not specify a stack pointer in this word, a default stack (128 decimal bytes) is allocated by FRUN immediately below the program. The initial stack pointer can also be set by an option of the linker.
44,45	Job Status Word (JSW). Used as a flag word for the monitor. Certain bits are maintained by the monitor exclusively while others may be set or cleared by the user job. Since the currently unassigned bits may be used in future releases of RT-11, user programs should not use these bits for internal flags.

Those bits in the following list marked by an asterisk are bits that can be set by the user job.

Bytes	Meanings and Use	
	Bit Number	**Meaning**
	15	USR swap bit. (SJ only.) The monitor sets this bit when programs do not require the USR to be swapped. See Section 2.2.5 for details on USR swapping.
	*14	Lower-case bit. Disables automatic conversion of lower-case to upper-case when set. EDIT sets this bit when the EL command is typed.
	*13	Reenter bit. When set, this bit indicates that the program may be restarted from the terminal with the REENTER command.
	*12	Special mode TT bit. When set, this bit indicates that the job is in a "special" keyboard mode of input. Refer to the explanation of the .TTYIN/.TTINR requests for details.
	10	Virtual image bit. (XM only.) When set, this bit indicates that the job to be loaded is a virtual image. It must be set in the execute file (with a .ASECT or PATCH) before the program is loaded.
	*11	Pass line to KMON bit. If this bit is set when a user program exits, it indicates that the user program is passing a command line to the KMON. The command line is stored in the CHAIN information area (500-776). Refer to the .EXIT example in Section 2.4.15. This bit is not available to foreground jobs under the FB and XM monitors.
	9	Overlay Bit. Set (by the linker) if the job uses the linker overlay structure.
	8	CHAIN bit. If this bit is set in a job's save image, words 500-776 are loaded 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 across .CHAINs.) The bit is set when a job is running if and only if the job was actually entered with .CHAIN.

*7 Error halt bit. (SJ only.) When set, this bit indicates a halt on an I/O error. If the user desires to halt when any I/O device error occurs, this bit should be set. (SJ only.)

Bytes	Meaning and Use	
	Bit Number	**Meaning**
	*6	Inhibit TT wait bit. For use with the FB monitor. When set, this bit inhibits the monitor from entering a console terminal wait state. Refer to the sections concerning .TTYIN/.TTINR and .TTYOUT/.TTOUTR for more information.
	*5	Filter escape sequences. This bit is ignored if bit 4 is not set. Bit 5 is set to specify that escape sequences are to be echoed (if not in special mode), but not passed to the program. If this bit is not set, escape sequences are passed to the user program, but not echoed.
	*4	Process escape sequences. This bit is set to enable any escape sequence support. If this bit is not set, the same support is provided as in version 2C.
	3	Reserved for system use. Users should not attempt to use this bit.
	2-0	Reserved for internal use.

46,47 USR load address. Normally 0, this word can be set to any valid word address in the user's program. If 0, the USR is loaded in the default location through an address contained in offset 266 of RMON. If this value is not 0, the USR is simply loaded at the specified address (address in word location). See Section 2.2.5, Swapping Algorithm, for details of use.

50,51 High memory address. The monitor maintains the highest address the user program can use in this word. The linker sets it initially to the high limit value. It is modified only by the .SETTOP monitor request.

52 EMT error code. If a monitor request results in an error, the code number of the error is always returned in byte 52 and the carry bit is set. Each monitor call has its own set of possible errors. It is recommended that the user program refer to byte 52 with absolute addressing rather than relative addressing. For example:

```
ERRBYT = 52
TSTB ERRBYT          ;RELATIVE ADDRESSING
TSTB @#ERRBYT        ;ABSOLUTE ADDRESSING
```

NOTE

Location 52 must always be addressed as a byte, never as a word, since byte 53 has a different function.

Bytes	Meaning and Use
53	User program error code (USERRB). If a user program encounters errors during execution, it indicates the error by using this byte. The KMON examines this byte when a program terminates. If a significant error is reported by the user program, the 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 with one of the following states:

- Success
- Warning
- Error
- Severe Error

The program status is successful when the execution of the program is completely free of any errors.

The warning status indicates that warning messages occurred, but the execution of the program is completed. The MACRO assembler sets the warning level bit when it detects errors at assembly time.

The error status indicates that a user error occurred and the execution of the program was not completed. This level is used when the program produces an output file even though the file may contain errors. 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 error 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 a user program. The user program reports status to RT-11 through byte 53, returning through a hard or soft exit.

The following bits correspond to the four status levels discussed previously.

Bytes	Meaning and Use

Bit	Meaning (if set to 1)
7-4	Reserved for future use (should not be set or cleared by program).
3	Severe error
2	Error
1	Warning
0	Success

Programs should never clear byte 53 and should only set it through a BISB instruction, as in the following example:

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

54,55 Address of the beginning of the resident monitor. RT-11 always loads the monitor into the highest available memory locations; this word points to its first location. It must never be altered by the user. Doing so causes RT-11 to malfunction.

56 Fill character (seven-bit ASCII). Some high-speed terminals require filler (null) characters after printing certain characters. Byte 56 should contain the ASCII seven-bit representation of the character after which fillers are required.

57 Fill count. This byte specifies the number of fill characters that are required. The number of characters is determined by hardware. If bytes 56 and 57 equal 0, no fill is required.

The terminals requiring fill characters are:

Terminal	No. of fills	Word 56 Value
Serial LA30 @ 300 baud	10 after <RET>	5015
Serial LA30 @ 150 baud	4 after <RET>	2015
Serial LA30 @ 110 baud	2 after <RET>	1015
VT05 @ 2400 baud	4 after <LF>	2012
VT05 @ 1200 baud	2 after <LF>	1012
VT05 @ 600 baud	1 after <LF>	412

2.2.5 Swapping Algorithm

Programmed requests are divided into two categories according to whether or not they require the USR to be in memory (see Table 2-2). Any request that requires the USR in memory can also require that a portion of the user program be saved temporarily in the system device swap file (that is, be "swapped out" and stored in the file SWAP.SYS) to provide room for the USR. The USR is then read into memory. In the XM monitor, the USR is always resident, and therefore never swapped. During normal operations, this swapping is invisible to the user. However, it is possible to optimize programs so that they require little or no swapping.

The following items should be considered if a swap operation is necessary:

1. The background job - If a .SETTOP request in a background job specifies an address beyond the point at which the USR normally resides, a swap is required when the USR is called. More details concerning the .SETTOP request are in Section 2.4.3.6.

2. The value of location 46 - If the user either assembles an address into word 46 or moves a value there while the program is running, RT-11 uses the contents of that word as an alternate place to swap the USR. If location 46 is 0, this indicates that the USR will be at its normal location in high memory.

3. Monitor offset 374 - The contents of monitor offset 374 indicates the size of the USR in bytes. This can be useful in planning memory allocation. (See Section 2.2.6.)

NOTES

1. If the USR does not require swapping, the value in location 46 is ignored. Swapping is a relatively time-consuming operation and should be avoided, if possible.

2. A foreground job must always have a value in location 46 unless it is certain that the USR will never be swapped. If the foreground job does not allow space for the USR and a swap is required, a fatal error occurs. (The SET USR NOSWAP command ensures that the USR is always resident.)

3. Care should be taken when specifying an alternate address in location 46. The SJ monitor does not verify the legality of the USR swap address, and if the area to be swapped overlays the resident monitor, the system is destroyed.

4. The user should also take care that the USR is never swapped over any of the following areas: the program stack; any parameter block for calls to the USR; any I/O buffers, device handlers, or completion routines being used when the USR is called.

For example:

```
.TITLE  USR.MAC
;THIS IS AN EXAMPLE OF THE WAY A BACKGROUND PROGRAM CAN AVOID
;UNNECESSARY USR SWAPPING.
        .MCALL  .SETTOP,.EXIT,.GVAL
USRLOC  = 266                           ;POINTER TO USR LOCATION IS
                                        ;AT 266 BYTES INTO RMON.
START:
        .GVAL   #AREA,#USRLOC           ;R0 -> USR
        TST     -(R0)                   ;POINT JUST BELOW
        CMP     R0,@#50                 ;DOES USR SWAP OVER US?
        BMI     1$                      ;NO, OK
        MOV     #-2,R0                  ;YES, USR MUST SWAP
1$:     .SETTOP                         ;ASK FOR MEMORY UP TO USR
        MOV     R0,HILIM                ;R0 = HIGH LIMIT OF MEMORY
                                        ;ACTUALLY GRANTED BY MONITOR.
        .EXIT
HILIM:  .WORD   0                       ;CONTAINS HI LIMIT OF MEMORY
AREA:   .BLKW   2                       ;EMT ARGUMENT BLOCK
        .END    START
```

2.2.6 Offset Words

There are several words that always have fixed positions relative to the start of the resident monitor. It is often advantageous for user programs to be able to access these words. This is done using the .GVAL programmed request in the following form:

.GVAL #area,#offse

Here, area is a two-word argument block and offse is a number from the following list.

OFFSET (Bytes)	Contents
266	Start of normal USR area. This is where the USR resides when it is called into memory and location 46 is 0. It is useful to be able to perform a .SETTOP in a background job so that the USR does not swap, and once called in, remains resident. (An example is in Section 2.2.5.)
270	Address of 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 added to RT-11 must also use this exit location.

OFFSET (Bytes)	Contents
272	Special device error word. This word is used by non-RT-11 file structured devices (such as MT and CT) to report errors to the monitor.
275	Unit number of system device (device from which system was last bootstrapped).
276	Monitor version number. The user can always access the version number to determine if the most recent monitor is in use. For RT-11 Version 3B, this value is 3.
277	Monitor release level. This number identifies the release level of the monitor version specified in byte 276. For version 3B, the value is 2.
300	Configuration word. This is a string of 16 bits that indicates information about either the hardware configuration of the system or a software condition. Another configuration word is available at offset 370 that contains additional data. The bits and their meanings are:

Bit #	Meaning
0	0 = SJ Monitor 1 = FB Monitor if bit 12 is not set, XM monitor if bit 12 is set
2	1 = graphics display hardware exists (VT11 or VS60)
3	1 = RT-11 BATCH is in control of the background
5	0 = 60-cycle clock 1 = 50-cycle clock
6	1 = FP11 floating-point hardware exists
7	0 = No foreground job is in memory 1 = Foreground job is in memory
8	1 = User is linked to the graphics scroller
9	1 = USR is permanently resident (via a SET USR NOSWAP - USR is always resident in XM)
11	1 = Processor is a PDP-11/03 (that is, there is no program status word on this processor)
12	1 = a mapped system running under XM monitor
13	1 = The system clock has a status register
14	1 = KW11-P clock exists and can be used by programs
15	1 = either an L clock or a P clock (depending on the system generation procedure used) is present

The other bits are reserved for future use and should not be used by user programs.

OFFSET (Bytes)	Contents
304-313	These locations contain the addresses of the console terminal control and status registers (but they are not used when the multi-terminal option is present). The order is: 304 Keyboard status 306 Keyboard buffer 310 Printer status 312 Printer buffer These locations can be changed, for example, to reflect a second terminal; thus RT-11 can be made to run on any terminal connected to the machine through the DL11 terminal interface.
314	The maximum file size allowed in a 0 length .ENTER. This can be adjusted by the user program or by using the PATCH program to be any reasonable value. The default value is 177777 (octal) blocks, allowing an essentially unlimited file size.
324	Address of .SYNCH entry. User interrupt routines can enter the monitor through this address to synchronize with the job they are servicing.
354	Address of VT11 or VS60 display processor display stop interrupt vector.
360	Move to PS routine. The routine is called by the .MTPS macro to do processor independent moves to the program status word.
362	Move from PS routine. The routine is called by the .MFPS macro to do processor independent moves from the program status word.
366	Indirect file and command language state word.
370	Extension configuration word. This is a string of 16 bits used to indicate the presence of an additional set of hardware options on the system. The bits and their meanings are:

Bit #	Meaning
0	1 = cache memory is present
1	1 = parity memory is present
2	1 = readable switch register is present
3	1 = writeable console display register is present
8	1 = EIS option is present

OFFSET (Bytes)	Contents
	Bit # — Meaning 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 14 — 1 = processor is PDP-11/70 15 — 1 = processor is PDP-11/60 The other bits are reserved for future use and should not be used by user programs.
372	SYSGEN options word. The bit pattern indicates important SYSGEN options and must not be modified by user programs or patches. The bits and their meanings are: Bit # — Meaning 0 — 1 = error logging option is present 1 — 1 = memory management option is present 2 — 1 = device I/O time-out option is present 9 — 1 = memory parity option is present 10 — 1 = SJ mark time option is present 11-12 — 00 = no escape sequences recognized 01 = option to recognize DIGITAL escape sequences is present 10 = option to recognize ANSI escape sequences is present 13 — 1 = multi-terminal option is present The other bits are reserved for future use and should not be used by user programs.
374	Size of USR in bytes. Programs should use this information to dynamically determine the size of the region needed to swap the USR.
377	Depth of nesting of indirect files (default is 3). This value must be referred to as a byte. It can be patched or set by programs to change the nesting depth of indirect files.
400	Internal offset for use by BATCH only.
402	Byte offset to fork request processor from start of resident monitor (contents of 54).

2.2.7 File Structure

RT-11 uses a contiguous file structure. This type of structure requires that every file on a device be made up of a contiguous group of physical blocks. Thus, a file that is 19 blocks long occupies 19 contiguous blocks on the device.

A contiguous area on a device can be in one of the following categories:

1. Permanent file. This is a file that has been created with .ENTER and then .CLOSEd. Any named files that appear in a directory listing are permanent files.

2. Tentative file. Any file that has been created with .ENTER but not .CLOSEd is a tentative file entry. When the .CLOSE request is given, the tentative entry becomes a permanent file. If a permanent file already exists under the same name, the old file is deleted when the tentative file is .CLOSEd. If a .CLOSE is never given, the tentative file is treated like an empty entry. The tentative file is deleted when a new tentative file with the same name is created.

3. Empty entry. When disk space is unused or a permanent file is deleted, an empty entry is created. Empty entries appear in a full directory listing as <UNUSED> n, where n is the decimal block length of the empty area.

Since a contiguous structure does not automatically consolidate unused disk space, a device can eventually become fragmented with many scattered empty entries. RT-11 has a SQUEEZE command to collect all empty areas and create a single empty entry at the end of a device.

2.2.8 Completion Routines

Completion routines are user-written routines that are entered following the completion of some external operation. A completion routine can be entered after an I/O data transfer, after some number of clock ticks or after a user-specified interrupt. On entry to an I/O completion routine, R0 contains the contents of the channel status word for the operation and R1 contains the octal channel number of the operation. The carry bit is not significant.

Completion routines are handled differently in the SJ and the FB or XM versions of RT-11. In the SJ version, completion routines are totally asynchronous and can interrupt one another. In FB and XM, completion routines do not interrupt each other. Instead they are queued and made to wait until the correct job is running. For example, if a foreground job is running and an I/O transfer initiated by a background job completes with a specified completion routine, the background routine is queued and does not execute until the foreground gives up control of the system. If the foreground is running and a foreground I/O transfer completes and wants a completion routine, that routine is entered immediately if the foreground is not already executing a completion routine. If it is in a completion routine, that routine continues to termination, at which point any other completion routines are entered in a first in/first out order. If the background is running and a foreground I/O transfer completes with a specified completion routine, the background is suspended and the foreground routine is entered immediately.

The restrictions that must be observed when writing completion routines are:

1. Completion functions cannot issue a request that would cause the USR to be swapped in. They are primarily used for issuing .READ and .WRITE requests, not for opening or closing files, etc. A fatal monitor error is generated if the USR is called from a completion routine.

2. Completion routines should never reside in the memory space that is used for the USR, since the USR can be interrupted when I/O terminates and the completion routine is entered. If the USR has overlaid the routine, control passes to a random place in the USR, with a HALT or error trap the likely result.

3. The routine must be exited with an RTS PC (because it is called from the monitor with a JSR PC,ADDR, where ADDR is the user-supplied entry point address).

4. If a completion routine uses registers other than R0 or R1, it must save them upon entry and restore them before exiting. Other requests cannot transfer data between the mainline program and the completion routine.

5. In XM, completion routines must remain mapped while the request is active and the routine can be called.

2.2.9 Using the System Macro Library

User programs for RT-11 should always be written using the macros provided in the system macro library (SYSMAC.MAC), supplied with RT-11. This ensures source level compatibility among all user programs and allows easy modification by redefining a macro. A listing of SYSMAC.MAC appears in Appendix B.

Suggestions for writing foreground programs are in Chapter 1 (FB Programming and Device Handlers). Chapter 1 should be read in conjunction with Chapter 2 before coding FB programs.

2.2.10 Error Reporting

In processing a programmed request, the monitor can detect an error condition that must be reported to the user program. RT-11 programmed requests use three methods of reporting these errors: the carry (C) bit, the error byte (byte 52 in the system communication area), and the monitor error message. The carry bit is returned clear after normal termination of a programmed request, and set after an abnormal termination. Almost all requests should be followed by a BCS or BCC instruction to detect a possible error. When the carry bit is set, the error byte usually contains additional information about the error. The meanings of values in the error byte are described individually for each request. In most cases, the user program should test the error byte when the carry bit is set. The values contained in the error byte are not significant when the carry bit is clear. Certain serious or non-recoverable error conditions cause a monitor error message to be printed at the console terminal. A user-program can use the .SERR programmed request to cause these errors to be reported through the carry bit and the error byte, in which case the error byte will contain a negative error code.

2.3 TYPES OF PROGRAMMED REQUESTS

There are three types of services that the monitor makes available to the user through programmed requests. These are:

1. Requests for file manipulation
2. Requests for data transfer
3. Requests for miscellaneous services

Table 2-1 summarizes the programmed requests in each of these categories alphabetically. Some requests function only in a FB and XM environment and are ignored under the SJ monitor. The EMT and function code for each request (where applicable) are shown in octal. It should be noted as a general rule that only six characters (such as .CHCOP) are significant to the MACRO assembler. Longer forms are shown for readability only.

Table 2-1
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
File Manipulation Requests		
.CHCOPY*	375 13	Establishes a link and allows one job to access another job's channel.
.CLOSE	374 6	Closes the specified channel.
.DELETE	375 0	Deletes the file from the specified device.
.ENTER	375 2	Creates a new file for output.
.LOOKUP	375 1	Opens an existing file for input and/or output via the specified channel.
.PURGE	374 3	Clears out a channel.
.RENAME	375 4	Changes the name of the indicated file to a new name. If this request is attempted with magtape, the handler returns an illegal operation code.
.REOPEN	375 6	Restores the parameters stored via a .SAVESTATUS request and reopens the channel for I/O.
.SAVESTATUS	375 5	Saves the status parameters of an open file in user memory and frees the channel for future use.

(continued on next page)

Table 2-1 (Cont.)
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
Data Transfer Requests		
.MTIN*	375 37	Operates as a .TTYIN for multi-terminal configuration.
.MTOUT*	375 37	Operates as a .TTYOUT for multi-terminal configuration.
.MTPRNT*	375 37	Operates as a .PRINT request for a multi-terminal configuration.
.PRINT	351 --	Outputs an ASCII string terminated by a 0 byte or a 200 byte.
.RCVD* .RCVDW* .RCVDC*	375 26	Receives data. Allows a job to read messages or data sent by another job in an FB environment. The three modes correspond to the READ, .READC, and .READW modes.
.READ	375 10	Transfers data on the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of I/O.
.READC	375 10	Transfers data on the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers asynchronously to the routine specified in the .READC request.
.READW	375 10	Transfers data via the specified channel to a memory buffer and returns control to the user program only after the transfer is complete.
.SDAT* .SDATC* .SDATW*	375 25	Allows the user to send messages or data to the other job in an FB environment. The three modes correspond to the .WRITE, .WRITC, and .WRITW modes.
.SPFUN	375 32	Performs special functions on magtape, cassette, diskette and some disk devices.
.TTYIN .TTINR	340 --	Transfers one character from the keyboard buffer to R0.
.TTYOUT .TTOUTR	341 --	Transfers one character from R0 to the terminal input buffer.

(continued on next page)

Table 2-1 (Cont.)
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
.WRITE	375 11	Transfers data on the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of the I/O.
.WRITC	375 11	Transfers data on the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers asynchronously to the routine specified in the .WRITC request.
.WRITW	375 11	Transfers data on the specified channel to a device and returns control to the user program only after the transfer is complete.
Miscellaneous Services		
.CDFN	375 15	Defines additional channels for I/O.
.CHAIN	374 10	Chains to another program (in the background job only).
.CRAW**	375 36	Creates a window in virtual memory.
.CRRG**	375 36	Creates a region in extended memory.
.CMKT	375 23	Cancels an unexpired mark time request.
.CNTXSW*	375 33	Requests that the indicated memory locations be part of the FB context switch process.
.CSIGEN	344 --	Calls the Command String Interpreter (CSI) in general mode.
.CSISPC	345 --	Calls the CSI in special mode.
.CSTAT*	375 27	Returns the status of the channel indicated.
.DATE	374 12	Moves the current date information into R0.
.DEVICE*	375 14	Allows the user to disable device interrupts in FB upon program termination.
.DSTATUS	342 --	Returns the status of a particular device.

(continued on next page)

Table 2-1 (Cont.)
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
.ELAW**	375 36	Cancels an address window in virtual memory.
.ELRG**	375 36	Cancels an allocated region in extended memory.
.EXIT	350 --	Exits the user program.
.FETCH	343 --	Loads device handlers into memory.
.GMCX**	375 36	Returns mapping status of a specified window.
.GTIM	375 21	Gets time of day.
.GTJB	375 20	Gets parameters of the current job.
.GTLIN	345 --	Accepts an input line from either an indirect file or from the console terminal.
.GVAL	375 34	Returns monitor fixed offsets in a pseudo-protected manner.
.HERR	374 5	Specifies termination of the job on fatal errors.
.HRESET	357 --	Terminates I/O transfers and does a .SRESET operation.
.INTEN	--- --	Notifies the monitor that an interrupt has occurred, requests system state and sets the processor priority to the correct value.
.LOCK	346 --	Makes the monitor User Service Routines (USR) permanently resident until .EXIT or .UNLOCK is executed. The user program is swapped out, if necessary.
.MAP*	375 36	Maps a virtual address window to extended memory.
.MFPS	--- --	Reads the priority bits in the processor status word (but does not read the condition codes).
.MRKT	375 22	Marks time; that is, sets an asynchronous routine to occur after a specified interval.
.MTATCH*	375 37	Attaches a terminal for exclusive use by the requesting job.

(continued on next page)

Table 2-1 (Cont.)
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
.MTDTCH*	375 37	Detaches a terminal from one job and frees it to be used by other jobs.
.MTGET*	375 37	Returns status of specified terminal to caller.
.MTSET*	375 37	Determines and modifies terminal status in a multi-terminal configuration.
.MTPS	--- --	Sets the priority bits, condition codes, and T bit in the processor status word.
.MTRCTO*	375 37	Resets the CTRL/O flag for the designated terminal.
.MWAIT*	374 11	Waits for messages to be processed.
.PROTECT*	375 31	Requests that vectors in the area from 0-476 be given exclusively to the current job.
.QSET	353 --	Increases the size of the monitor I/O queue.
.RCTRLO	355 --	Enables output to the terminal.
.RELEAS	343 --	Removes device handlers from memory.
.RSUM*	374 2	Causes the main line of the job to be resumed when it was suspended with .SPND.
.SCCA	375 35	Enables intercept of CTRL/C commands.
.SERR	374 4	Inhibits most fatal errors from aborting the current job.
.SETTOP	354 --	Specifies the highest memory location to be used by the user program.
.SFPA	375 30	Sets user interrupt for floating point processor exceptions.
.SPND*	374 1	Causes the running job to be suspended.
.SRESET	352 --	Resets all channels and releases the device handlers from memory.
.SYNCH	--- --	Enables user program to perform monitor programmed requests from within an interrupt service routine.

(continued on next page)

Table 2-1 (Cont.)
Summary of Programmed Requests

Mnemonic	EMT Code	Purpose
.TLOCK*	374 7	Indicates if the USR is currently being used by another job and performs a .LOCK if the USR is available.
.TRPSET	375 3	Sets a user intercept for traps to locations 4 and 10.
.TWAIT*	375 24	Suspends the running job for a specified amount of time.
.UNLOCK	347 --	Releases the USR if a .LOCK was done and swaps in the user program, if required.
.UNMAP*	375 36	Unmaps a virtual memory address window.
.UNPROTECT*	374 31	Cancels the .PROTECT vector protection request.
..V1..	--- --	Provides compatibility with version 1 format.
..V2..	--- --	Provides compatibility with version 2 format.
.WAIT	374 0	Waits for completion of all I/O on a specified channel.

*FB and XM monitors

**XM monitor only

Requests requiring the USR (as explained in Section 2.2.5) differ between the SJ and FB monitors. Table 2-2 indicates which requests require the USR to be in memory. The .CLOSE request on non-file-structured devices (LP:, PC:, TT:, etc.) does not require the USR under any monitor.

The USR is not reentrant and cannot be shared by concurrent jobs. When the USR is in use by one job, another job requiring it must queue up for it. This is particularly important for concurrent jobs when devices such as magnetic tape or cassette are active.

For example, USR file operations on tape devices require a linear search of the tape. When a background job is running the USR, the foreground job is locked out until the tape operation is completed. Since that can take considerable time, the programmer should be aware of the problem. In the FB and XM monitors, the .TLOCK request (see Section 2.4.56) can be used by a foreground job to check USR availability.

Table 2-2
Requests Requiring the USR

Request	SJ	FB	XM
.CDFN	Yes*	No	No
.CHAIN	No	No	No
.CHCOPY	--	No	No
.CLOSE (see Note 1)	Yes	Yes	Yes
.CMKT	No	No	No
.CNTXSW	--	No	No
.CRAW	--	--	No
.CRRG	--	--	No
.CSIGEN	Yes	Yes	Yes
.CSISPC	Yes	Yes	Yes
.CSTAT	--	No	No
.DATE	No	No	No
.DELETE	Yes	Yes	Yes
.DEVICE	--	No	No
.DSTATUS	Yes	Yes	Yes
.ELAW	--	--	No
.ELRG	--	--	No
.ENTER	Yes	Yes	Yes
.EXIT	Yes	Yes	Yes
.FETCH	Yes	Yes	Yes
.GMCX	--	--	No
.GTIM	No	No	No
.GTJB	--	No	No
.GTLIN	Yes*	Yes	Yes
.GVAL	No	No	No
.HERR	No	No	No
.HRESET	Yes*	No	No
.INTEN	No	No	No
.LOCK (see Note 2)	Yes	Yes	Yes
.LOOKUP	Yes	Yes	Yes
.MAP	--	--	No
.MFPS	No	No	No
.MRKT	No	No	No
.MTATCH	--	No	No
.MTDTCH	--	No	No
.MTGET	--	No	No
.MTIN	--	No	No
.MTOUT	--	No	No
.MTPRNT	--	No	No
.MTPS	No	No	No
.MTRCTO	--	No	No
.MTSET	--	No	No
.MWAIT	--	No	No
.PRINT	No	No	No

* Those requests marked with an asterisk always require a fresh copy of the USR to be read in before they can be executed. When executing such a request, the USR must be read in from the system device even if there is a copy of the USR presently in memory.

Note 1: Only if channel was opened with .ENTER.

Note 2: Only if USR is in a swapping state.

Note 3: Only if USR is not in use by the other job.

(continued on next page)

Table 2-2 (Cont.)
Requests Requiring the USR

Request	SJ	FB	XM
.PROTECT	--	No	No
.PURGE	No	No	No
.QSET	Yes*	Yes*	Yes*
.RCTRLO	No	No	No
.RCVD/.RCVDC/.RCVDW	--	No	No
.READ/.READC/.READW	No	No	No
.RELEAS	Yes	Yes	Yes
.RENAME	Yes	Yes	Yes
.REOPEN	No	No	No
.RSUM/.SPND	--	No	No
.SAVESTATUS	No	No	No
.SCCA	No	No	No
.SDAT/.SDATC/.SDATW	--	No	No
.SERR	No	No	No
.SETTOP	No	No	No
.SFPA	No	No	No
.SPFUN	No	No	No
.SRESET	Yes*	No	No
.SYNCH	No	No	No
.TLOCK (see Note 3)	Yes	Yes	Yes
.TRPSET	No	No	No
.TTINR/.TTYIN	No	No	No
.TTOUTR/.TTYOUT	No	No	No
.TWAIT	--	No	No
.UNLOCK	No	No	No
.UNMAP	--	--	No
.UNPROTECT	--	No	No
.WAIT	No	No	No
.WRITE/.WRITC/.WRITW	No	No	No

* Those requests marked with an asterisk always require a fresh copy of the USR to be read in before they can be executed. When executing such a request, the USR must be read in from the system device even if there is a copy of the USR presently in memory.

Note 1: Only if channel was opened with .ENTER.

Note 2: Only if USR is in a swapping state.

Note 3: Only if USR is not in use by the other job.

2.3.1 System Macros

The following macros are included in the system macro library, but are not programmed requests because they do not generate an EMT instruction:

..V2..
..V1..

They can be used in the same manner as the other macro calls; their explanations follow.

..V1../..V2..

2.3.1.1 ..V1../..V2..

Any version 1 and/or version 2 program that uses system macros must specify the version format in which the macro calls are to be expanded. Assembly errors at macro calls result if the proper version designation is not made. In version 3B, ..V1.. and ..V2.. are unnecessary since the expansions are made automatically. The ..V1.. and ..V2.. macros are retained only for compatibility with earlier systems.

The ..V1.. macro call enables all macro expansions to occur in Version 1 format.

Macro Call:
```
.MCALL ..V1..
..V1..
```

This causes all macros in the program to be assembled in version 1 form and the symbol ...V1 to be set equal to 1. User programs should not use the ...V1 symbol.

To cause all macro expansions to occur in version 2 format, the ..V2.. macro call is used.

Macro Call:
```
.MCALL ..V2..
..V2..
```

The ..V2.. macro causes the ...V1 symbol to equal 2. As with the V1 case, user programs should not use the ...V2 symbol.

Run-time or assembly errors can occur if both the ..V1.. and ..V2.. macro calls are used in a program.

All examples in this chapter illustrate the format for version 3 and later systems.

NOTE

> It is possible for user programs to exist in which version 1 and version 2 or 3 macros are present. This is allowable by invoking the ..V1.. macro and using those macros that have no version 1 counterpart as if the ..V1.. macro had not been used.
>
> This causes all macros that existed in version 1 to assemble in version 1 format, while those macros new to version 2 or version 3 are correctly generated as version 2 or version 3 macros. Note that in this case a macro that existed in version 1 (such as .READ) will expand in the version 1 format.

2.4 PROGRAMMED REQUEST USAGE

This section presents the programmed requests alphabetically and describes each one in detail. The following parameters are commonly used as arguments in the various calls:

addr	an address, the meaning of which depends on the request being used.
area	a pointer to the EMT argument list (for those requests that require a list) -- see Section 2.2.3.
blk	a block number specifying the relative block in a file where an I/O transfer is to begin.
buf	a buffer address specifying a memory location into which or from which an I/O transfer is to be performed; this address has to be word-aligned, i.e., an even address and not a byte or odd address.
cblk	the address of the five-word block where channel status information is stored.
chan	a channel number in the range 0-377(octal).
chrcnt	a character count in the range 1-255 (decimal).
code	a flag used to indicate whether the code in an area form (EMT 375) of a programmed request is to be set.
crtn	the entry point of a completion routine -- see Section 2.2.8.
dblk	the address of a four-word Radix-50 descriptor of the file to be operated upon -- see Section 2.2.2.
func	a numerical code indicating the function to be performed.
num	a number, the value of which depends on the request.
seqnum	file number -- for cassette operations if this argument is blank, a value of 0 is assumed.

For magtape operation, it describes a file sequence number that can have the following values:

Value	Meaning
0	For .LOOKUP, this value rewinds the magtape and spaces forward until the file name is found. For .ENTER it rewinds the magtape and spaces forward until the file name is found or until the logical end of tape is detected. If the file name is found, an error return is taken.

n — Where n is any positive number. This value positions the magtape at file sequence number n. If the file represented by the FSN is greater than two files away from the beginning of tape, a rewind is performed. If not, the tape is backspaced to the beginning of the file.

Value	Meaning
-1	For .LOOKUP or .ENTER, this value suppresses rewinding and searches for a file name from the current tape position. Note that if the position is unknown, the handler executes a positioning algorithm that involves backspacing until an EOF label is found. The user should not use any other value since all other negative values are reserved for future use.
-2	For .ENTER, the tape is rewound and spaces forward until the file is found or end of tape is detected. The file is then entered causing a new end of tape when the file is closed.

unit — the logical unit number of a particular terminal in a multi-terminal system.

wcnt — a word count specifying the number of words to be transferred to or from the buffer during an I/O operation.

The RT-11 MACRO assembler supports keyword macro arguments. All of the arguments described above can be encoded using their keyword form (see the PDP-11 MACRO-11 Language Reference Manual for details).

A new argument code is included for all EMT 375 area versions of the macros. It is used for explicit control in expanding an EMT programmed request. In the 375 EMTs, the high byte of the area (pointed to by R0) contains an identifying code for the request. Normally, this byte is set if the macro invocation of the programmed request specifies the area argument, and remains unaffected if the programmed request omits the area argument. If the macro invocation contains CODE=SET, the high byte of the first word of the area is always be set to the appropriate code. This is true whether or not area is specified.

If CODE=NOSET is in the macro invocation, the high byte of the first word of area remains unaffected. This is true whether or not area is specified. The latter case can be used to avoid setting the code when the programmed request is being set up. This might be done because it is known to be set correctly from an earlier invocation of the request using the same area, or because the code was statically set during the assembly process.

Additional information concerning these parameters (and others not defined here) is provided as necessary under each request.

.CDFN

2.4.1 .CDFN

The .CDFN request redefines the number of I/O channels (see Section 2.2.1). Each job, whether foreground or background, is initially provided with 16(decimal) I/O channels, numbered 0-15. .CDFN allows the number to be expanded to as many as 255 (decimal) channels (0-254).

The space used to contain the new channels is taken from within the user program. Each I/O channel requires five words of memory. Therefore, the user must allocate 5*n words of memory, where n is the number of channels to be defined.

It is recommended that the .CDFN request be used at the beginning of a program, before any I/O operations have been initiated. If more than one .CDFN request is used, the channel areas must either start at the same location or not overlap at all. The two requests .SRESET and .HRESET cause the user's channels to revert to the original 16 channels defined at program initiation. Hence, any .CDFNs must be reissued after using .SRESET or .HRESET.

Note that .CDFN defines new channels; the space for the previously defined channels cannot be used. Thus, a .CDFN for 20(decimal) channels (while 16 original channels are defined) creates 20 new I/O channels; the space for the original 16 is unused, but the contents of the old channel set are copied to the new channel set.

Note that if a program is overlaid, channel 15 (decimal) is used by the overlay handler and should not be modified. (Other channels can be defined and used as usual.)

Macro Call: .CDFN area,addr,num

where:

area	is the address of a three-word EMT argument block
addr	is the address where the I/O channels begin
num	is the number of I/O channels to be created

Request Format:

R0 → area:

15	0
addr	
num	

Errors:

Code	Explanation
0	An attempt was made to define fewer channels than already exist.

Example:

```
.TITLE  CDFN.MAC
;THIS EXAMPLE DEFINES 40 (DECIMAL) CHANNELS TO START
;AT LOCATION CHANL. AN ERROR OCCURS IF 40 OR MORE CHANNELS
;ARE ALREADY DEFINED.
        .MCALL  .CDFN,.PRINT,.EXIT
START:  .CDFN   #R0LIST,#CHANL,#40.
        BCS     BADCDF
        .PRINT  #MSG1
        .EXIT
BADCDF: .PRINT  #MSG2
        .EXIT
MSG1:   .ASCIZ  /.CDFN O.K./
.EVEN
MSG2:   .ASCIZ  /BAD .CDFN/
        .EVEN
R0LIST: .BLKW   3               ;EMT ARGUMENT LIST
CHANL:  .BLKW   40.*5           ;ROOM FOR CHANNELS
        .END    START
```

.CHAIN

2.4.2 .CHAIN

This request allows a background program to pass control directly to another background program without operator intervention. Since this process can be repeated, a large "chain" of programs can be strung together.

The area from locations 500-507 contains the device name and file name (in Radix-50) to be chained to. The area from locations 510-777 is used to pass information between the chained programs.

Macro Call: .CHAIN

Request Format:

R0 =	10	0

Notes:

1. No assumptions should be made concerning which areas of memory remain intact across a .CHAIN. In general, only the resident monitor and locations 500-777 are preserved across a .CHAIN.

2. I/O channels are left open across a .CHAIN for use by the new program. However, new I/O channels opened with a .CDFN request are not available in this way. Since the monitor reverts to the original 16 channels during a .CHAIN, programs that leave files open across a .CHAIN should not use .CDFN. Furthermore, non-resident device handlers are released during a .CHAIN, and must be .FETCHed again by the new program.

3. An executing program can determine whether it was CHAINed to or RUN from the keyboard by examining bit 8 of the JSW. The monitor sets this bit if the program was invoked with .CHAIN. If the program was invoked with R or RUN command, this bit remains cleared. If bit 8 is set, the information in locations 500-777 is preserved from the program that issued the .CHAIN, and is available for the currently executing program to use.

   An example of a calling and a called program is MACRO and CREF. MACRO places important information in the chain area, locations 500-777, then chains to CREF. CREF tests bit 8 of the JSW. If it is clear, it means that CREF was invoked with the R or RUN command and the chain area does not contain useful information. CREF aborts itself immediately. If bit 8 is set, it means that CREF was invoked with .CHAIN and the chain area contains information placed there by MACRO. In this case, CREF executes properly.

Errors:

.CHAIN is implemented by simulating the monitor RUN command and can produce any errors that RUN can produce. If an error occurs, the .CHAIN is abandoned and the keyboard monitor is entered.

When using .CHAIN, care should be taken for initial stack placement, since the program being chained to is started. The linker normally defaults the initial stack to 1000(octal); if caution is not observed, the stack can destroy chain data before it can be used.

Example:

```
        .TITLE  CHAIN.MAC
;THIS EXAMPLE CHAINS TO THE PROGRAM CALLED MYPROG.SAV
;AND PASSES A TYPED LINE TO THE PROGRAM.
        .MCALL  .CHAIN,.TTYIN
START:  MOV     #500,R1                 ;SET UP TO CHAIN
        MOV     #CHPTR,R2               ;DEVICE, FILE NAME TO 500-511
        .REPT   4
        MOV     (R2)+,(R1)+
        .ENDR
LOOP:   .TTYIN                          ;NOW GET A COMMAND LINE
        MOVB    R0,(R1)+                ;AND PASS IT TO THE JOB
        CMPB    R0,#12                  ;IN LOCATIONS 512 AND UP
        BNE     LOOP                    ;LOOP UNTIL LINE FEED
        CLRB    (R1)+                   ;PUT IN A NULL BYTE
        .CHAIN
CHPTR:  .RAD50  /DK /
        .RAD50  /MYPROG/
        .RAD50  /SAV/
        .END    START
```

.CHCOPY

2.4.3 .CHCOPY (FB and XM Only)

The .CHCOPY request opens a channel for input, logically connecting it to a file that is currently open by the other job for either input or output. This request can be used by either the foreground or the background. .CHCOPY must be issued before the first .READ or .WRITE.

.CHCOPY is legal only on files on disk (including diskette) or DECtape; however, no errors are detected by the system if another device is used. (To close a channel following use of .CHCOPY, use either the .CLOSE or .PURGE request.)

Macro Call: .CHCOPY area,chan,ochan

where:	area	is the address of a two-word EMT argument block
	chan	is the channel the current job will use to read the data
	ochan	is the channel number of the other job's channel to be copied

Request Format:

R0 → area:

13	chan
ochan	

Notes:

1. If the other job's channel was opened with .ENTER in order to create a file, the copier's channel indicates a file that extends to the highest block that the creator of the file had written at the time the .CHCOPY was executed.

2. A channel open on a non-file-structured device should not be copied, because intermixture of buffer requests can result.

3. A program can write to a file (that is being created by the other job) on a copied channel just as it could if it were the creator. When the copier's channel is closed, however, no directory update takes place.

Errors:

Code	Explanation
0	Other job does not exist, does not have enough channels defined, or does not have the specified channel (ochan) open.
1	Channel (chan) already open.

Example:

```
.TITLE  CHCOPF.MAC
;THIS IS THE FOREGROUND PROGRAM TO BE RUN IN
;CONJUNCTION WITH CHCOPY.MAC FOR THE EXECUTION OF
;THE CHCOPY EXAMPLE.
        .MCALL  .LOOKUP,.PRINT,.SDATW,.EXIT,.RCVDW

START:  MOV     #AREA,R5
        .LOOKUP R5,#1,#FILE
        BCS     LKERR
        .SDATW  R5,#BUFR,#2         ;PASS BLOCK # AND CHANNEL#
                                    ;TO BACKGROUND JOB
        BCS     NJERR               ;NOT THERE
        .RCVDW  R5,#BUF2,#2         ;WAIT FOR RETURN MESSAGE
        .EXIT
NJERR:  MOV     #NJMSG,R0
PMSG:   .PRINT
OK:     .EXIT
LKERR:  MOV     #LKMSG,R0
        BR      PMSG
FILE:   .RAD50  /DK TEST  TMP/
AREA:   .BLKW   5
BUFR:   .WORD   0                   ;BLOCK #
        .WORD   1                   ;CHANNEL #
BUF2:   .BLKW   3
LKMSG:  .ASCIZ  /LOOKUP ERROR/
NJMSG:  .ASCIZ  /NO BACKGROUND JOB/
        .EVEN
        .END    START


.TITLE  CHCOPY.MAC
;IN THIS EXAMPLE, .CHCOPY IS USED TO READ DATA CURRENTLY
;BEING WRITTEN BY THE OTHER JOB. THE CORRECT BLOCK
;NUMBER AND CHANNEL TO READ IS OBTAINED BY A .RCVDW COMMAND.
;THE CHANNEL NUMBER WILL BE IN MSG+4. THE CHCOPF.MAC PROGRAM
;MUST BE EXECUTED IN THE FOREGROUND.
        .MCALL  .CHCOPY,.RCVDW,.PURGE,.READW,.EXIT,.PRINT
ST:     .PURGE  #0                  ;MAKE SURE WE HAVE CLEAR
                                    ;CHANNEL
        .RCVDW  #AREA,#MSG,#2       ;READ TWO WORDS, BLOCK #
                                    ;AND CHANNEL
        BCS     NOJOB               ;NO JOB THERE
        .CHCOPY #AREA,#0,MSG+4 ;CHANNEL # IS IN THERE
        BCS     BUSY                ;BUT BUSY
        .READW  #AREA,#0,#BUFF,#256.,MSG+2 ;GET THE CORRECT BLOCK
        BCS     RDERR
        .PRINT  #OKMSG
        .EXIT
NOJOB:  .PRINT  #MSG1
        .EXIT
BUSY:   .PRINT  #MSG2
        .EXIT
RDERR:  .PRINT  #MSG3
        .EXIT
AREA:   .BLKW   5
MSG:    .BLKW   5
BUFF:   .BLKW   256.
MSG1:   .ASCIZ  /NO JOB!/
MSG2:   .ASCIZ  /BUSY!/
MSG3:   .ASCIZ  /READ ERROR/
OKMSG:  .ASCIZ  /READ OK/
.EVEN
        .EXIT
        .END    ST
```

.CLOSE

2.4.4 .CLOSE

The .CLOSE request terminates activity on the specified channel and frees it for use in another operation. The handler for the associated device must be in memory.

Macro Call: .CLOSE chan

Request Format:

R0 =	6	chan

A .CLOSE request specifying a channel that is not opened is ignored.

A file opened with .LOOKUP does not require any directory operations when a .CLOSE is issued and the USR does not have to be in memory for the .CLOSE.

A .CLOSE is required on any channel opened for output if the associated file is to become permanent.

A .CLOSE performed on a file opened with .ENTER causes the device directory to be updated to make that file permanent. If the device associated with the specified channel already contains a file with the same name and file type, the old copy is deleted when the new file is made permanent. When an .ENTERed file is .CLOSEd, its permanent length reflects the highest block written since it was entered. For example, if the highest block written is block number 0, the file is given a length of 1; if the file was never written, it is given a length of 0. If this length is less than the size of the area allocated at .ENTER time, the unused blocks are reclaimed as an empty area on the device. In magtape operations, the .CLOSE request causes the handler to write an ANSI EOF1 label in software mode (using MM.SYS or MT.SYS) and to close the channel in hardware mode (using MMHD.SYS or MTHD.SYS).

Errors:

.CLOSE does not return any errors (unless the .SERR system service has been issued). If the device handler for the operation is not in memory, and the .CLOSE request requires updating of the device directory, a fatal monitor error is generated.

Example:

The examples for the .CSISPC and .WRITW requests show typical uses for .CLOSE.

.CMKT

2.4.5 .CMKT (FB and XM Only; SJ Monitor SYSGEN Option)

The .CMKT request causes one or more outstanding mark time requests to be cancelled (mark time requests are discussed in Section 2.4.28).

Macro Call: .CMKT area,id,time

where:

area	is the address of a three-word EMT argument block
id	is a number used to identify each mark time request to be cancelled. If more than one mark time request has the same id, the request with the earliest expiration time is cancelled. If id = 0, all nonsystem mark time requests (that is, those in the range 1-177377) for the issuing job are cancelled.
time	is the pointer to a two-word area in which the monitor returns the amount of time remaining in the cancelled request. The first word contains the high-order time, the second contains the low-order. If an address of 0 is specified, no value is returned. If id = 0, the time parameter is ignored and need not be indicated.

Request Format:

R0 → area:

23	0
id	
time	

Notes:

1. Cancelling a mark time request frees the associated queue element for other uses.

2. A mark time request can be converted into a timed wait by issuing a .CMKT followed by a .TWAIT, and specifying the same time area.

3. If the mark time request to be cancelled has already expired and is waiting in the job's completion queue, .CMKT returns an error code of 0. It does not remove the expired request from the completion queue. The completion routine will eventually be run.

Errors:

Code	Explanation
0	The id was not zero; a mark time request with the specified identification number could not be found (implying that the request was never issued or that it has already expired).

Example:

See the example following the description of the .MRKT request.

.CNTXSW

2.4.6 .CNTXSW (FB and XM Only)

A context switch is an operation performed when a transition is made from running one job to running the other. The .CNTXSW request is used to specify locations to be included in the switching of jobs between background and foreground.

The system always saves the parameters it needs to uniquely identify and execute a job. These parameters include all registers and the following locations:

34,36	Vector for TRAP instruction
40-52	System communication area

If an .SFPA request has been executed with a non-zero address, all floating point registers and the floating point status are also saved.

It is possible that both jobs want to share the use of a particular location that is not included in normal context switch operations. For example, if a program uses the IOT instruction to perform an internal user function (such as printing error messages), the program must set up the vector at 20 and 22 to point to an internal IOT trap handling routine. If both foreground and background wish to use IOT, the IOT vector must always point to the proper location for the job that is executing. Including locations 20 and 22 in the .CNTXSW list for both jobs accomplishes this. In the XM monitor, both IOT and BPT vectors are automatically context switched. The procedure described above is not necessary for jobs running under the XM monitor.

If .CNTXSW is issued more than once, only the latest list is used; the previous address list is discarded. Thus, all addresses to be switched must be included in one list. If the address (addr) is 0, no extra locations are switched. The list cannot be in an area into which the USR swaps, nor can it be modified while a job is running.

In the XM monitor, the .CNTXSW request is ignored for virtual jobs, since they do not share memory with other jobs. The IOT, BPT, and TRAP vectors are simulated for virtual jobs by the monitor. The virtual job sets up the vector in its own virtual space by any of the usual methods (such as a direct move or an .ASECT). When the monitor receives a synchronous trap from a virtual job that was caused by an IOT, BPT, or TRAP instruction, it checks for a valid trap vector and dispatches the trap to the user program in user mapping mode. An invalid trap vector address will abort the job with the following fatal error message:

?MON-F-ILL SST (illegal synchronous system trap)

Macro Call: .CNTXSW area,addr

where: area — is the address of a 2-word EMT argument block

addr — is a pointer to a list of addresses terminated by a zero word. The addresses in the list must be even and:

a. in the range 2-476, or
b. in the user job area, or
c. in the I/O page (addresses 160000-177776).

Request Format:

R0→area:

33	0
addr	

Errors:

Code	Explanation
0	One or more of the above conditions was violated.

Example:

```
        .TITLE  CNTXSW.MAC
;IN THIS EXAMPLE, .CNTXSW REQUEST IS USED TO SPECIFY THAT LOCATION 20
;AND 22 (IOT VECTORS) AND CERTAIN NECESSARY EAE REGISTERS BE CONTEXT
;SWITCHED. THIS ALLOWS BOTH JOBS TO USE IOT AND THE EAE SIMULTANEOUSLY
;YET INDEPENDENTLY.
        .MCALL  .CNTXSW,.PRINT,.EXIT
START:  MOV     #LIST,R0        ;SET R0 TO OUR OWN LIST
        .CNTXSW ,#SWAPLS        ;THE LIST OF ADDRS IS
                                ;AT SWAPLS.
        BCC     1$
        .PRINT  #ADDERR         ;ADDRESS ERROR(SHOULD NOT OCCUR)
        .EXIT
1$:     .PRINT  #CNTOK
        .EXIT
SWAPLS: .WORD   20              ;ADDRESSES TO INCLUDE IN LIST
        .WORD   22
        .WORD   177302
        .WORD   177304
        .WORD   177310
        .WORD   0
LIST:   .BYTE   0,33            ;FUNCTION CODE WORD
        .WORD   0               ;THE MACRO FILLS THIS ONE.
ADDERR: .ASCIZ  /ADDRESSING ERROR/
        .EVEN
CNTOK:  .ASCIZ  /CONTEXT SWITCH O.K./
        .EVEN
        .END    START
```

.CSIGEN

2.4.7 .CSIGEN

The .CSIGEN request calls the Command String Interpreter (CSI) in general mode to process a standard RT-11 command string. In general mode, all file .LOOKUPs and .ENTERs as well as handler .FETCHs are performed. This request gets the program's user command string (dev:output-filespec=dev:input-filespec/options...) into the program, and the following operations occur:

1. The devices specified in the command line are fetched.

2. .LOOKUP and/or .ENTER requests on the files are performed.

3. The option information is placed on the stack.

When called in general mode, the CSI closes channels 0-10 (octal).

.CSIGEN loads all necessary handlers and opens the files as specified. The area specified for the device handlers must be large enough to hold all the necessary handlers simultaneously. If the device handlers exceed the area available, the user program can be destroyed. (The system, however, is protected.)

When control returns to the user program after a call to .CSIGEN, register R0 points to the first available location above the handlers, the stack contains the option information, and all the specified files have been opened for input and/or output. The association is as follows: the three possible output files are assigned to channels 0, 1, and 2(octal); the six input slots are assigned to channels 3 through 10(octal). A null specification causes the associated channel to remain inactive. For example, consider the following string:

```
*,LP:=F1,F2
```

Channel 0 is inactive since the first slot is null. Channel 1 is associated with the line printer, and channel 2 is inactive. Channels 3 and 4 are associated with two files on DK:, while channels 5 through 10 are inactive. The user program can determine whether a channel is inactive by issuing a .WAIT request on the associated channel, which returns an error if the channel is not open.

Options and their associated values are returned on the stack. The first word of the stack contains the number of options. See Section 2.4.8.1 for a description of the way option information is passed.

Macro Call: .CSIGEN devspc,defext,cstrng[,linbuf]

where:

devspc — is the address of the memory area where the device handlers (if any) are to be loaded.

defext — is the address of a four-word block that contains the Radix-50 default file types. These file types are used when a file is specified without a file type.

cstrng — is the address of the ASCIZ input string or a #0 if input is to come from the console terminal. (In an FB environment only, if the input is from the console terminal, an .UNLOCK of the USR is automatically performed, even if the USR is locked at the time.) If the string is in memory, it must not contain a <RET><LF> (octal 15 and 12), but must terminate with a zero byte. If the cstring field is left blank, input is automatically taken from the console terminal. This string, whether in memory or entered at the console, must obey all the rules for a standard RT-11 command string.

linbuf — is the address where the original command string is to be stored. This is a user-supplied area 81 decimal bytes in length. The command string is stored in this area and is terminated with a zero byte instead of <RET> <LF> (octal 15 and 12).

Notes:

The four-word block pointed to by defext is arranged as:

Word 1:	default file type for all input channels
Words 2,3,and 4:	default file types for output channels 0,1,2, respectively

If there is no default for a particular channel, the associated word must contain 0. All file types are expressed in Radix-50. For example, the following block can be used to set up default file types for a macro assembler:

```
DEFEXT:  .RAD50   "MAC"
         .RAD50   "OBJ"
         .RAD50   "LST"
         .WORD    0
```

In the command string:

```
*DT0:ALPHA,DT1:BETA=DT2:INPUT
```

the default file type for input is MAC; for output, OBJ and LST. The following cases are legal:

```
*DT0:OUTPUT=
*DT2:INPUT
```

In other words, the equal sign is not necessary if only input files are specified.

An optional argument (linbuf) is available in the .CSIGEN format that provides the user with an area to receive the original input string. The input string is returned as an ASCIZ string and can be printed through a .PRINT request.

.CSIGEN automatically takes its input line from an indirect command file if console terminal input is specified (cstring = #0) and the program issuing the .CSIGEN is invoked through an indirect command file.

Errors:

If CSI errors occur and input was from the console terminal, an error message describing the fault is printed on the terminal and the CSI retries the command. If the input was from a string, the carry bit is set and byte 52 contains the error code. The options and option-count are purged from the stack. The errors are:

Code	Explanation
0	Illegal command (bad separators, illegal file name, command too long, etc.).
1	A device specified is not found in the system tables.
2	Unused.
3	An attempt to .ENTER a file failed because of a full directory.
4	An input file was not found in a .LOOKUP.

Example:

```
.TITLE  CSIGEN.MAC
;THIS EXAMPLE USES THE GENERAL MODE OF THE CSI IN A PROGRAM
;TO COPY AN INPUT FILE TO AN OUTPUT FILE. COMMAND INPUT TO THE CSI
;IS FROM THE CONSOLE TERMINAL.
        .MCALL  .CSIGEN,.READW,.PRINT,.EXIT,.WRITW,.CLOSE,.SRESET
ERRWD=52

START:  .CSIGEN #DSPACE,#DEXT   ;GET STRING FROM TERMINAL
        MOV     R0,BUFF         ;R0 HAS FIRST FREE LOCATION
        CLR     INBLK           ;INPUT BLOCK #
        MOV     #LIST,R5        ;EMT ARGUMENT LIST
READ:   .READW  R5,#3,BUFF,#256.,INBLK ;READ CHANNEL 3
        BCC     2$              ;NO ERRORS
        TSTB    @#ERRWD         ;EOF ERROR?
        BEQ     EOF             ;YES
        MOV     #INERR,R0
1$:     .PRINT                  ;ERROR MESSAGE
        CLR     R0              ;HARD EXIT
        .EXIT
2$:     .WRITW  R5,#0,BUFF,#256.,INBLK ;WRITE THE BLOCK
        BCC     NOERR           ;NO ERROR WRITING
        MOV     #WTERR,R0
        BR      1$              ;HARD OUTPUT ERROR
NOERR:  INC     INBLK           ;GET NEXT BLOCK
        BR      READ            ;LOOP UNTIL DONE
EOF:    .CLOSE  #0              ;CLOSE OUTPUT CHANNEL
        .CLOSE  #3              ;AND INPUT CHANNEL
        .SRESET                 ;RELEASE HANDLER FROM MEMORY
        .EXIT                   ;EXIT
DEXT:   .WORD   0,0,0,0         ;NO DEFAULT EXTENSIONS
BUFF:   .WORD   0               ;I/O BUFFER START
INBLK:  .WORD   0               ;RELATIVE BLOCK TO READ/WRITE
LIST:   .BLKW   5               ;EMT ARGUMENT LIST
INERR:  .ASCIZ  /INPUT ERROR/
        .EVEN
WTERR:  .ASCIZ  /OUTPUT ERROR/
        .EVEN
DSPACE=.                        ;HANDLER SPACE
        .END    START
```

.CSISPC

2.4.8 .CSISPC

The .CSISPC request calls the Command String Interpreter in special mode to parse the command string and return file descriptors and options to the program. In this mode, the CSI does not perform any handler fetches, .CLOSEs, .ENTERs or .LOOKUPs.

Options and their associated values are returned on the stack. However, the optional argument (linbuf) provides the user program with the original command string.

.CSISPC automatically takes its input line from an indirect command file if console terminal input is specified (cstrng = #0) and the program issuing the .CSISPC is invoked through an indirect command file.

Macro Call: .CSISPC outspc,defext,cstrng[,linbuf]

where:

outspc — is the address of the 39-word block to contain the file descriptors produced by .CSISPC. This area can overlay the space allocated to cstring, if desired.

defext — is the address of a four-word block that contains the Radix-50 default file types. These file types are used when a file is specified without a file type.

cstrng — is the address of the ASCIZ input string or a #0 if input is to come from the console terminal. If the string is in memory, it must not contain a <RET><LF> (octal 15 and 12), but must terminate with a zero byte. If cstrng is blank, input is automatically taken from the console terminal.

linbuf — is the address where the original command string is to be stored. This is a user-specified area 81 decimal bytes in length. The command string is stored in this area and is terminated with a zero byte instead of <RET> <LF> (octal 15 and 12).

Notes:

The 39-word file description consists of nine file descriptor blocks (five words for each of three possible output files; four words for each of six possible input files), which correspond to the nine possible files (three output, six input). If any of the nine possible file names are not specified, the corresponding descriptor block is filled with 0s.

The five-word blocks hold Radix-50 four words representing dev:file.type, and one word representing the size specification given in the string. (A size specification is a decimal number enclosed in square brackets [], and following the output file descriptor.) For example:

```
*DT3:LIST.MAC[15]=PC:
```

Using special mode, the CSI returns in the first five-word slot:

```
16101     Radix-50 for DT3
46173     Radix-50 for LIS
76400     Radix-50 for T
50553     Radix-50 for MAC
00017     Octal value of size request
```

In the fourth slot (starting at an offset of 36 octal bytes into outspc), the CSI returns:

```
62170     Radix-50 for PC
0         No file name
0         Specified
0         No file type given
```

Since this is an input file, only four words are returned.

Errors:

Errors are the same as in general mode except that illegal device specifications are checked only for output file specifications with null file names. Since .LOOKUPs and .ENTERs are not done, the valid error codes are:

Code	Explanation
0	Illegal command line
1	Illegal device

Example:

```
.TITLE  CSISPC.MAC
;THIS EXAMPLE ILLUSTRATES THE USE OF THE SPECIAL MODE OF CSI.
;THIS EXAMPLE COULD BE A PROGRAM TO READ A FILE WHICH IS NOT IN
;RT-11 FORMAT TO A FILE UNDER RT-11.
        .MCALL  .CSISPC,.PRINT,.EXIT,.ENTER,.CLOSE

START:  .CSISPC #OUTSPC,#DEXT,#CSTRNG ;GET INPUT FROM A
                                      ;STRING IN MEMORY
        BCC     2$
        MOV     #SYNERR,R0          ;SYNTAX ERROR
1$:     .PRINT                      ;ERROR MESSAGE
        .EXIT
2$:     .ENTER  #LIST,#0,#OUTSPC,#64. ;ENTER FILE UNDER RT-11
        BCC     3$
        MOV     #ENMSG,R0           ;ENTER FAILED
        BR      1$
3$:     JSR     R5,INPUT            ;ROUTINE INPUT WILL USE
                                    ;THE INFORMATION AT
                                    ;#OUTSPC+36 TO READ INPUT
                                    ;FROM THE NON-RT11 DEVICE.
                                    ;INPUT IS PROCESSED AND
                                    ;WRITTEN VIA .WRITW REQUFSTS
        .CLOSE  #0                  ;MAKE OUTPUT FILE PERMANENT.
        .EXIT                       ;AND EXIT PROGRAM
CSTRNG: .ASCIZ  "DT4:RTFIL.MAC=DT2:DOS.MAC"
        .EVEN
DEXT:   .WORD   0,0,0,0             ;NO DEFAULT EXTENSIONS
LIST:   .BLKW   5                   ;LIST FOR EMT CALLS
SYNERR: .ASCIZ  "CSI ERROR"
ENMSG:  .ASCIZ  "ENTER FAILED"
        .EVEN
INPUT:  RTS     R5
OUTSPC=.                            ;CSI LIST GOES HERE
        .END    START
```

2.4.8.1 **Passing Option Information** - In both general and special modes of the CSI, options and their associated values are returned on the stack. A CSI option is a slash (/) followed by any character. The CSI does not restrict the option to printing characters, although it is suggested that printing characters be used wherever possible. The option can be followed by an optional value, which is indicated by a : separator. The : separator is followed by either an octal number, a decimal number or by one to three alphanumeric characters, the first of which must be alphabetic. Decimal values are indicated by terminating the number with a decimal point (/N:14.). If no decimal point is present, the number is assumed to be octal. Options can be associated with files with the CSI. For example:

```
*DK:FOO/A,DT4:FILE.OBJ/A:100
```

In this case, there are two A options. The first is associated with the input file DK:FOO. The second is associated with the input file DT4:FILE.OBJ, and has a value of 100(octal). The format of the stack output of the CSI for options is as follows:

Word #	Value	Meaning
1 (top of stack)	N	Number of options found in command string. If N=0, no options were found.
2	Option value and file number	Even byte = seven-bit ASCII option value. Bits 8-14 = Number (0-10) of the file with which the option is associated. Bit 15 = 1 if the option had a value. = 0 if the option had no value.
3	Option value or next option	If bit 15 of word 2 is set, word 3 contains the option value. If bit 15 is not set, word 3 contains the next option value.

For example, if the input to the CSI is:

```
*FILE/B:20.,FIL2/E=DT3:INPUT/X:SY:20
```

on return, the stack is:

Stack Pointer->	4	Three options appeared (X option has two values and is treated as two options).
	101530	Last option=X; with file 3, has a value.
	20	Value of option X=20 (octal)
	101530	Next option =X; with file 3, has a value.
	075250	Next value of option X=RAD50 code for SY:.
	505	Next option=E; associated with file 1, no value.
	100102	Option=B; associated with file 0 and has a value of 20 (decimal) or 24
	24	(octal).

As an extended example, assume the following string was input for the CSI in general mode:

```
*FILE[8.],LP:,SY:FILE2[20.]=PC:,DT1:IN1/B,DT2:IN2/M:7
```

Assume also that the default file type block is:

```
DEFEXT:  .RAD50   'MAC'    ;INPUT FILE TYPE
         .RAD50   'OP1'    ;FIRST OUTPUT FILE TYPE
         .RAD50   'OP2'    ;SECOND OUTPUT FILE TYPE
         .RAD50   'OP3'    ;THIRD OUTPUT FILE TYPE
```

The result of this CSI call are:

1. An eight-block file named FILE.OP1 is entered on channel 0 on device DK:; channel 1 is open for output to the device LP:; a 28-block (to show that it is a decimal number) file named FILE2.OP3 is entered on the system device on channel 2.

2. Channel 3 is open for input from paper tape; channel 4 is open for input from a file IN1.MAC on device DT1:; channel 5 is open for input from IN2.MAC on device DT2:.

3. The stack contains options and values as follows:

Contents	Explanation
2	Two options found in string.
102515	Second option is M, associated with Channel 5; has a value.
7	Numeric value is 7 (octal).
2102	Option is B, associated with Channel 4; has no value.

If the CSI were called in special mode, the stack would be the same as for the general mode call, and the descriptor table would contain:

```
OUTSPC:  15270      ;.RAD50   'DK'
         23364      ;.RAD50   'FIL'
         17500      ;.RAD50   'E'
         60137      ;.RAD50   'OP1'
            10      ;LENGTH OF 8 BLOCKS (DECIMAL)
         46600      ;.RAD50   'LP'
             0      ;NO NAME OR LENGTH SPECIFIED
             0
             0
             0
         75250      ;.RAD50   'SY'
         23364      ;.RAD50   'FIL'
         22100      ;.RAD50   'E2'
         60141      ;.RAD50   'OP3'
            24      ;LENGTH OF 20 (DECIMAL)
         62170      ;.RAD50   'PC'
             0
             0
             0
         16077      ;.RAD50   'DT1'
         35217      ;.RAD50   'IN1'
             0      ;.RAD50   '   '
         50553      ;.RAD50   'MAC'
         16100      ;.RAD50   'DT2'
         35220      ;.RAD50   'IN2'
             0      ;.RAD50   '   '
         50553      ;.RAD50   'MAC'
             0
             .
             .
             .
             0       (12 more zero words
                              are returned)
```

Keyboard error messages that can occur from incorrect use of the CSI when input is from the console keyboard include:

Message	Meaning
?CSI-F-Illegal command	Syntax error.
?CSI-F-File not found	Input file was not found.
?CSI-F-Device full	Output file does not fit.
?CSI-F-Illegal device	Device specified does not exist.

Notes:

1. In many cases, the user program does not need to process options in CSI calls. However, the user at the console can inadvertently enter options. In this case, it is wise for the program to save the value of the stack pointer before the call to the CSI, and restore it after the call. In this way, no extraneous values are left on the stack. Note that even a command string with no options causes a word to be pushed onto the stack.

2. In the FB monitor, calls to the CSI that require console terminal input always do an implicit .UNLOCK of the USR. This should be kept in mind when using .LOCK calls.

.CSTAT

2.4.9 .CSTAT (FB and XM Only)

This request furnishes the user with information about a channel. It is supported only in the FB and XM environments; no information is returned by the SJ monitor.

Macro Call: .CSTAT area,chan,addr

where:

- area — is the address of a two-word EMT argument block
- chan — is the number of the channel about which information is desired
- addr — is the address of a six-word block to contain the status

Request Format:

R0→area:

27	chan
addr	

Notes:

The six words passed back to the user are:

1. Channel status word (bit 0 set = hard error;
   bit 13 set = end of file)

2. Starting block number of file (0 if sequential-access device or if channel was opened with a non-file-structured .LOOKUP or .ENTER)

3. Length of file (no information if non-file-structured device or if channel was opened with a non-file-structured .LOOKUP or .ENTER)

4. Highest relative block written since file was opened (no information if non-file-structured device)

5. Unit number of device with which this channel is associated

6. Radix-50 of the device name with which the channel is associated (this is a physical device name, unaffected by any user name assignment in effect)

The fourth word (highest block) is maintained by the .WRITE/.WRITC/.WRITW requests. If data is being written on this channel, the highest relative block number is kept in this word.

Errors:

Code	Explanation
0	The channel is not open.

Example:

```
.TITLE  CSTAT.MAC
;IN THIS EXAMPLE, .CSTAT IS USED TO DETERMINE THE .RAD50
;REPRESENTATION OF THE DEVICE WITH WHICH THE CHANNEL IS ASSOCIATED.
        .MCALL  .CSTAT,.CSIGEN,.PRINT,.EXIT
ST:     .CSIGEN #DEVSDC,#DEFEXT ;OPEN FILES
        .CSTAT  #AREA,#0,#ADDR  ;GET THE STATUS
        BCS     NOCHAN          ;CHANNEL 0 NOT OPEN
        MOV     #ADDR+10,R5     ;POINT TO UNIT #
        MOV     (R5)+,R0        ;UNIT # TO R0
        ADD     (PC)+,R0        ;MAKE IT RAD50
        .RAD50  /  0/
        ADD     (R5),R0         ;GET DEVICE NAME
        MOV     R0,DEVNAM       ;DEVNAM HAS RAD50 DEVICE NAME
        .EXIT
AREA:   .BLKW   5               ;EMT ARG LIST
ADDR:   .BLKW   6               ;AREA FOR CHANNEL STATUS
DEVNAM: .WORD   0               ;STORAGE FOR DEVICE NAME
DEFEXT: .WORD   0,0,0,0
NOCHAN: .PRINT  #MSG
        .EXIT
MSG:    .ASCIZ  /NO OUTPUT FILE/
        .EVEN
DEVSDC=.
        .END    ST
```

.DATE

2.4.10 .DATE

This request returns the current date information from the system date word in R0. The date word returned is in the following format:

Bit: $\underbrace{13 \ldots 10}_{\text{MONTH}}\ \underbrace{9 \ldots 5}_{\text{DAY}}\ \underbrace{4 \ldots 0}_{\text{YEAR}}$

The year value in bits 4-0 is the actual year minus 72.

NOTE

RT-11 does not support month and year roll-over. The keyboard monitor DATE command must be issued to change the month and year appropriately.

Macro Call: .DATE

Request Format:

R0 =	12	0

Errors:

No errors are returned. A zero result in R0 indicates that the user has not entered a date.

Example:

This example is a subroutine that can be assembled separately and linked with a user's program.

```
.TITLE  DATE.MAC
;
; CALLING SEQUENCE:
;
;       JSR     PC,DATE
;
; INPUT: NONE
;
; OUTPUT: R0 = DAY (1-31)
;         R1 = MONTH (1-12)
;         R2 = YEAR -72
;
; ERROR: CARRY SET INDICATES NO DATE SPECIFIED
;
        .MCALL  .DATE
DATE::
        .DATE                           ;GET THE SYSTEM DATE
        MOV     R0,R2                   ;COPY THE DATE
        BEQ     10$                     ;BRANCH IF NO DATE
        BIC     #^C37,R2                ;ISOLATE THE YEAR
        ASR     R0                      ;PUT THE MONTH ON A BYTE BOUNDARY
        ASR     R0                      ;
        MOV     R0,R1                   ;COPY THE DATE
        SWAB    R1                      ;PUT THE MONTH IN THE LOW BYTE
        BIC     #^C37,R1                ;ISOLATE THE MONTH
        ASR     R0                      ;SHIFT THE DAY TO THE BYTE BOUNDARY
        ASR     R0                      ;
        ASR     R0                      ;
        BIC     #^C37,R0                ;ISOLATE THE DAY
        CLC                             ;INDICATE NO ERROR
        BR      20$                     ;RETURN
10$:    SEC                             ;NO DATE.  INDICATE ERROR
20$:    RTS     PC

        .END
```

.DELETE

2.4.11 .DELETE

The .DELETE request deletes a named file from an indicated device. This request generates a monitor error if a hard I/O error is detected during directory I/O. The .SERR programmed request can be used to allow the program to process the error. .DELETE is illegal for magtapes.

Macro Call: .DELETE area,chan,dblk,seqnum

where:

area	is the address of a three-word EMT argument block.
chan	is the device channel number in the range 0-377 (octal)
dblk	is the pointer to the address of a four-word Radix-50 descriptor of the file to be deleted.
seqnum	file number for cassette operations: if this argument is blank, a value of 0 is assumed.

Request Format:

R0 → area:		
	0	chan
	dblk	
	seqnum	

Note:

The channel specified in the .DELETE request must not be open when the request is made, or an error will occur. The file is deleted from the device, and an empty (UNUSED) entry of the same size is put in its place. A .DELETE issued to a non-file-structured device is ignored. .DELETE requires that the handler to be used be in memory at the time the request is made. When the .DELETE is complete, the specified channel is left inactive.

Errors:

Code	Explanation
0	Channel is active
1	File was not found in the device directory
2	Illegal operation

Example:

```
        .TITLE  DELETE.MAC
;THIS EXAMPLE USES THE SPECIAL MODE OF CSI TO DELETE FILES.
;INSPEC IS THE ADDRESS OF THE FIRST INPUT SLOT IN THE CSI
;INPUT TABLE.
        .MCALL  .SRESET,.CSISPC,.DELETE,.PRINT,.EXIT
START:  .SRESET                 ;MAKE SURE CHANNELS
                                ;ARE FREE
        .CSISPC #OUTSPC,#DEFEXT ;GET COMMAND LINE
                                ;TERMINAL DIALOG WAS
                                ;DT:FILE
        .DELETE #LIST,#0,#INSPC ;USE CHANNEL 0 TO
                                ;DELETE THE FILE
                                ;WHICH IS AT THE
                                ;FIRST INPUT SLOT.
        BCC     1$              ;OK, LOOP AGAIN
        .PRINT  #NOFILE         ;NO SUCH FILE
1$:     .EXIT                   ;EXIT
NOFILE: .ASCIZ  /FILE NOT FOUND/
        .EVEN
DEFEXT: .RAD50  /MAC/           ;.MAC INPUT EXTENSION
        .WORD   0,0,0           ;NO OUTPUT DEFAULTS
LIST:   .BLKW   2               ;EMT ARG LIST
OUTSPC=.
INSPC=.+36
        .BLKW   39.
        .END    START
```

.DEVICE

2.4.12 .DEVICE (FB and XM Only)

This request allows the user to set up a list of addresses to be loaded with specified values when a program is terminated. Upon an .EXIT or CTRL/C, this list is picked up by the system and the appropriate addresses are filled with the corresponding values. This function is primarily designed to allow user programs to load device registers with necessary values. In particular, it is used to turn off a device's interrupt enable bit when the program servicing the device terminates. Successive calls to .DEVICE are allowed when the user needs to link requested tables. When the job is terminated for any reason, the list is scanned once. At that point, the monitor disables the feature until another .DEVICE call is executed. Thus, background programs that are reenterable should include .DEVICE as a part of the reenter code.

The .DEVICE request is ignored when it is issued by a virtual job running under the XM monitor.

Macro Call: .DEVICE area,addr[,link]

where:

	area	is the address of a two-word EMT argument block.
	addr	is the address of a list of two-word elements, each composed of a one-word address and a one-word value to be put at that address.

link — is the optional argument, L, that allows linking of tables on successive calls to .DEVICE. If the argument is omitted, the list referenced in the previous .DEVICE request is replaced by the new list. The argument must be supplied to cause linking of lists; however, linked and unlinked list types cannot be mixed.

Request format:

Non-linking

R0 → area:

14	0
addr	

Linking

R0 → area:

14	1
addr	

NOTE

The list referenced by addr must be either in linking format or non-linking format. The different formats are shown below. Both formats must be terminated with a separate, zero-value word. Linking format must also have a zero-value word as its first word.

Non-linking

addr →

address
value
address
value
⋮
address
value
0

Linking

addr →

0
address
value
address
value
⋮
address
value
0

Errors:

None.

Example:

```
.TITLE  DEVICE.MAC
;THE FOLLOWING EXAMPLE SHOWS .DEVICE IS USED TO DISABLE
;INTERRUPTS FROM THE AFC11 (A-D CONVERTER
;SUB-SYSTEM).
        .MCALL  .DEVICE,.EXIT

START:  .DEVICE #LIST
        .EXIT
LIST:   .BYTE   0,14            ;EMT ARG LIST
        .WORD   ATOD
ATOD:   172570                  ;ADDRESS IS 172570
        0                       ;JAM A 0 INTO IT
        0                       ;THIS 0 TERMINATES THE LIST.
        .END    START
```

.DSTATUS

2.4.13 .DSTATUS

This request is used to obtain information about a particular device.

Macro Call: .DSTATUS retspc,dnam

where: retspc — is the four-word space used to store the status information.

dnam — is the pointer to the Radix-50 device name.

.DSTATUS looks for the device specified by dnam and, if found, returns four words of status starting at the address specified by retspc. The four words returned are:

1. Status Word

Bits 7-0: contain a number that identifies the device in question. The values (octal) currently defined are:

0 =	RK05 Disk
1 =	TC11 DECtape
2 =	Reserved
3 =	Line Printer
4 =	Console Terminal or Batch Handler
5 =	RL01 Disk
6 =	RX02 Diskette
7 =	PC11 High-speed paper tape reader and punch
10 =	Reserved
11 =	Magtape (TM11, TMA11)
12 =	RF11 Disk
13 =	TA11 Cassette
14 =	Card Reader (CR11,CM11)
15 =	Reserved
16 =	RJ03/4 Fixed-head Disks
17 =	Reserved
20 =	TJU16 Magtape
21 =	RP02 Disk
22 =	RX01 Diskette
23 =	RK06/07 Disk
24 =	Error Log Handler
25 =	Null Handler
26-30 =	Reserved (NETWORKS)
31-33 =	Reserved (DIBOL LP,LQ,LR,LS)

Bit 15: 1= Random-access device (disk, DECtape)
0= Sequential-access device (line printer, paper tape, card reader, magtape, cassette, terminal)

Bit 14: 1= Read-only device (card reader, paper tape reader)

Bit 13: 1= Write-only device (line printer, paper tape punch)

Bit 12: 1= Non RT-11 directory-structured device (magtape, cassette)

Bit 11:	1=	Enter handler abort entry every time a job is aborted.
	0=	Handler abort entry taken only if there is an active queue element belonging to aborted job.
Bit 10:	1=	Handler accepts .SPFUN requests (for example, MT, CT, DX).
	0=	.SPFUN requests are rejected as illegal.

2. Handler size.

   The size of the device handler, in bytes.

3. Load address +6.

   Non-zero implies the handler is now in memory; zero implies it must be .FETCHed before it can be used. The address of the handler is the load address +6.

4. Device size.

   The size of the device (in 256-word blocks) for block-replaceable devices; 0 for sequential-access devices. The last block on the device is the device size minus 1.

The device name can be a user-assigned name. DSTATUS information is extracted from block 0 of the device handler. Therefore, this request requires the handler file for the device to be present on the system volume, unless the device is the system device. The system device handler is always memory resident.

Errors:

Code	Explanation
0	Device not found in tables.

Example:

```
.TITLE  DSTATU.MAC
;THIS EXAMPLE SHOWS HOW TO DETERMINE IF A PARTICULAR DEVICE HANDLER
;IS IN MEMORY AND, IF IT IS NOT, HOW TO FETCH IT THERE.
        .MCALL  .DSTATUS,.PRINT,.EXIT,.FETCH

START:  .DSTATUS #CORE,#FPTR    ;GET STATUS OF DEVICE
        BCC     1$
        .PRINT  #ILLDEV         ;DEVICE NOT IN TABLES
        .EXIT
1$:     TST     CORE+4          ;IS DEVICE RESIDENT?
        BNE     2$
        .FETCH  #HNDLR,#FPTR    ;NO, GET IT
        BCC     2$
        .PRINT  #FEFAIL         ;FETCH FAILED
        .EXIT
2$:     .PRINT  #FECHOK
        .EXIT
CORE:   .BLKW   4       ;DSATUS GOES HERE
FPTR:   .RAD50  /DT0/   ;DEVICE NAME
        .RAD50  /FILE  MAC/ ;FILE NAME
FEFAIL: .ASCIZ  /FETCH FAILED/
ILLDEV: .ASCIZ  /ILLEGAL DEVICE/
        .EVEN
FECHOK: .ASCIZ  /FETCH O.K./
        .EVEN
HNDLR=.                 ;HANDLER WILL GO HERE
        .END    START
```

.ENTER

2.4.14 .ENTER

The .ENTER request allocates space on the specified device and creates a tentative entry for the named file. The channel number specified is associated with the file. (Note that if the program is overlaid, channel 15 is used by the overlay handler and should not be modified.)

Macro Call: .ENTER area,chan,dblk,len,seqnum

where:

area
: is the address of a four-word EMT argument block

chan
: a channel number in the range 0-377 (octal)

dblk
: the address of a four-word Radix-50 descriptor of the file to be operated upon

len
: is the file size specification. If the argument is left blank, it is not set to 0 in area. The #0 must be specified to accomplish this. If an argument is left blank, the corresponding location in area is assumed to be set.

  The value of this argument determines the file length allocation as follows:

  0 - either half the largest empty entry or the entire second-largest empty entry, whichever is larger. (A maximum size for non-specific .ENTERs can be patched in the monitor by changing RMON offset 314).

  m - a file of m blocks. The size, m, can exceed the maximum mentioned above.

  -1 - the largest empty entry on the device.

seqnum
: file number for cassette. If this argument is blank, a value of 0 is assumed.

  For magtape it describes a file sequence number that can have the following values:

  0 - means rewind the magtape and space forward until the file name is found or until logical end-of-tape is detected. If file name is found, delete it and continue tape search.

  n - means position magtape at file sequence number n. If the file represented by the file sequence number is greater than two files away from beginning of tape, then a rewind is performed. If not, the tape is backspaced to the beginning of the file.

-1 - means space to the logical end-of-tape and enter file.

-2 - means rewind the magtape and space forward until the file name is found, or until logical end-of-tape is detected. The magtape is now positioned correctly. A new logical end-of-tape is implied.

Request Format:

R0 → area:	2	chan
	dblk	
	len	
	seqnum	

The file created with an .ENTER is not a permanent file until the .CLOSE on that channel is given. Thus, the newly created file is not available to .LOOKUP and the channel cannot be used by .SAVESTATUS requests. However, it is possible to go back and read data that has just been written into the file by referencing the appropriate block number. When the .CLOSE to the channel is given, any already existing permanent file of the same name on the same device is deleted and the new file becomes permanent. Although space is allocated to a file during the .ENTER operation, the actual length of the file is determined when .CLOSE is requested.

Each job can have up to 256 files open on the system at any time. If required, all 256 can be opened for output with the .ENTER function. .ENTER requires that the device handler be in memory when the request is made. Thus, a .FETCH should normally be executed before an .ENTER can be done. On return, R0 contains the size of the area actually allocated for use.

Notes:

When using the zero-length feature of .ENTER, it must be kept in mind that the space allocated is less than the largest empty space. This can have an important effect in transferring files between devices (particularly DECtape and diskette) that have a relatively small capacity. For example, to transfer a 200-block file to a DECtape on which the largest available empty space is 300 blocks, a zero-length transfer does not work. Since the .ENTER allocates half the largest space, only 150 blocks are really allocated and an output error occurs during the transfer. However, when transferring from A to B and the length is unknown on A, do a .LOOKUP first. This request returns the length and this value can be used to do a fixed-length .ENTER. If a specific length of 200 is requested, however, the transfer proceeds without error. The .ENTER request also generates hard errors when problems are encountered during directory operations. These errors can be detected after the operation with the .SERR request.

Errors:

Code	Explanation
0	Channel is in use.
1	In a fixed length request, no space greater than or equal to m was found, or in a non-specific request, the device or the directory was found to be full.

Example:

```
        .TITLE  ENTER.MAC
        ;.ENTER MAY BE USED TO OPEN A FILE ON A SPECIFIED DEVICE, AND
        ;THEN WRITE DATA FROM MEMORY INTO THAT FILE AS FOLLOWS:
                .MCALL  .ENTER,.WRITW,.CLOSE,.PRINT
                .MCALL  .SRESET,.EXIT,.FETCH,.SETTOP

START:  .SRESET                         ;MAKE SURE ALL CHANNELS
                                        ;ARE CLOSED.
        .SETTOP #-2                     ;ASK FOR ALL AVAILABLE MEMORY
        .FETCH  LIMIT+2,#FPRT           ;FETCH DEVICE HANDLER
        BCS     BADFET                  ;.FETCH ERROR, PROBABLY
                                        ;ILLEGAL DEVICE.
        .ENTER  #AREA,#0,#FPRT,#0 ;OPEN A FILE ON THE DEVICE
                                        ;SPECIFIED, LENGTH 0 WILL
                                        ;GIVE 1/2 OF LARGEST EMPTY
                                        ;SPACE NOW AVAILABLE.
        BCS     BADENT                  ;FAILED, CHANNEL PROBABLY BUSY
        .WRITW  #AREA,#0,#BUFF,#END-BUFF/2,#0
                                        ;WRITE DATA FROM MEMORY, THE
                                        ;SIZE IS # OF WORDS BETWEEN
                                        ;BUFF AND END, START AT BLOCK 0.
        BCS     BADWRT                  ;WRITE FAILURE.
        .CLOSE  #0                      ;CLOSE THE FILE
        .EXIT                           ;AND GO TO KEYBOARD MONITOR.
FPRT:   .RAD50  /DK /                   ;FILE WILL BE ON DK
        .RAD50  /FILE  EXT/             ;NAMED FILE.EXT
AREA:   .BLKW   10                      ;EMT ARGUMENT LIST
BADFET: .PRINT  #FMSG
        .EXIT
BADENT: .PRINT  #EMSG
        .EXIT
BADWRT: .PRINT  #WMSG
        .EXIT
FMSG:   .ASCIZ  /BAD FETCH/
EMSG:   .ASCIZ  /BAD ENTER/
WMSG:   .ASCIZ  /WRITE ERROR/
        .EVEN
LIMIT:  .LIMIT                          ;PROGRAM LIMITS
BUFF:
        .REPT   400                     ;THIS IS BUFFER TO BE WRITTEN OUT
        .WORD   0,1
        .ENDR
END:
        .END    START
```

.EXIT

2.4.15 .EXIT

The .EXIT request causes the user program to terminate. When used from a background job under the FB monitor or XM monitor, or in SJ, .EXIT causes KMON to run in the background area. All outstanding mark time requests are cancelled. Any I/O requests and/or completion routines pending for that job are allowed to complete. If part of the background job resides where KMON and USR are to be read, the user job is written onto the system swap blocks (the file SWAP.SYS). KMON and USR are then loaded and control goes to KMON in the background area. If R0 = 0 when the .EXIT is done, an implicit .HRESET is executed when KMON is entered, disabling the subsequent use of REENTER, START or CLOSE.

The .EXIT request allows a user program to pass command lines to KMON in the chain information area (locations 500-777(octal)) for execution after the job exits. This operation is performed in the following manner:

1. The word (not byte) location 510 must contain the total number of bytes of command lines to be passed to KMON.

2. The command lines are stored beginning at location 512. The lines must be .ASCIZ strings with no embedded carriage return or line feed. For example:

```
        .=510
        .WORD B-A
A:      .ASCIZ /COPY A.MAC B.MAC/
        .ASCIZ /DELETE A.MAC/
B =     .
```

3. The user program must set bit 11 in the JSW immediately prior to doing an .EXIT. The .EXIT must be issued with R0 = 0.

When the .EXIT request is used to input command lines to KMON, the following restrictions are in effect:

1. If the feature is used by a program that is invoked through an indirect file, the indirect file context is aborted prior to executing the supplied command lines. Any unexecuted lines in the indirect file are never executed.

2. An indirect file can be invoked using this mechanism only if a single line containing the indirect file specification is passed to KMON. Attempts to pass multiple indirect files or combinations of indirect command files and other KMON commands yield incorrect results.

EXIT also resets any .CDFN and .QSET calls that were done and executes an .UNLOCK if a .LOCK has been done. Thus, the .CLOSE command from the keyboard monitor does not operate for programs that perform .CDFN requests.

.EXIT from a completion routine is illegal.

NOTE

It is the responsibility of the user program to ensure that the data being passed to KMON is not destroyed during the .EXIT request. Extreme care should be exercised to ascertain that the user stack does not overwrite this data area.

Macro Call: .EXIT

Errors:

None.

Example:

The following example shows how a program can execute a keyboard command after exiting.

```
.TITLE  EXIT.MAC
CHNIF$  = 4000
JSW     = 44
        .MCALL  .EXIT
FINI:   MOV     #510,R0         ;R0 -> COMMUNICATION AREA
        MOV     #CMDSTR,R1      ;R1 -> COMMAND LIST
        MOV     #FINI,SP        ;MAKE SURE THAT THE STACK IS
                                ;NOT IN THE COMMUNICATION AREA
10$:    MOV     (R1)+,(R0)+     ;COPY COMMAND STRING
        CMP     R1,#CMDEND      ;DONE?
        BLO     10$             ;BR IF NOT
        BIS     #CHNIF$,@#JSW   ;SET THE BIT THAT
                                ;TELLS KMON WE LEFT
                                ;A COMMAND LINE FOR IT
        CLR     R0              ;R0 MUST BE ZERO
        .EXIT
CMDSTR: .WORD   CMDEND-CMDST1
CMDST1: .ASCIZ "DIRECT/FULL *.MAC"
CMDEND:
        .EVEN
        .END FINI
```

.FETCH/.RELEAS

2.4.16 .FETCH/.RELEAS

The .FETCH request loads device handlers into memory from the system device.

Macro Call: .FETCH addr,dnam

where:

addr	is the address where the device handler is to be loaded.
dnam	is the pointer to the Radix-50 device name.

The storage address for the device handler is passed on the stack. When the .FETCH is complete, R0 points to the first available location above the handler. If the handler is already in memory, R0 keeps the same value as was initially pushed onto the stack. If the argument on the stack is less than 400(octal), it is assumed that a handler .RELEAS is being done. (.RELEAS does not dismiss a handler that was LOADed from the KMON; an UNLOAD must be done.) After a .RELEAS, a .FETCH must be issued in order to use the device again.

Several requests require a device handler to be in memory for successful operation. These include:

.CLOSE	.READC	.READ
.LOOKUP	.WRITC	.WRITE
.ENTER	.READW	.SPFUN
.RENAME	.WRITW	.DELETE

It is necessary for all handlers to be resident before using a .FETCH in the XM monitor; a fatal error occurs otherwise. In the FB monitor, this is necessary only if the .FETCH is issued from within a foreground job (rather than from a background job).

Errors:

Code	Explanation
0	The device name specified does not exist, or there is no handler for that device in the system.

NOTE

I/O operations cannot be executed on devices unless the handler is resident in memory.

Example:

```
.TITLE  FETCH.MAC
;IN THIS EXAMPLE, THE TT AND PC HANDLERS ARE FETCHED INTO MEMORY
;IN PREPARATION FOR THEIR USE BY A PROGRAM. THE PROGRAM SETS ASIDE
;HANDLER SPACE FROM ITS FREE MEMORY AREA.
        .MCALL  .FETCH,.PRINT,.EXIT,.SETTOP
START:
        MOV     LIMIT+2,FREE        ;SET UP FREE MEMORY POINTER
        .SETTOP #-2                 ;ASK FOR ALL AVAILABLE MEMORY
        MOV     R0,LIMIT+2          ;SAVE THE NEW HIGH LIMIT
        .FETCH  FREE,#TTNAME        ;FETCH HANDLERS AT THE 1ST
                                    ;FREE LOCATION IN MEMORY
        BCS     FERR                ;FETCH ERROR
        MOV     R0,R2               ;R2 => NEXT FREE LOCATION
        .FETCH  R2,#PCNAME          ;FETCH PC HANDLER
                                    ;IMMEDIATELY FOLLOWING
                                    ;TT HANDLER, R0 POINTS
                                    ;TO THE TOP OF PC
                                    ;HANDLER ON RETURN
                                    ;FROM THAT CALL.
        BCS     FERR                ;NO PC HANDLER
        MOV     R0,FREE             ;UPDATE FREE MEMORY
                                    ;POINTER TO POINT TO
                                    ;NEW BOTTOM OF FREE
                                    ;AREA(TOP OF HANDLERS).


        .PRINT  #OK
        .EXIT
OK:     .ASCIZ  /FETCH O.K./
        .EVEN
FERR:   .PRINT  #MSG                ;PRINT ERROR MESSAGE
        .EXIT                       ;AND EXIT
TTNAME: .RAD50  "TT "               ;DEVICE NAMES
PCNAME: .RAD50  "PC "
MSG:    .ASCIZ  "DEVICE NOT FOUND"  ;ERROR MESSAGE
        .EVEN
FREE:   .WORD   0                   ;FREE MEMORY POINTER
LIMIT:  .LIMIT                      ;PROGRAM LIMITS. 2ND
                                    ;WORD IS THE HIGH LIMIT
        .END    START
```

The .RELEAS request notifies the monitor that a FETCHed device handler is no longer needed. The .RELEAS request does not modify memory contents nor does it change any free space pointers. The .RELEAS is ignored if the handler is:

1. Part of RMON (that is, the system device), or

2. Not currently resident, or

3. Resident because of a LOAD command to the keyboard monitor

.RELEAS from the foreground job under the FB monitor or from any job under the XM monitor is always ignored, since the foreground job in FB and all jobs in XM can only use handlers that have been LOADed.

Macro Call: .RELEAS dnam

where: dnam is the address of the Radix-50 device name.

Errors:

Code	Explanation
0	Handler name was illegal.

Example:

```
        .TITLE  RELEAS.MAC
;IN THIS EXAMPLE, THE DECTAPE HANDLER (DT) IS LOADED INTO MEMORY,
;USED, THEN RELEASED. IF THE SYSTEM DEVICE IS DECTAPE, THE HANDLER IS
;ALWAYS RESIDENT, AND .FETCH WILL RETURN HSPACE IN R0.
        .MCALL  .FETCH,.RELEAS,.EXIT

START:  .FETCH  LIMIT+2,#DTNAME ;LOAD DT HANDLER
        BCS     FERR            ;NOT AVAILABLE

; USE HANDLER

        .RELEAS #DTNAME         ;MARK DT NO LONGER IN
                                ;MEMORY.
        BR      START
FERR:   HALT                    ;DT NOT AVAILABLE
DTNAME: .RAD50  /DT /           ;NAME FOR DT HANDLER
LIMIT:  .LIMIT                  ;PROGRAM LIMITS

        .END    START
```

.FORK

2.4.17 .FORK

.FORK can be used within a standard RT-11 device driver to request a synchronous system process after an interrupt occurs. The request does not use the EMT instruction but issues a subroutine call to the monitor. The .FORK call must be preceded by an .INTEN call, and the address of a four-word block must be supplied with the request. The user program must not have left any information on the stack between the .INTEN and the .FORK call. The contents of registers R4 and R5 are preserved through the call, and on return registers R0-R3 are available for use.

The .FORK request is used when access to a shared resource must be serialized or when a lengthy but non-time-critical section of code must be executed. The .FORK request is linked into a queue and serviced on a first-in/first-out basis. On return to the driver instruction following the call, the interrupt has been dismissed and the driver is executing at priority 0. Therefore, the .FORK request must not be used where it can be reentered using the same fork block by another interrupt, for example. It also should not be used with devices that have continuous interrupts that cannot be disabled. Chapter 1 of this manual has additional information on the .FORK request.

Macro call: .FORK fkblk

where: fkblk is a four-word block of memory allocated within the driver.

Errors:

None.

Note:

For use within a user interrupt service routine, monitor fixed offset 402 (FORK) contains the offset from the start of the resident monitor to the .FORK request processor. A .FORK request can be done by computing the address of the .FORK request processor and using a subroutine instruction. (Under the XM monitor, only privileged jobs can contain user interrupt service routines.) For example:

```
MOV    @#54,R4      ;GET BASE OF RMON
ADD    #402,R4      ;OFFSET TO FORK PROCESSOR
JSR    R5,@R4       ;CALL FORK PROCESSOR
.WORD  BLOCK-.      ;FORK BLOCK
```

.GTIM

2.4.18 .GTIM

.GTIM allows user programs to access the current time of day. The time is returned in two words, and is given in terms of clock ticks past midnight.

Macro Call: .GTIM area,addr

where: area is the address of a two-word EMT argument block.

addr is a pointer to the two-word area where the time is to be returned.

Request Format:

R0 → area:	21	0
	addr	

The high-order time is returned in the first word, the low-order time in the second word. User programs must make the conversion from clock ticks to hours, minutes, and seconds.

The basic clock frequency (50 or 60 Hz) can be determined from the configuration word in the monitor (see Section 2.2.6). In the FB monitor, the time of day is automatically reset after 24:00 when a .GTIM is done; in the SJ monitor, it is not. The month is not automatically updated in either monitor.

The default clock rate is 60-cycle. Consult the RT-11 System Generation Manual if conversion to a 50-cycle rate is necessary.

NOTE

There are also several SYSLIB routines that perform time conversion. They are as follows:

1. CVTTIM (see Section 4.3.5)
2. TIMASC (see Section 4.3.98)
3. TIME (see Section 4.3.99)
4. SECNDS (see Section 4.3.93)

Errors:

None.

Example:

```
.TITLE  GTIM.MAC
        .MCALL  .GTIM,.EXIT
START:
        .GTIM   #LIST,#TIME

        .EXIT
TIME:   .WORD   0,0             ;LOW AND HI ORDER TIME
                                ;RETURNED HERE.

LIST:   .BLKW   2               ;ARGUMENTS FOR THE EMT

        .END    START
```

.GTJB

2.4.19 .GTJB

The .GTJB request passes a job number, the low memory limit and other job parameters back to the user program.

In the SJ monitor, the job number and low memory limit are always 0. In the FB or XM monitor, the job number can either be 0 or 2. If the job number equals 0 (background job), word 3 equals 0.

Word 4 describes where the I/O channel words begin. This is normally an address within the resident monitor. When a .CDFN is executed, however, the start of the I/O channel area changes to the user-specified area.

Macro Call: .GTJB area,addr

where: area is the address of a two-word EMT argument block.

addr is the address of an eight-word block into which the parameters are passed. The values returned are:

Word 1 - Job Number.
0=Background
2=Foreground
2 - High memory limit of job partition
3 - Low memory limit of job partition
4 - Beginning of I/O channel space
5 - Address of job's impure area in FB and XM monitors
6-8 - Reserved for future use

Request Format:

R0 → area:		
	20	0
	addr	

Errors:

None.

Example:

```
        .TITLE  GTJB.MAC
        ;USE .GTJB TO DETERMINE IF THIS PROGRAM IS EXECUTING AS A FOREGROUND
        ;OR A BACKGROUND JOB.
                .MCALL  .GTJB,.PRINT,.EXIT

START:
                .GTJB   #LIST,#JOBARG   ;R0 POINTS TO 1ST WORD ON
                                        ;RETURN FROM CALL.
                MOV     #FMSG,R1
                TST     JOBARG          ;BACKGROUND?
                BNE     1$              ;NO, PRINT FMSG
                MOV     #BMSG,R1
1$:             .PRINT  R1

                .EXIT

FMSG:           .ASCIZ  /PROGRAM IN FOREGROUND/
BMSG:           .ASCIZ  /PROGRAM IN BACKGROUND/
                .EVEN

LIST:           .BLKW   2               ;ARGUMENTS FOR THE EMT
JOBARG:         .BLKW   8.              ;JOB PARAMETERS PASSED BACK HERE.

                .END    START
```

.GTLIN

2.4.20 .GTLIN

This request is used to collect a line of input from either the console terminal or an indirect command file, if one is active. This request is similar to .CSIGEN and .CSISPC in that it requires the USR, but no format checking is done on the input line. Normally, .GTLIN collects a line of input from the console terminal and returns it in the buffer specified by the user. However, if there is an indirect command file active, .GTLIN collects the line of input from the command file just as though it were coming from the terminal.

An optional prompt string argument is supported to allow the user to be queried for input at the terminal. (It is similar to the CSI's asterisk.) The prompt string argument is an ASCIZ character string in the same format as that used by the .PRINT request. If input is from an indirect command file and the SET TTT QUIET option is in effect, this prompt is suppressed. If SET TT QUIET is not in effect, the prompt is printed before the line is collected, regardless of whether the input comes from the terminal or an indirect file. The prompt appears only once. It is not reissued if an input line is cancelled from the terminal by CTRL/U or multiple DELETEs.

User programs that require nonstandard command format, such as the UIC specification for FILEX, can use the .GTLIN request to accept the command string input line. .GTLIN tracks indirect command files and the user program can do a pre-pass of the input line to remove the nonstandard syntax before passing the edited line to .CSIGEN or .CSISPC.

Macro Call: .GTLIN linbuf[,prompt]

where: linbuf — is the address of the buffer to receive the input line. This is a user-specified area up to 81 decimal bytes in length. The input line is stored in this area and is terminated with a zero byte instead of RET LF (octal 15 and 12).

prompt — is an optional argument and is the address of a prompt string to be printed on the console terminal. The prompt string has the same format as the argument of a .PRINT request.

NOTE

The only requests that can take their input from an indirect command file are .CSIGEN, .CSISPC and .GTLIN. The .TTYIN and .TTINR requests cannot get characters from an indirect command file; their input comes from the console terminal (or from a BATCH file if BATCH is running). The .TTYIN and .TTINR requests are useful for information that is dynamic in nature. For instance, the response to a system query when deleting all files with a .MAC file type or when initializing a disk is usually collected through a .TTYIN so that confirmation can be done interactively, even though the process may have been invoked through an indirect command file. However, the response to the linker's "TRANSFER SYMBOL" query would normally be collected through a .GTLIN, so that the LINK command could be invoked and the start address specified from an indirect file. Note also that if there is no active indirect command file, .GTLIN simply collects an input line from the console terminal by using .TTYINs.

Errors:

None.

Example:

This example prompts the terminal and accepts a line of input. If the first input character is in the range A through M, the example prints the line back at the terminal.

```
.TITLE  GTLIN.MAC
        .MCALL  .GTLIN, .PRINT,.EXIT

START:  .GTLIN  #LINBUF,#PROMPT ;GET A LINE OF INPUT
        MOVB    LINBUF,R0       ;PICK UP FIRST CHARACTER
        BEQ     EXIT            ;IF BLANK LINE-EXIT
        CMPB    R0,#'A          ;NOT BLANK, DOES IT BEGIN WITH A-M?
        BLO     START           ;IF LO, NO, JUST GET ANOTHER LINE
        CMPB    R0,#'M          ;MAYBE, CHECK HIGH LIMIT
        BHI     START           ;IF HI, NOT IN RANGE
        .PRINT  #LINBUF         ;OK, PRINT LINE
        BR      START           ;GO GET ANOTHER
EXIT:   .EXIT                   ;BACK TO MONITOR

PROMPT: .ASCII  /MY PROGRAM>/<200>

LINBUF: .BLKB   82.             ;LINE BUFFER
        .EVEN

        .END    START
```

.GVAL

2.4.21 .GVAL

This request returns a monitor fixed offset value in R0 where it can be accessed by the user. This request must be used in the XM monitor to access monitor fixed offset locations, but it should also be used in other RT-11 monitors. The .GVAL request is a read-only operation and provides protection for information obtained from the monitor.

Macro Call: .GVAL area,offse

where: area — is the address of a two-word EMT argument block.

offse — is the displacement from the beginning of the monitor of the word to be returned in R0.

Request Format:

R0 → area:

34	0
offse	

Errors:

Code	Explanation
0	The offset requested is beyond the limits of the resident monitor.

Example:

The following example demonstrates use of the .GVAL request by getting the monitor version number and update number from the resident monitor.

```
.TITLE  GVAL.MAC
        .MCALL  .GVAL,.EXIT
UPDATE  = 276                   ;OFFSET TO MONITOR
                                ;VERSION NUMBER
START:
        .GVAL   #AREA,#UPDATE   ;GET MONITOR VERSION
                                ;NUMBER AND UPDATE
                                ;IN R0
        MOVB    R0,MONVER       ;STORE VERSION #
        SWAB    R0              ;UPDATE TO LOW BYTE
        MOVB    R0,MONUPD       ;STORE UPDATE #
        .EXIT
MONVER: .BLKB                   ;MONITOR VERSION #
MONUPD: .BLKB                   ;MONITOR UPDATE #
AREA:   .BLKW   2
        .END    START
```

.HERR/.SERR

2.4.22 .HERR/.SERR

.HERR and .SERR are complementary requests used to govern monitor behavior for serious error conditions. During program execution, certain error conditions can arise that cause the executing program to be aborted (see Table 2-3). Normally, these errors cause program termination with one of the ?MON- error messages. However, in certain cases it is not feasible to abort the program because of these errors. For example, a multi-user program must be able to retain control and merely abort the user who generated the error. .SERR accomplishes this by inhibiting the monitor from aborting the job. Instead, it causes an error return to the offending EMT to be taken. On return from that request, the carry bit is set and byte 52 contains a negative value indicating the error condition that occurred. In some cases (such as the .LOOKUP and ENTER requests), the .SERR request leaves channels open.

.HERR turns off user error interception; it allows the system to abort the job on fatal errors and generate an error message. (.HERR is the default case.)

Macro Calls: .HERR

.SERR

Request Formats:

R0 =	4	0
R0 =	5	0

Errors:

Table 2-3 contains a list of the errors that are returned if soft error recovery is in effect. Traps to 4 and 10, and floating point exception traps are not inhibited. These errors have their own recovery mechanism.

Table 2-3
Soft Error Codes (SERR)

Code	Explanation
-1	Called USR from completion routine.
-2	No device handler; this operation needs one.
-3	Error doing directory I/O.
-4	.FETCH error. Either an I/O error occurred while reading the handler, or tried to load it over USR or RMON.
-5	Error reading an overlay.
-6	No more room for files in the directory.
-7	Illegal address (FB only); tried to perform a monitor operation outside the job partition.
-10	Illegal channel number; number is greater than actual number of channels which exist.
-11	Illegal EMT; an illegal function code has been decoded.

Example:

```
.TITLE  MERR.MAC
;THIS EXAMPLE CAUSES A NORMALLY FATAL ERROR TO GENERATE ERRORS
;BACK TO THE USER PROGRAM. THE ERROR RETURNED IS USED TO PRINT
;AN APPROPRIATE MESSAGE.
        .MCALL  .FETCH,.ENTER,.MERR,.SERR
        .MCALL  .EXIT,.PRINT

ST:     .SERR                   ;TURN ON SOFTWARE ERROR
                                ;RETURNS
        .FETCH  #HDLR,#PTR      ;GET A DEVICE HANDLER
        BCS     FCHERR
        .ENTER  #AREA,#1,#PTR   ;OPEN A FILE ON CHANNEL 1
        BCS     ENERR
        .MERR                   ;NOW PERMIT ?M-ERRORS.
        .EXIT

FCHERR: MOVB    @#52,R0         ;WAS IT FATAL
        BMI     FTLERR          ;YES
        .PRINT  #FMSG           ;NO... NO DEVICE BY THAT NAME
        .EXIT
ENERR:  MOVB    @#52,R0
        BMI     FTLERR
        .PRINT  #EMSG
        .EXIT
FTLERR: NEG     R0              ;THIS WILL TURN POSITIVE
        DEC     R0              ;ADJUST BY ONE
        ASL     R0              ;MAKE IT AN INDEX
        MOV     TBL(R0),R0      ;PUT MESSAGE ADDRESS INTO R0
        .PRINT                  ;AND PRINT IT.
        .EXIT

TBL:    M1                      ;CAN'T OCCUR IN THIS PROGRAM
        M2                      ;NO DEVICE HANDLER IN MEMORY
        M3                      ;DIRECTORY I/O ERROR
        M4                      ;FETCH ERROR
        M5                      ;IMPOSSIBLE FOR THIS PROGRAM
        M6                      ;NO ROOM IN DIRECTORY
        M7                      ;ILLEGAL ADDRESS (F/B)
        M10                     ;ILLEGAL CHANNEL
        M11                     ;ILLEGAL EMT

M1:                             ;CAN'T OCCUR IN THIS PROGRAM
M2:     .ASCIZ  /NO DEVICE HANDLER/
M3:     .ASCIZ  "DIRECTORY I/O ERROR"
M4:     .ASCIZ  /ERROR DOING FETCH/
M5:                             ;NOT APPLICABLE TO THIS PROGRAM
M6:     .ASCIZ  /NO ROOM IN DIRECTORY/
M7:     .ASCIZ  /ADDRESS CHECK ERROR/
M10:    .ASCIZ  /ILLEGAL CHANNEL/
M11:    .ASCIZ  /ILLEGAL EMT/
FMSG:   .ASCIZ  /FETCH FAILED/
EMSG:   .ASCIZ  /ENTER FAILED/
        .EVEN
HDLR:   .BLKW   300             ;LEAVE 300 (OCTAL) FOR HANDLER
PTR:    .RAD50  /DT4/           ;DEVICE AND FILE NAME.
        .RAD50  /EXAMPL/
        .RAD50  /MAC/
AREA:   .BLKW   4               ;EMT AREA
        .END    ST
```

.HRESET

2.4.23 .HRESET

This request stops all I/O transfers in progress for the issuing job, and then performs an .SRESET (see Section 2.4.54). (.HRESET is not used to clear a hard-error condition.) Note that in the SJ environment, a hardware RESET instruction is used to terminate I/O, while in a FB environment, only the I/O associated with the job that issued the .HRESET is affected. All other transfers continue.

Macro call: .HRESET

Errors:

None.

Example:

See the example for .SRESET for format.

.INTEN

2.4.24 .INTEN

This request is used by user program interrupt service routines to:

1. Notify the monitor that an interrupt has occurred and to switch to system state.

2. Set the processor priority to the correct value.

The .INTEN request is not an EMT monitor request but rather a subroutine call to the monitor.

All external interrupts cause the processor to go to priority level 7. .INTEN is used to lower the priority to the value at which the device should be run. On return from .INTEN, the device interrupt can be serviced, at which point the interrupt routine returns with an RTS PC. It is very important to note that an RTI does not return correctly from an interrupt routine that specifies an .INTEN.

Macro Call: .INTEN prio[,pic]

where: prio — is the processor priority at which the user needs to run the interrupt routine, normally the priority at which the device requests an interrupt.

pic — is an optional argument that should be non-blank if the interrupt routine is written as a PIC (position independent code) routine. Any interrupt routine written as a device handler must be a PIC routine and must use this argument.

Errors:

None.

Example:

See the example for .SYNCH.

.LOCK /.UNLOCK

2.4.25 .LOCK/.UNLOCK

.LOCK

The .LOCK request is used to keep the USR in memory for a series of operations. If all the conditions that cause swapping are satisfied, the part of the user program over which the USR swaps is written into the system swap blocks (the file SWAP.SYS) and the USR is loaded. Otherwise, the copy of the USR in memory is used, and no swapping occurs. A .LOCK request always causes the USR to be loaded in memory if it is not already in memory. The USR is not released until an .UNLOCK request is given. (Note that in an FB system, calling the CSI can also perform an implicit .UNLOCK.) A program that has many USR requests to make can .LOCK the USR in memory, make all the requests, and then .UNLOCK the USR; no time is spent doing unnecessary swapping.

In a FB environment, a .LOCK inhibits the other job from using the USR. Note that the .LOCK request reduces time spent in file handling by eliminating the swapping of the USR in and out of memory. .LOCK causes the USR to be read into memory or swapped into memory. After a .LOCK has been executed, an .UNLOCK request must be executed to release the USR from memory. The .LOCK/.UNLOCK requests are complementary and must be matched. That is, if three .LOCK requests are issued, at least three .UNLOCKs must be done, otherwise the USR is not released. More .UNLOCKs than .LOCKs can be issued without error.

Macro Call: .LOCK

Notes:

1. It is vital that the .LOCK call not come from within the area into which the USR will be swapped. If this should occur, the return from the .LOCK request would not be to the user program, but to the USR itself, since the LOCK function inhibits the user program from being re-read.

2. Once a .LOCK has been performed, it is not advisable for the program to destroy the area the USR is in, even though no further use of the USR is required. This causes unpredictable results when an .UNLOCK is done.

3. If a foreground job performs a .LOCK request while the background job owns the USR, foreground execution is suspended until the USR is available. In this case, it is possible for the background to lock out the foreground (see the .TLOCK request).

Errors:

None.

Example:

See the example following .UNLOCK.

.UNLOCK

The .UNLOCK request releases the User Service Routine (USR) from memory if it was placed there with a .LOCK request. If the .LOCK required a swap, the .UNLOCK loads the user program back into memory. There is a .LOCK count. Each time the user does a .LOCK, the lock count is incremented. When the user does an .UNLOCK, the lock count is decremented. When it goes to 0, the user program is swapped back in. See note 1.

Macro Call: .UNLOCK

Notes:

1. It is important that at least as many .UNLOCKs are given as .LOCKs. If more .LOCK requests are done, the USR remains locked in memory. It does no harm to give more .UNLOCKs than are required; those that are extra are ignored.

2. The .LOCK/.UNLOCK pairs should be used only when absolutely necessary when running two jobs in the FB system. When a job .LOCKs the USR, the other job cannot use it until it is .UNLOCKed. Thus, the USR should not be .LOCKed unnecessarily, as this can degrade performance in some cases.

3. In an FB system, calling the CSI with input coming from the console terminal performs an implicit .UNLOCK.

4. It is especially important that the .UNLOCK not be in the area that the USR swaps into. Otherwise, the request can never be executed.

Errors:

None.

Example:

The following example tries to obtain as much memory as it can (with the .SETTOP request). Most likely this does, in a background job, make the USR non-resident (unless a SET USR NOSWAP command is done at the keyboard), and swapping must take place for each .LOOKUP given. Using the .LOCK, the USR is brought into memory and remains there until the .UNLOCK is given.

The second .LOOKUP makes use of the fact that the arguments have already been set up at LIST. Thus, it is possible to increment the channel number, put in a new file pointer and then give a simple .LOOKUP, which does not cause any arguments to be moved into LIST.

```
.TITLE  LOCK.MAC
;THIS EXAMPLE SHOWS THE USAGE OF .LOCK, .UNLOCK, AND THEIR
;INTERACTION WITH THE SYSTEM.
        .MCALL  .LOCK,.UNLOCK,.LOOKUP
        .MCALL  .SETTOP,.PRINT,.EXIT

START:

SYSPTR=54

        .SETTOP @#SYSPTR        ;TRY FOR ALL OF MEMORY
        MOV     R0,TOP          ;R0 HAS THE TOP
        .LOCK                   ;BRING USR INTO MEMORY
        .LOOKUP #LIST,#0,#FILE1 ;LOOKUP A FILE ON CHANNEL 0
        BCC     1$              ;ON ERROR, PRINT A
2$:     .PRINT  #LMSG           ;MESSAGE AND EXIT
        .EXIT
1$:     MOV     #LIST,R0
        INC     (R0)            ;DO LOOKUP ON CHANNEL 1
        MOV     #FILE2,2(R0)    ;NEW POINTER
        .LOOKUP         ;ALL ARGS ARE FILLED IN
        BCS     2$
        .UNLOCK                 ;NOW RELEASE USR
        .EXIT


LIST:   .BLKW   3               ;SPACE FOR ARGUMENTS
FILE1:  .RAD50  /DK /
        .RAD50  /FILE1 MAC/
FILE2:  .RAD50  /DK /
        .RAD50  /FILE2 MAC/
TOP:    .WORD   0
LMSG:   .ASCIZ  /LOOKUP ERROR/
        .EVEN

        .END    START
```

.LOOKUP

2.4.26 .LOOKUP

The .LOOKUP request associates a specified channel with a device and existing file, for the purpose of performing I/O operations. The channel used is then busy until one of the following requests is executed:

.CLOSE
.SAVESTATUS
.SRESET
.HRESET
.PURGE
.CSIGEN (if the channel is in the range 0-10 octal)

Note that if the program is overlaid, channel 15 (decimal) or 17 (octal) is used by the overlay handler and should not be modified.

If the first word of the file name in dblk is 0 and the device is a file-structured device, absolute block 0 of the device is designated as the beginning of the file. This technique is called a non-file-structured .LOOKUP, and allows I/O operations to access any physical block on the device. If a file name is specified for a device that is not file-structured (such as PC:FILE.TYP), the name is ignored.

The handler for the selected device must be in memory for a .LOOKUP. On return from the .LOOKUP, R0 contains the length in blocks of the file just opened. On a return from a .LOOKUP for a non-directory file-structured device, R0 contains 0 for the length.

NOTE

> Care should be exercised when doing a non-file-structured .LOOKUP on a file-structured device, since corruption of the device directory can occur and effectively destroy the disk. (The RT-11 directory starts in absolute block 6.)
>
> In particular, avoid doing a .LOOKUP or .ENTER with a file specification that is missing the file value. If the device type is not known in advance and is to be entered from the keyboard, include a dummy file name with the .LOOKUP or .ENTER, even when it is assumed that the device is always non-file-structured.

Macro Call: .LOOKUP area,chan,dblk,seqnum

where:

- area — is the address of a three-word EMT argument block.
- chan — a channel number in the range 0-377(octal).
- dblk — the address of a four-word Radix-50 descriptor of the file to be operated upon.
- seqnum — file number. For cassette operations, if this argument is blank, a value of 0 is assumed. For magtape, it describes a file sequence number that can have the following values.
  - -1 means suppress rewind and search for a file name from the current tape position. If the position is not known, then the handler executes a positioning algorithm that involves backspacing until an end-of-file label is found.*
  - 0 means rewind the magtape and space forward until the file name is found.

---

* It is important that -1 be specified and no other negative number.

n where n is any positive number. This means position the tape at file sequence number n. If the file represented by the file sequence number is greater than two files away from the beginning-of-tape, then a rewind is performed. If not, the tape is backspaced to the beginning of the file.

Request Format:

R0 → area:		
	1	chan
	dblk	
	seqnum	

Errors:

Code	Explanation
0	Channel already open.
1	File indicated was not found on the device.

Example:

```
.TITLE  LOOKUP.MAC
;IN THIS EXAMPLE, THE FILE "DATA.001" ON DEVICE DT3:
;IS OPENED FOR INPUT  ON CHANNEL 7.
        .MCALL  .FETCH,.LOOKUP,.PRINT,.EXIT

START:
ERRBYT=52
        .FETCH  #HSPACE,#DT3N   ;GET DEVICE HANDLER
        BCS     FERR            ;DT3 IS NOT AVAILABLE
        .LOOKUP #LIST,#7,#DT3N  ;LOOKUP THE FILE
                                ;ON CHANNEL 7
        BCC     LDONE           ;FILE WAS FOUND
        TSTB    @#ERRBYT                ;ERROR, WHAT'S WRONG?
        BNE     NFD             ;FILE NOT FOUND
        .PRINT  #CAMSG          ;PRINT  'CHANNEL ACTIVE'
        .EXIT
NFD:    .PRINT  #NFMSG          ;FILE NOT FOUND
        .EXIT
CAMSG:  .ASCIZ  /CHANNEL ACTIVE/
NFMSG:  .ASCIZ  /FILE NOT FOUND/ ;ERROR MESSAGES
DTMSG:  .ASCIZ  /DT3 NOT AVAILABLE/
        .EVEN
FERR:   .PRINT  #DTMSG
        .EXIT
LDONE:                          ;PROGRAM CAN NOW
                                ;ISSUE READS  AND
                                ;WRITES TO FILE
                                ;DATA.001 VIA
                                ;CHANNEL 7

        .EXIT

LIST:   .BLKW   5
DT3N:   .RAD50  "DT3"           ;DEVICE
        .RAD50  "DAT"           ;FILENAME
        .RAD50  "A  "           ;FILENAME
        .RAD50  "001"           ;EXTENSION

HSPACE:                 ;RESERVED SPACE FOR DT
        .=.+400                 ;HANDLER

        .END    START
```

.MFPS/.MTPS

2.4.27 .MFPS/.MTPS

The .MFPS and .MTPS macro calls allow processor-independent user access to the processor status word. The contents of R0 are preserved across either call.

The .MFPS call is used to read the priority bits only; condition codes are destroyed during the call and must be directly accessed (using conditional branch instructions) if they are to be read in a processor-independent manner.

In the XM monitor, .MFPS and .MTPS can be used only by privileged jobs; they are not available for use by virtual jobs.

Macro Call: .MFPS addr

where:	addr	is the address into which the processor status is to be stored; if addr is not present, the value is returned on the stack. Note that only the priority bits are significant.

The .MTPS call is used to set the priority, condition codes, and trace trap bit with the value designated in the call.

Macro Call: .MTPS addr

where:	addr	is the address of the word to be placed in the processor status word; if addr is not present, the processor status word is taken from the stack. Note that the high byte on the stack is set to 0 when addr is present. If addr is not present, the user should set the stack to the appropriate value. In either case, the lower byte on the stack is put in the processor status word.

Notes:

It is possible to do .MFPS and .MTPS operations and also access the condition codes by a special technique. The routines $MFPS and $MTPS are accessed by monitor fixed offsets, and condition codes are normally destroyed in the macros when the addresses of the routines are constructed. If the programmer constructs them in advance, calls can be made that return condition codes. An example of this technique follows:

```
                                        ;Fixed offset values
                                        ;$MTPS 360
                                        ;$MFPS 362
          ADD @#54,PMTPS                ;Relocate PMTPS
          ADD @#54,PMFPS                ;Relocate PMFPS
               .
               .
               .
          JSR PC,@PMFPS                 ;Put PSW
               .
               .
               .
          JSR PC,@PMTPS                 ;Put PSW
               .
               .
               .
PMTPS:    .WORD 360
PMFPS:    .WORD 362
               .
               .
               .
```

Errors:

None.

Example:

```
        .TITLE  MFPS
        .MCALL  .MFPS,.MTPS,.EXIT

START:  JSR     PC,PICKQ        ;PICK A QUEUE ELEMENT
                .
                .
                .
        .EXIT

PICKQ:  .MFPS                   ;SAVE PREVIOUS PRIORITY ON STACK
        MOV     #QHEAD,R4       ;POINT TO QUEUE HEAD
        .MTPS   #340            ;RAISE PRIORITY TO 7
        MOV     @R4,R5          ;R5 POINTS TO NEXT ELEMENT
        BEQ     10$             ;NO MORE ELEMENTS AVAILABLE
        MOV     @R5,@R4         ;RELINK THE QUEUE
        .MTPS                   ;RESTORE PREVIOUS PRIORITY
        CLZ                     ;FLAG SUCCESS
        BR      20$
10$:    .MTPS                   ;RESTORE PREVIOUS PRIORITY
        SEZ                     ;INDICATE FAILURE
20$:    RTS     PC

QHEAD:  .WORD   Q1              ;QUEUE HEAD

;THREE QUEUE ELEMENTS
Q1:     .WORD   Q2,0,0
Q2:     .WORD   Q3,0,0
Q3:     .WORD   0,0,0

        .END    START
```

.MRKT

2.4.28 .MRKT (FB and XM Only; SJ Monitor SYSGEN Option)

The .MRKT request schedules a completion routine to be entered after a specified time interval (clock ticks) has elapsed. .MRKT is an optional feature in the SJ monitor; timer support must be selected at system generation time to obtain it.

.MRKT requests require a queue element taken from the same list as the I/O queue elements. The element is in use until either the completion routine is entered or a cancel mark time request is issued. The user should allocate enough queue elements to handle at least as many mark time and I/O requests as he expects to have pending simultaneously.

Macro Call: .MRKT area,time,crtn,id

where:

- area — is the address of a four-word EMT argument block.
- time — is the pointer to the two words containing the time interval (high-order first, low-order second).
- crtn — is the entry point of a completion routine.
- id — is a number assigned by the user to identify the particular request to the completion routine and to any cancel mark time requests. The number must not be within the range 177400 - 177777; these are reserved for system use. The number need not be unique (several .MRKT requests can specify the same id). On entry to the completion routine, the id number is in R0.

Request Format:

R0 → area:		
	22	0
	time	
	crtn	
	id	

Errors:

Code	Explanation
0	No queue element was available.

Example:

```
.TITLE  MRKT.MAC
;IN THIS EXAMPLE, A MARK TIME IS SET UP TO TIME OUT AN I/O
;TRANSFER. IF THE MARK TIME EXPIRES BEFORE THE TRANSFER IS DONE
;A MESSAGE IS PRINTED. IF THE I/O TRANSFER COMPLETES BEFORE THE
;MARK TIME, THE MARK TIME IS CANCELLED.
        .MCALL  .READ,.WAIT,.MRKT,.CMKT
        .MCALL  .QSET,.PRINT,.EXIT,.LOOKUP

ST:     .LOOKUP #AREA,#0,#FILE  ;OPEN A FILE
        BCS     LKERR           ;FILE NOT FOUND
        MOV     #AREA,-(SP)     ;EMT LIST TO STACK
        .QSET   #QUEUE,#5       ;ALLOCATE 5 MORE ELEMENTS
        .MRKT   (SP),#INTRVL,#MRTN,#1 ;SET TIMER GOING
        BCS     NOMRKT          ;FAILED.
        .READ   #RDLST          ;START  I/O TRANSFER
        BCS     RDERR
        .WAIT   #0              ;AND WAIT A WHILE.
        .CMKT   (SP),#1         ;SEE IF MARK TIME IS
                                ;DONE.
        BCS     NOTDUN          ;FAILED. THAT MEANS THAT
                                ;THE MARK TIME ALREADY
                                ;EXPIRED.

        .EXIT


MRTN:   .CMKT   (SP),#1         ;OK, KILL THE TIMER.
        .PRINT  #FAIL           ;DON'T WORRY ABOUT AN
                                ;ERROR HERE.
        RTS     PC
LKERR:  .PRINT  #LM
        .EXIT
RDERR:  .PRINT  #RDMSG
        .EXIT
NOTDUN: .PRINT  #FAIL
        .EXIT
NOMRKT: .PRINT  #NOQ
        .EXIT
NOQ:    .ASCIZ  /NO QUEUE ELEMENTS AVAILABLE/
FAIL:   .ASCIZ  /MARK TIME COMPLETED BEFORE TRANSFER/
LM:     .ASCIZ  /LOOKUP ERROR/
RDMSG:  .ASCIZ  /READ ERROR/
        .EVEN
INTRVL: .WORD   0,13.           ;ALLOW 13 CLOCK
                                ;TICKS FOR TRANSFER.
QUEUE:  .BLKW   5*7             ;AREA FOR QUEUE ELEMENTS
AREA:   .BLKW   5               ;A FEW WORDS FOR EMT LIST
FILE:   .RAD50  /DK FILE  TST/
RDLST:  .BYTE   0               ;CHANNEL 0
        .BYTE   10              ;A READ
BLOCK:  .WORD   0               ;BLOCK #
        .WORD   BUFF            ;BUFFER
        .WORD   256.            ;1 BLOCK
        .WORD   1
BUFF:   .BLKW   256.

        .END    ST
```

.MTATCH

2.4.29 .MTATCH (FB and XM Monitor SYSGEN Option)

This request attaches a terminal for exclusive use by the requesting job. This operation must be performed before any job can use a terminal with multi-terminal programmed requests.

Macro Call: .MTATCH area,addr,unit

where:

area	is the address of a three-word EMT argument block
addr	is the optional address of an asynchronous terminal status word or it must be 0. (The Asynchronous Terminal Status word is a SYSGEN option.)
unit	is the logical unit number (lun) of the terminal.

Request Format:

R0 → area:

37	5
addr	
	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52:

Codes	Explanation
2	Non-existent lun
3	Illegal request, function code out of range
4	Unit attached by another job
5	In the XM monitor, the optional status word address is not in valid user virtual address space.

Example:

The following example demonstrates use of the multi-terminal asynchronous status option by polling all terminals to determine which terminals are on-line. The example in the section on the .MTSET request also illustrates the .MTATCH request.

```
        .MCALL  .MTATCH,.MTPRNT,.EXIT

AS.CAR  = 200                           ;IF SET, INDICATES CARRIER
                                        ; PRESENT.
START:
        CLR     R1                      ;INITIALIZE LUN
        MOV     #AST,R2                 ;R2 -> ASYNCHRONOUS STATUS WORD
10$:    .MTATCH #MTA,R2,R1              ;ATTEMPT TO ATTACH TERMINAL
        BCC     15$                     ;BR IF SUCCESSFUL
        CLRB    TAI(R1)                 ;SIGNAL ATTACH UNSUCCESSFUL
        BR      20$                     ;GO TRY NEXT LUN
15$:    MOVB    #1,TAI(R1)              ;SIGNAL ATTACH SUCCESSFUL
        BIT     #AS.CAR,(R2)            ;TERMINAL ON LINE?
        BEQ     20$                     ;BR IF HUNG UP
        .MTPRNT #MTA,#HELLO,R1          ;SAY HELLO
20$:    ADD     #2,R2                   ;R2 -> NEXT AST WORD
        INC     R1                      ;NEXT LUN
        CMP     R1,#16.                 ;DONE?
        BLO     10$                     ;BR IF NOT
        .EXIT
AST:    .BLKW   16.                     ;ASYNCHRONOUS TERMINAL STATUS WORDS
MTA:    .BLKW   3                       ;PARAMETER BLOCK FOR EMT'S
HELLO:  .ASCIZ  /WE'RE ON THE AIR/
TAI:    .BLKB   16.                     ;0 => TERMINAL NOT ATTACHED
                                        ;1 => TERMINAL ATTACHED
        .END    START
```

.MTDTCH

2.4.30 .MTDTCH (FB and XM Monitor SYSGEN Option)

This request detaches a terminal from one job and makes it available for other jobs. When a terminal is detached, it is deactivated and unsolicited interrupts are ignored. Input is disabled immediately, but any characters in the output buffer are allowed to print. Attempts to detach a terminal attached by another job results in an error. However, attempts to detach an unattached terminal are ignored.

Macro Call: .MTDTCH area,unit

where: area — is a three-word EMT argument block.

unit — is the logical unit number (lun) of the terminal.

Request Format:

R0→area:

37	6
(unused)	
..	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52.

Code	Explanation
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range

Example:

```
.TITLE  MTDTCH.MAC
        .MCALL  .MTDTCH,.MTPRNT,.MTATCH,.EXIT,.PRINT
START:
        .MTATCH #MTA,#0,#3          ;ATTACH TO LUN 3
        BCC     1$                  ;ATTACH ERROR
        .MTPRNT #MTA,#MESS,#3       ;PRINT MESSAGE
        .MTDTCH #MTA,#3             ;DETACH LUN 3
        .EXIT
1$:     .PRINT  #ATTERR             ;ATTACH ERROR
                                    ;(PRINTED ON CONSOLE)
        .EXIT
ATTERR: .ASCIZ/ATTACH ERROR/
MESS:   .ASCIZ/DETACHING TERMINAL/
        .EVEN
MTA:    .BLKW   3
        .END    START
```

.MTGET

2.4.31 .MTGET (FB and XM Monitor SYSGEN Option)

This request returns the status of the specified terminal unit to the caller.

When the program returns from the request, the status block contains the following information:

Byte Offset	Contents
0	Terminal configuration word 1. The bit definitions are the same as those for the .MTSET request.
2	Terminal configuration word 2
4	Character requiring fillers
5	Number of fillers
6	Carriage width (byte).
7	Current carriage position (byte).

Macro Call: .MTGET area,addr,unit

where:

- area — is the address of a three-word EMT argument block.
- addr — is the address of a four-word status block where the status information is returned (see Section 2.4.36).
- unit — is the logical unit number (lun) of the terminal whose status is requested.

Request Format:

R0→area:	37	1
	addr	
	..	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52.

Code	Explanation
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range.
5	In the XM monitor, the optional status word address is not in valid user virtual address space.

Example:

See the example at the end of the .MTSET section.

.MTIN

2.4.32 .MTIN (FB and XM Monitor SYSGEN Option)

The .MTIN request is the multi-terminal form of the .TTYIN request. It does not use a queue element, but it does require an argument block. The .MTIN request moves one or more characters from the input ring buffer to the user's buffer specified by addr. The terminal must be attached and an updated user buffer address is returned in R0 if the request is successful. If bit 6 is set in the M.TSTS (see MTSET) word, the .MTIN request returns immediately with the carry bit set (code 0) if there is no input available (no line if bit 12 is clear; no characters in buffer if bit 12 is set in M.TSTS). If these conditions do not exist, the .MTIN request waits until input is available and the job is suspended until input is available.

If a multiple character request was made and the number of characters requested is not available, the request can either wait for the characters to become available, or it can return with a partial transfer. If bit 6 of M.TSTS is clear, the request waits for more characters. If bit 6 is set, the request returns with a partial transfer. In the latter case, R0 contains the updated buffer address (pointing past the last character transferred), the C bit is set, and the error code is 0.

The .MTIN request has the following form:

Macro Call: .MTIN area,addr,unit,chrcnt

where:	area	is the address of a three-word EMT argument block.
	addr	is the byte address of the user buffer.

unit — is the logical unit number of the terminal input.

chrcnt — is a character count indicating the number of characters to transfer. The valid range is from 1 to 255 (decimal).

Request Format:

R0→area:

37	2
addr	
chrcnt	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52.

Code	Explanation
0	No input available -- bit 6 is set in JSW (for system console) or M.TSTS
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range
5	In the XM monitor, the optional status word address is not in valid user virtual address space.

Example:

See the example at the end of the .MTSET section.

.MTOUT

2.4.33 .MTOUT (FB and XM Monitor SYSGEN Option)

This request is the complement to the .MTIN request. It is the multi-terminal form of the .TTYOUT request. It does not use a queue element, but does require an argument block. The .MTOUT request moves one or more characters from the user's buffer to the output ring buffer of the terminal. The terminal must be attached. An updated user buffer address is returned in R0 if the request is successful. If there is no room in the output ring buffer, the carry bit is set and an error code of 0 is returned in byte 52 if bit 6 is set in M.TSTS. Otherwise, the request waits for room and the job is suspended until room becomes available.

If a multiple character request was made and there is not enough room in the output ring buffer to transfer the requested number of characters, the request can either wait for enough room to become available, or it can return with a partial transfer. If bit 6 in M.TSTS is clear, the request waits until it can complete the full transfer. If bit 6 is set, the request returns with a partial transfer. In the latter case, R0 contains the updated buffer address (pointing past the last character transferred), the C bit is set, and the error code is 0.

The .MTOUT request has the following form:

Macro Call: .MTOUT area,addr,unit,chrcnt

where:

area	is the address of a three-word EMT argument block.
addr	is the address of the caller's input buffer.
unit	is the unit number of the terminal.
chrcnt	is a character count indicating the number of characters to transfer. The valid range is from 1 to 255 (decimal).

Request Format:

R0→area:

37	3
addr	
chrcnt	unit

Errors:

Code	Explanation
0	No room in output buffer
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range

Example:

See the example at the end of the .MTSET section.

.MTPRNT

2.4.34 .MTPRNT (FB and XM Monitor SYSGEN Option)

This request operates in a multi-terminal environment in the same way as the .PRINT request. It allows one or more lines to be printed at the specified terminal (see Section 2.4.36 for more details). The request does not return until the transfer is complete.

Macro Call: .MTPRNT area,addr,unit

where:

area	is the address of a three-word EMT argument block.
addr	is the starting address of the character string to be printed. The string must be terminated with a null byte or a 200 byte, similar to the string used with the .PRINT request. The null byte causes a carriage return/line feed combination to be printed after the string. The 200 byte suppresses the carriage return/line feed combination and leaves the carriage positioned after the last character of the string.

```
For example:  .ASCII /string/<200>
  or          .ASCIZ /string/
```

unit — is the unit number associated with the terminal.

Request Format:

R0 → area:

37	7
addr	
..	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52.

Code	Explanation
1	Illegal unit number, lun not attached
2	Non-existent lun
5	In the XM monitor*, the optional status word address is not in valid user virtual address space.

Example:

See the example at the end of the .MTSET section.

.MTRCTO

2.4.35 .MTRCTO (FB and XM Monitor SYSGEN Option)

The .MTRCTO request operates in a multi-terminal environment in the same way as the .RCTRLO request. It resets the CTRL/O switch of the specified terminal and enables terminal output.

Macro Call: .MTRCTO area,unit

where: area — is the address of a three-word EMT argument block.

unit — is the unit number associated with the terminal.

Request Format:

R0 → area:

37	4
(unused)	
..	unit

Errors:

When the carry bit is set, the following errors are returned in byte 52.

Code	Explanation
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range

Example:

See the example at the end of the .MTSET section.

.MTSET

2.4.36 .MTSET (FB and XM Monitor SYSGEN Option)

This multi-terminal request allows the user program to set terminal and line characteristics. It also determines the input/output mode of the terminal service requests for the specified terminal. This request has the following form:

Macro Call: .MTSET area,addr,unit

where:

area		is the address of a three-word EMT argument block.
addr		is the address of a four-word status block containing the line and terminal status being requested.
unit		is the logical unit number associated with the line and terminal.

The user is required to supply information to the status block. It has the following structure:

R0→area:

M.TSTS	
M.TST2	
M.FCNT	M.TFIL
M.TSTW	M.TWID

Offset	Description
0 (M.TSTS)	Terminal configuration word 1
2 (M.TST2)	Reserved for future use
4 (M.TFIL)	Character requiring fillers
5 (M.FCNT)	Number of fillers
6 (M.TWID)	Carriage width
7 (M.TSTW)	Terminal state byte

The bit definitions for the configuration word (M.TSTS) are as follows:

Mask	Bit	Meaning
1	0	Hardware tab
2	1	Output RET/LF when carriage width exceeded
4	2	Hardware form feed
10	3	Process CTRL/F and CTRL/B as normal characters
20	4	Enable escape sequence processing
40	5	Filter escape sequences
100	6	Inhibit TT wait (similar to TCBIT$ in the JSW)
200	7	XON/XOFF processing enabled
7400	8-11	Line speed (baud rate) mask (defined in full below)
10000	12	Character mode input (similar to TTSPC$ in the JSW)
20000	13	Terminal is remote (Read Only bit)
40000	14	Lower to upper case conversion disabled
100000	15	Use backspace for rubout (video type display)

Bits 8 through 11 of configuration word 1 (M.TSTS) indicate the terminal baud rate (DZ11 only). The values are as follows:

Octal Value of Line Speed Mask (M.TSTS bits 11-8)	Baud 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	(unused)

The bit definitions for M.TSTW are as follows:

Value	Bit	Meaning
2000	10	Terminal is shared console
4000	11	Terminal has hung up
10000	12	Terminal interface is DZ11
40000	14	Double CTRL/C was struck
100000	15	Terminal is acting as console (local DL11 only)

Request Format:

R0→area:

37	0
addr	
--	unit

Errors:

Code	Explanation
1	Illegal unit number, lun not attached
2	Non-existent lun
3	Illegal request, function code out of range
5	In the XM monitor, the optional status word address is not in valid user virtual address space

Example:

This program checks and changes the characteristics of a particular terminal.

```
.MCALL  .MTATCH,.MTPRNT,.MTGET,.MTIN,.MTOUT,.MTSET,.EXIT
.MCALL  .PRINT,.MTRCTO
HNGUP$  = 4000                       ;INDICATES TERMINAL IS HUNG UP
TTSPC$  = 10000                      ;SPECIAL MODE
TTLC$   = 40000                      ;LOWER CASE MODE
AS.INP  = 40000                      ;INDICATES INPUT AVAILABLE
M.TSTS  = 0                          ;TERMINAL STATUS WORD
M.TSTW  = 7                          ;TERMINAL STATE BYTE

START:
        CLR     R1                   ;INITIALIZE LUN
        MOV     #AST,R2              ;R2 -> ASYNCHRONOUS TERMINAL STATUS WORDS
10$:    .MTATCH #MTA,R2,R1           ;ATTACH TERMINAL
        BCC     20$                  ;BR IF SUCCESSFUL
        CLRB    TAI(R1)              ;SIGNAL ATTACH FAILED, DON'T CARE WHY
        BR      30$                  ;PROCEED WITH NEXT LUN
20$:    MOVB    #1,TAI(R1)           ;ATTACH SUCCESSFUL
        MOV     R1,R3                ;COPY LUN
        ASL     R3                   ;MULTIPLY BY 8 TO PRODUCE OFFSET
        ASL     R3                   ;TO THE TERMINAL STATUS BLOCK
        ASL     R3                   ;
        ADD     #TSB,R3              ;R3 -> LUN'S TSB BLOCK
        .MTGET  #MTA,R3,R1           ;GET LUN'S STATUS
        BIS     #TTSPC$+TTLC$,M.TSTS(R3)        ;SET SPECIAL MODE & LOWER CASE
        .MTSET  #MTA,R3,R1           ;SET LUN'S STATUS
        BITB    #HNGUP$/400,M.TSTW(R3)  ;TERMINAL ON LINE?
        BNE     30$                  ;BR IF HUNGUP
        .MTRCTO #MTA,R1              ;RESET CTRL/O
        .MTPRNT #MTA,#HELLO,R1       ;SAY HELLO
30$:    ADD     #2,R2                ;R2 -> NEXT AST WORD
        INC     R1                   ;NEXT LUN
        CMP     R1,#16.              ;DONE?
        BLO     10$                  ;BR IF NOT

LOOP:                                ;READ & ECHO FOREVER UNLESS ERROR OCCURS
        CLR     R1                   ;INITIALIZE LUN
        MOV     #AST,R2              ;R2 -> AST WORDS
10$:    TSTB    TAI(R1)              ;TERMINAL ATTACHED?
        BEQ     20$                  ;BR IF NOT
        BIT     #AS.INP,(R2)         ;ANY INPUT?
        BEQ     20$                  ;BR IF NOT
        .MTIN   #MTA,#MTCHAR,R1,#1 ;READ A CHAR
        BCS     ERR                  ;BR IF SOME ERROR
        .MTOUT  #MTA,#MTCHAR,R1,#1  ;OUTPUT CHAR
        BCS     ERR                  ;BR IF SOME ERROR
20$:    ADD     #2,R2                ;POINT TO NEXT AST WORD
        INC     R1                   ;NEXT LUN
        CMP     R1,#16.              ;DONE ALL
        BLO     10$                  ;BR IF NOT
        BR      LOOP                 ;REPEAT FOREVER
ERR:    .PRINT  #UNEXP               ;UNEXPECTED ERROR
        .EXIT
AST:    .BLKW   16.                  ;ASYNCHRONOUS TERMINAL STATUS WORDS
                                     ;1 PER LUN
TAI:    .BLKB   16.                  ;TERMINAL ATTACHED INDICATOR LIST
                                     ;1 BYTE PER LUN. 0=> NOT ATTACHED
        .EVEN
MTA:    .BLKW   4                    ;EMT AREA
MTCHAR: .BYTE   0                    ;INPUT STORAGE FOR CHARACTER READ
HELLO:  .ASCIZ  /HAVE A GOOD DAY/
UNEXP:  .ASCIZ  /UNEXPECTED ERROR, PROGRAM ABORTING/
        .EVEN
TSB:    .BLKW   16.*4.               ;TERMINAL STATUS BLOCKS 16. BLOCKS OF 4 WORDS
        .END    START
```

.MWAIT

2.4.37 .MWAIT (FB and XM Only)

This request is similar to the .WAIT request. .MWAIT, however, suspends execution until all messages sent by the other job have been received. It provides a means for ensuring that a required message has been processed. It should be used primarily in conjunction with the .RCVD or .SDAT modes of message handling, where no action is taken when a message is completed.

Macro Call: .MWAIT

Request Format:

R0 =		
	11	0

Errors:

None.

Example:

```
.TITLE  MWAIT.MAC
;THIS PROGRAM REQUESTS A MESSAGE, DOES SOME INTERMEDIATE PROCESSING,
;AND THE WAITS UNTIL THE MESSAGE IS ACTUALLY SENT.
        .MCALL  .MWAIT,.RCVD,.EXIT,.PRINT

WORDS=255.
START:
        .RCVD   #AREA,#RBUFF,#WORDS ;GET MESSAGE.

                                    ;INTERMEDIATE PROCESS

        MOV     #RRUFF+2,R5
        .MWAIT                      ;MAKE SURE WE HAVE IT.
        CMPB    (R5)+,#'A           ;FIRST CHARACTER AN A?
        BNE     BADMSG              ;NO, INVALID MESSAGE

        .EXIT
BADMSG: .PRINT  #MSG
        .EXIT
MSG:    .ASCIZ  /BAD MESSAGE/
AREA:   .BLKW   10
RBUFF:  .BLKW   256.
        .EVEN
        .END    START
```

.PRINT

2.4.38 .PRINT

The .PRINT request causes output to be printed at the console terminal. When a foreground job is running and a change occurs in the job producing output, a B> or F> appears. Any text following the message has been printed by the job indicated (foreground or background) until another B> or F> is printed. The string to be printed can be terminated with either a null (0) byte or a 200 byte.

If the null (ASCIZ) format is used, the output is automatically followed by a carriage return/line feed combination. If a 200 byte terminates the string, no carriage return/line feed combination is generated.

Control returns to the user program after all characters have been placed in the output buffer.

The foreground job issues a message immediately using .PRINT no matter what the state of the background job. Thus, for urgent messages, .PRINT should be used (rather than .TTYIN or .TTOUTR). The .PRINT request forces a console switch and guarantees printing of the input line. If a background job is doing a prompt and has printed an asterisk but no carriage return/line feed combination, the console belongs to the background and .TTYOUTs from the foreground are not printed until a carriage return is typed to the background. The foreground job can force its message through by doing a .PRINT instead of the .TTYOUT.

Macro Call: .PRINT addr

where: addr is the address of the string to be printed.

Errors:

None.

Example:

```
.TITLE  PRINT.MAC
        .MCALL  .PRINT,.EXIT

START:
        .PRINT  #S2
        .PRINT  #S1

        .EXIT

S1:     .ASCIZ  /THIS WILL HAVE CR-LF FOLLOWING/
S2:     .ASCII  /THIS WILL NOT HAVE CR-LF/
        .BYTE   200
        .EVEN

        .END    START
```

.PROTECT/.UNPROTECT

2.4.39 .PROTECT/.UNPROTECT (FB and XM Only)

The .PROTECT request is used by a job to obtain exclusive control of a vector (two words) in the region 0-476. If it is successful, it indicates that the locations are not currently in use by another job or by the monitor, in which case the job can place an interrupt address and priority into the protected locations and begin using the associated device.

Macro Call: .PROTECT area,addr

where: area — is the address of a three-word EMT argument block.

addr — is the address of the word pair to be protected. The argument, addr, must be a multiple of four, and must be less than 476 (octal). The two words at addr and addr+2 are protected.

Request Format:

R0 → area:

31	0
addr	

Errors:

Code	Explanation
0	Protect failure; locations already in use.
1	Address greater than 476 or not a multiple of 4.

Example:

```
.TITLE  PROTEC.MAC
;THIS EXAMPLE SHOWS THE USE OF .PROTECT TO GAIN CONTROL
;OF THE UDC11 VECTORS.
        .MCALL  .PROTECT,.PRINT,.EXIT
ST:     MOV     #AREA,-(SP)
        MOV     #234,R5             ;UDC VECTOR ADDRESS
        .PROTECT (SP),R5            ;PROTECT 234,236
        BCS     ERR                 ;YOU CAN'T
        MOV     #UDCINT,(R5)+       ;INITIALIZE THE VECTORS.
        MOV     #340,(R5)           ;AT LEVEL 7

        .EXIT
ERR:    .PRINT  #NOVEC
        .EXIT
AREA:   .BLKW   5
NOVEC:  .ASCIZ  /VECTORS ALREADY IN USE/
        .EVEN

UDCINT: RTI
        .END    ST
```

The .UNPROTECT request is the complement of the .PROTECT request. It cancels any protected vectors in the 0 to 476 area.

Macro Call: .UNPROTECT area,addr

where: area — is the address of a two-word EMT argument block

addr — is the address of the protected vector pair that is going to be cancelled.

Request Format:

R0 → area:

31	1
addr	

Errors:

Code	Explanation
1	Address (addr) is greater than 476(octal) or is not a multiple of four.

Example:

```
.TITLE  UNPRO.MAC
;.UNPROT ALLOWS ANOTHER JOB TO USE THE
;SAME VECTORS THAT WERE PREVIOUSLY
;PROTECTED WITH A .PROTECT REQUEST

        .MCALL  .PROTECT,.UNPROTECT,.EXIT,.PRINT
START:
        .PROTECT        #AREA,#234      ;PROTECT 234 & 236
        BCS     INUSE                   ;PROBABLY IN USE
                                        ;BY ANOTHER JOB

;PROGRAM NOW PROCEEDS TO USE THE VECTORS
;WHEN DONE, A .UNPROTECT IS ISSUED, FREEING
;THE VECTORS FOR USE BY ANOTHER JOB

        .UNPROTECT      #AREA,#234      ;UNPROTECT 234 & 236
        .EXIT
INUSE:  .PRINT  #ERR
        .EXIT
AREA:.BLKW      2
ERR:    .ASCIZ/PROTECT ERROR/
        .EVEN
        .END    START
```

.PURGE

2.4.40 .PURGE

The .PURGE request is used to deactivate a channel without performing a .HRESET, .SRESET, .SAVESTATUS, or .CLOSE request. It merely frees a channel without taking any other action. If a tentative file has been .ENTERed on the channel, it is discarded. Purging an inactive channel acts as a no-op.

Macro Call: .PURGE chan

Request Format:

R0 = | 3 | chan |

Errors:

None.

Example:

```
.TITLE  PURGE.MAC
;THIS EXAMPLE VERIFIES THAT CHANNEL 0-7 ARE FREE.
        .MCALL  .PURGE,.EXIT

START:
        CLR     R1                      ;START WITH CHANNEL 0
1$:     .PURGE  R1                      ;PURGE A CHANNEL
        INC     R1                      ;BUMP TO NEXT CHANNEL
        CMP     R1,#8.                  ;IS IT AT CHANNEL 8 YET?
        BLO     1$                      ;NO, KEEP GOING

        .EXIT
        .END    START
```

.QSET

2.4.41 .QSET

All RT-11 I/O transfers are done through a centralized queue management system. Each non-synchronous transfer request such as a .WRITE requires a queue element until it completes. If I/O traffic is very heavy and not enough queue elements are available, the program issuing the I/O requests can be blocked until a queue element becomes available. In an FB system, the other job can run while the first job waits for the element.

The .QSET request is used to make the RT-11 I/O queue larger (that is, to add available entries to the queue). A general rule to follow is that each program should contain one more queue element than the total number of I/O requests that will be active simultaneously on different channels. Timing and message requests such as .TWAIT and .SDAT also cause elements to be used and must be considered when allocating queue elements for a program. Note that if synchronous I/O is done (.READW/.WRITW, etc.) and no timing requests are done, no additional queue elements need be allocated.

Each time .QSET is called, a contiguous area of memory is divided into seven-word segments (10-word segments for the XM monitor) and is added to the queue for that job. .QSET can be called as many times as required. The queue set up by multiple .QSET requests is a linked list. Thus, .QSET need not be called with strictly contiguous arguments. The space used for the new elements is allocated from the user's program space. Care must be taken so that the program in no way alters the elements once they are set up. The .SRESET and .HRESET requests discard all user-defined queue elements; therefore any .QSETs must be reissued.

Care should also be taken to allocate sufficient memory for the number of queue elements requested. The elements in the queue are altered asynchronously by the monitor; if enough space is not allocated, destructive references occur in an unexpected area of memory. Other restrictions on the placement of queue elements are that the USR must not swap over them and they must not be in an overlay region. For jobs that run under the XM monitor, queue elements must be allocated in the lower 28K words of memory, since they must be accessible in kernel mapping.

NOTE

Programs that are to run in both FB and XM environments should always allocate 10 words for each queue element.

The following requests require queue elements:

.TWAIT	.WRITE
.MRKT	.WRITC
.READ	.WRITW
.READC	.SDAT
.READW	.SDATC
.RCVD	.SDATW
.RCVDC	
.RCVDW	

Macro Call: .QSET addr,len

where: addr is the address at which the new elements are to start.

len is the number of entries to be added. In the FB monitor, each queue entry is seven words long; hence the space set aside for the queue should be len*7 words. In the XM monitor, 10 words per queue element are required.

Errors:

None.

Example:

```
.TITLE  QSET.MAC
        .MCALL  .QSET,.EXIT
START:
        .QSET   #Q1,#5          ;ADD 5 ELEMENTS TO THE QUEUE
                                ;STARTING AT Q1
        .QSET   #Q3,#3          ;AND 3 MORE AT Q3.

        .EXIT

Q1:     .BLKW   7*5.            ;FIRST QUEUE AREA (35 DECIMAL WORDS)
Q3:     .BLKW   7*3.            ;SECOND QUEUE AREA (21 DECIMAL WORDS)

        .END    START
```

.RCTRLO

2.4.42 .RCTRLO

The .RCTRLO request ensures that the console terminal is able to print. Since CTRL/O struck while output is directed to the console terminal inhibits the output from printing until either another CTRL/O is struck or until the program resets the CTRL/O switch, a program that has a message that must appear at the console should reset the CTRL/O switch.

Macro Call: .RCTRLO

Errors:

None.

Example:

```
.TITLE  RCTRLO.MAC
;IN THIS EXAMPLE, THE USER PROGRAM FIRST CALLS THE CSI IN GENERAL MODE,
;THEN PROCESSES THE COMMAND. WHEN FINISHED, IT RETURNS TO THE CSI FOR
;ANOTHER COMMAND LINE. TO MAKE SURE THAT THE PROMPTING '*' TYPED BY
;THE CSI IS NOT INHIBITED BY A CTRL O IN EFFECT FROM THE LAST OPERATION,
;TERMINAL OUTPUT IS RE-ENABLED VIA A .RCTRLO COMMAND PRIOR TO THE
;CSI CALL.
        .MCALL  .RCTRLO,.CSIGEN,.EXIT

START:  .RCTRLO                         ;MAKE SURE TT OUTPUT IS
                                        ;ENABLED
        .CSIGEN #DSPACE,#DEXT,#0        ;CALL CSI-IT WILL TYPE
                                        ;"*"

                                        ;PROCESS COMMAND

        JMP     START                   ;GET NEXT COMMAND

DEXT:   0                               ;NO DEFAULT EXTENSIONS
        0
        0
        0
DSPACE: .=.+400                 ;HANDLER SPACE

        .END    START
```

.RCVD/.RCVDC/.RCVDW

2.4.43 .RCVD/.RCVDC/.RCVDW (FB and XM Only)

There are three forms of the receive data request; these are used in conjunction with the .SDAT (Send Data) requests to allow a general data/message transfer system for communication between a foreground and a background job. .RCVD requests can be thought of as .READ requests where data transfer is not from a peripheral device but from the other job in the system. Additional queue elements should be allocated for buffered I/O operations in .RCVD and .RCVDC requests (see .QSET).

.RCVD

This request is used to receive data and continue execution. The request is posted and the issuing job continues execution. At some point when the job needs to have the transmitted message, an .MWAIT should be executed. This causes the job to be suspended until the message has been received.

Macro Call: .RCVD area,buf,wcnt

where:

area	is the address of a five-word EMT argument block.
buf	is the address of the buffer to which the message is to be sent.
wcnt	is the number of words to be transferred.

Request Format:

R0→area:

26	0
(unused)	
buf	
wcnt	
1	

Word 0 (the first word) of the message buffer contains the number of words transmitted whenever the .RCVD is complete. Thus, the space allocated for the message should always be at least one word larger than the actual message size expected.

The word count is a variable number, and as such, the .SDAT/.RCVD combination can be used to transmit a few words or entire buffers. The .RCVD operation is only complete when a .SDAT is issued from the other job.

Programs using .RCVD/.SDAT must be carefully designed to either always transmit/receive data in a fixed format or to have the capability of handling variable formats. The messages are all processed in first in/first out order. Thus, the receiver must be certain it is receiving the message it actually wants.

Errors:

Code	Explanation
0	No other job exists in the system.

Example:

An example follows the .RCVDW section.

.RCVDC

The .RCVDC request receives data and enters a completion routine when the message is received. The .RCVDC request is posted and the issuing job continues to execute. When the other job sends a message, the completion routine specified is entered.

Macro Call: RCVDC area,buf,wcnt,crtn

where:

area	is the address of a five-word EMT argument block.
buf	is the address of the buffer to which the message is to be sent.

wcnt	is the number of words to be transmitted.
crtn	is the address of a completion routine to be entered.

As in .RCVD word 0 of the buffer contains the number of words transmitted when the transfer is complete.

Request Format:

R0 → area:

26	0
(unused)	
buf	
wcnt	
crtn	

Errors:

Code	Explanation
0	No other job exists in the system.

Example:

An example follows the .RCVDW section.

.RCVDW

.RCVDW is used to receive data and wait. A message request is posted and the job issuing the request is suspended until the other job sends a message to the issuing job. When the issuing job runs again, the message has been received, and word 0 of the buffer indicates the number of words transmitted.

Macro Call: .RCVDW area,buf,wcnt

where:	area	is the address of a five-word EMT argument block.
	buf	is the address of the buffer to which the message is to be sent.
	wcnt	is the number of words to be transmitted.

Request Format:

R0 → area:

26	0
(unused)	
buf	
wcnt	
0	

Errors:

Code	Explanation
0	No other job exists in the system.

Example:

```
.TITLE  RCVDWF.MAC
;THIS IS THE FOREGROUND JOB WHICH SENDS THE DATA TO THE RCVDW EXAMPLE.
        .MCALL  .SDATW,.PRINT,.EXIT

START:  MOV     #AREA,R5
        .SDATW  R5,#BUFR,#4      ;SEND 4 WORD FILE NAME
        BCC     1$
        .PRINT  #NJMSG
1$:     .EXIT
AREA:   .BLKW   5
BUFR:   .RAD50  /DK TEST  TMP/
NJMSG:  .ASCIZ  /NO B JOB/
        .EVEN
        .END    START

.TITLE  RCVD.MAC
;IN THIS EXAMPLE, THE RUNNING JOB RECEIVES A MESSAGE FROM THE
;SECOND JOB AND INTERPRETS IT AS THE DEVICE AND FILENAME OF A FILE
;TO BE OPENED AND USED. IN THIS CASE, THE MESSAGE WAS IN .RAD50 FORMAT,
;AND THE RECEIVING PROGRAM DID NOT USE THE TRANSMITTED LENGTH FOR ANY
;PURPOSE. THE ISSUING JOB IS SUSPENDED UNTIL THE INDICATED DATA
;IS TRANSMITTED. EITHER OF THE OTHER MODES COULD HAVE ALSO BEEN USED TO
;RECEIVE THE MESSAGE.
        .MCALL  .RCVDW,.PURGE,.LOOKUP,.EXIT,.PRINT

START:
        MOV     #AREA,R5         ;R5=EMT ARG. AREA
        .RCVDW  R5,#FILE,#4      ;REQUEST MESSAGE AND WAIT
        BCS     MERR             ;AN ERROR?
        .PURGE  #0               ;CLEAR CHANNEL 0
        .LOOKUP R5,#0,#FILE+2    ;LOOKUP INDICATED FILE
        BCS     LKERR            ;ERROR

        .EXIT

AREA:   .BLKW   10               ;LEAVE SPACE FOR SAFETY
FILE:   .BLKW   1                ;ACTUAL WORD COUNT IS HERE
        .BLKW   4                ;DEV:FILE.EXT ARE HERE

MERR:   .PRINT  #MMSG
        .EXIT
LKERR:  .PRINT  #LKMSG
        .EXIT
MMSG:   .ASCIZ  /MESSAGE ERROR/
LKMSG:  .ASCIZ  /LOOKUP ERROR/
        .EVEN
        .END    START
```

.READ/.READC/.READW

2.4.44 .READ/.READC/.READW

RT-11 provides three modes of I/O: .READ/.WRITE, .READC/.WRITC, and .READW/.WRITW.

In the case of .READ and .READC, additional queue elements should be allocated for buffered I/O operations (see .QSET).

NOTE

Upon return from any .READ, .READC or .READW programmed request, R0 contains the number of words requested if the read is from a sequential-access device (for example, paper tape). If the read is from a random-access device (disk or DECtape) R0 contains the actual number of words that will be read (.READ or .READC) or have been read (.READW). This number is less than the requested word count if an attempt is made to read past end-of-file, but a partial transfer is possible. In the case of a partial transfer, no error is indicated if a read request is shortened. Therefore, a program should always use the returned word count as the number of words available. For example, suppose a file is five blocks long (it has block numbers 0 to 4) and a request is issued to read 512 words, starting at block 4. Since 512 words is two blocks, and block 4 is the last block of the file, this is an attempt to read past end-of-file. The monitor detects this fact and shortens the request to 256 words. On return from the request, R0 contains 256 indicating that a partial transfer occurred. Also note that since the request is shortened to an exact number of blocks, a request for 256 words either succeeds or fails, but cannot be shortened.

An error is reported if a read is attempted with a block number that is beyond the end of file. The carry bit is set, and error code 0 appears in byte 52. No data is transferred, in this case. R0 contains a zero.

.READ

The .READ request transfers a specified number of words from the device associated with the specified channel to memory. The channel is associated with the device when a .LOOKUP or .ENTER request is executed. Control returns to the user program immediately after the .READ is initiated, possibly before the transfer is completed. No special action is taken by the monitor when the transfer is completed.

Macro Call: .READ area,chan,buf,wcnt,blk

where:

	area	is the address of a five-word EMT argument block.
	chan	is a channel number in the range 0-377 (octal).
	buf	is the address of the buffer to receive the data read.

wcnt — is the number of words to be read.

blk — is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. The user program normally updates blk before it is used again. If blk=0, TT: issues an uparrow (^) prompt and LP: issues a form feed. (This is true for all .READ and .WRITE requests.)

The .READ and .READC requests perform the following operations:

1. Tell the monitor to do a read from the device and immediately return to the caller .

2. Execute as soon as all previous I/O requests to the device handlers have been completed. Note that a read from RK1: must wait for a previous read to RK0: to complete. This is a hardware restriction because the controller looks at all I/O operations sequentially.

3. Return read errors on the return from the .READ and .READC or the .WAIT request. Errors can occur on the read or on the wait, but only one error is returned. Therefore, the program must check for an error when the read is complete. The wait request returns an error, but it doesn't indicate which read caused the error.

4. During the .READ and .READC requests, the monitor keeps track of errors in the channel status word. If an error occurs before the monitor can return to the caller, the error is reported on the return from the read request with the carry bit set and the error values in R0. If the error occurs after return from the read request, the error is reported on return from the next .WAIT, or the next .READ/.READC. Some errors can be returned from .READ/.READC requests immediately, before any I/O operation takes place. One condition that causes an immediate error return is an attempt to read beyond end-of-file.

5. Errors reported on the return from the read request are:

   a. Non-existent device/unit

   b. Non-existent block

   c. In general, errors that don't require data tranfers but are controller errors or EOF errors.

Request Format:

R0→area:		
	10	chan
	blk	
	buf	
	wcnt	
	1	

When the user program needs to access the data read on the specified channel, a .WAIT request should be issued. This ensures that the data has been read completely. If an error occurred during the transfer, the .WAIT request indicates the error.

Errors:

Code	Explanation
0	Attempt to read past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.READC

The .READC request transfers a specified number of words from the indicated channel to memory. Control returns to the user program immediately after the .READC is initiated. Attempting to read past end-of-file also causes an immediate return, in this case with the carry bit set and the error byte set to 0. Execution of the user program continues until the .READC is complete, then control passes to the routine specified in the request. When an RTS PC is executed in the completion routine, control returns to the user program.

Macro Call: .READC area,chan,buf,wcnt,crtn,blk

where:

- area — is the address of a five-word EMT argument block.
- chan — is a channel number in the range 0-377 (octal).
- buf — is the address of the buffer to receive the data read.
- wcnt — is the number of words to be read.
- crtn — is the address of the user's completion routine. The address of the completion routine must be above 500 (octal).
- blk — is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. The user program normally updates blk before it is used again.

When a completion routine is called, error or end-of-file information for a channel is not cleared. The next .WAIT or .READ/.READC on the channel (from either mainline code or a completion routine) produces an immediate return with the C bit set and the error code in byte 52.

Request Format:

R0→area:

10	chan
blk	
buf	
wcnt	
crtn	

When entering a .READC completion routine the following are true:

1. R0 contains the contents of the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer; consequently, the data may not be reliable. The end-of-file bit may be set.

2. R1 contains the channel number of the operation. This is useful when the same completion function is to be used for several different transfers.

3. On a file-structured transfer, a shortened read is reported at the return from the .READC request, not when the completion routine is called.

4. Registers R0 and R1 can be used by the routine, but all other registers must be saved and restored. Data cannot be passed between the main program and completion routines in any register or on the stack.

Errors:

Code	Explanation
0	Attempt to read past end-of-file -- no data was read
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.READW

The .READW request transfers a specified number of words from the indicated channel to memory. Control returns to the user program when the .READW is complete and/or an error is detected.

Macro Call: .READW area,chan,buf,wcnt,blk

where:

area	is the address of a five-word EMT argument block.
chan	is a channel number in the range 0-377 (octal).
buf	is the address of the buffer to receive the data read.
wcnt	is the number of words to be read -- each .READ request can transfer a maximum of 32K words.
blk	is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. The user program normally updates blk before it is used again.

Request Format:

R0→area:

10	chan
blk	
buf	
wcnt	
0	

On return from this call, a hardware error has occurred if the carry bit is set. If no error occurred, the data is in memory at the specified address. In an FB environment, the other job can be run while the issuing job is waiting for the I/O to complete.

Errors:

Code	Explanation
0	Attempt to read past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.RENAME

2.4.45 .RENAME

The .RENAME request causes an immediate change of name of the file specified and gives that file the current date in its directory entry. An error occurs if the channel specified is already open.

Macro Call: .RENAME area,chan,dblk

where: area — is the address of a two-word EMT argument block.

chan — a channel number in the range 0-377 (octal)

dblk — a block number specifying the relative file where an I/O transfer is to begin.

Request Format:

R0→area:

4	chan
dblk	

The dblk argument consists of two consecutive Radix-50 device and file specifications. For example:

```
        .RENAME   #AREA,#7,#DBLK      ;USE CHANNEL 7
        BCS       RNMERR              ;NOT FOUND
        .
        .
        .
DBLK:   .RAD50    /DT3/
        .RAD50    /OLDFIL/
        .RAD50    /MAC/
        .RAD50    /DT3/
        .RAD50    /NEWFIL/
        .RAD50    /MAC/
```

The first string represents the file to be renamed and the device where it is stored. The second represents the new file name. If a file with the same name as the new file name specified already exists on the indicated device, it is deleted. The second occurrence of the device name DT3 is necessary for proper operation, and should not be omitted. The specified channel is left inactive when the .RENAME is complete. .RENAME requires that the handler to be used be resident at the time the .RENAME request is made. If it is not, a monitor error occurs. Note that .RENAME is legal only on files on block-replaceable devices (disks and DECtape). In magtape operations, the handler returns an illegal operation code in byte 52 if a .RENAME request is attempted. (.RENAMEs to other devices are ignored.)

Errors:

Code	Explanation
0	Channel open
1	File not found
2	Illegal Operation

Example:

```
.TITLE  RENAME.MAC
;IN THIS EXAMPLE, THE FILE "DATA.TMP" ON DT0: IS RENAMED TO "DATA.001".
        .MCALL  .FETCH,.PRINT
        .MCALL  .EXIT,.RENAME

START:  .FETCH  #HSPACE,#NAMBLK ;GET HANDLER
        BCS     FERR            ;SOME ERROR
        .RENAME #AREA,#0,#NAMBLK ;DO THE RENAME
        BCS     RNMERR          ;ERROR
        .EXIT
FERR:   .PRINT  #FMSG
        .EXIT
RNMERR: .PRINT  #RNMSG
        .EXIT
AREA:   .BLKW   5               ;ROOM FOR ARGS.
NAMBLK: .RAD50  /DT0DATA  TMP/  ;OLD NAME
        .RAD50  /DT0DATA  001/  ;NEW NAME
FMSG:   .ASCIZ  /FETCH?/        ;ERROR MESSAGES
RNMSG:  .ASCIZ  /RENAME?/
        .EVEN
HSPACE=.

        .END    START
```

.REOPEN

2.4.46 .REOPEN

The .REOPEN request reassociates the specified channel with a file on which a .SAVESTATUS was performed. The .SAVESTATUS/.REOPEN combination is useful when a large number of files must be operated on at one time. As many files as are needed can be opened with .LOOKUP, and their status preserved with .SAVESTATUS. When data is required from a file, a .REOPEN enables the program to read from the file. The

.REOPEN need not be done on the same channel as the original .LOOKUP and .SAVESTATUS.

Macro Call: .REOPEN area,chan,cblk

where:

area		is the address of a two-word EMT argument block.
chan		is a channel number in the range 0-377 (octal).
cblk		is the address of the five-word block where the channel status information was stored.

Request Format:

R0 → area:	6	chan
	cblk	

Errors:

Code	Explanation
0	The specified channel is in use. The .REOPEN has not been done.

Example:

Refer to the example following the description of .SAVESTATUS.

.SAVESTATUS

2.4.47 .SAVESTATUS

The .SAVESTATUS request stores five words of channel status information into a user-specified area of memory. These words contain all the information RT-11 requires to completely define a file. When a .SAVESTATUS is done, the data words are placed in memory, the specified channel is freed, and the file is closed. When the saved channel data is required, the .REOPEN request is used.

.SAVESTATUS can only be used if a file has been opened with .LOOKUP. If .ENTER was used, .SAVESTATUS is illegal and returns an error. Note that .SAVESTATUS is not legal only on magtape or cassette files.

The .SAVESTATUS/.REOPEN requests jointly are used to open many files on a limited number of channels or to allow all .LOOKUPs to be done at once to avoid USR swapping.

While the .SAVESTATUS/.REOPEN combination is very useful, care must be observed when using it. In particular, the following cases should be avoided:

1. If a .SAVESTATUS is performed and the same file is then deleted before it is reopened, it becomes available as an empty space that could be used by the .ENTER command. If this sequence occurs, the contents of the file supposedly saved changes.

2. Although the device handler for the required peripheral need not be in memory for execution of a .REOPEN, the handler must be in memory when a .READ or .WRITE is executed, or a fatal error is generated.

One of the more common uses of .SAVESTATUS and .REOPEN is to consolidate all directory access motion and code at one place in the program. All files necessary are opened and their status saved, then they are re-opened one at a time as needed. USR swapping can be minimized by .LOCKing in the USR, doing .LOOKUPs as needed, using .SAVESTATUS to save the file data, and then .UNLOCKing the USR. The user should be aware of the consequences of locking in the USR in a foreground/background environment. If the background job locks in the USR when the foreground job requires it, the foreground job is delayed until the background job unlocks the USR.

Macro Call: .SAVESTATUS area,chan,cblk

where:

area	is the address of a two-word EMT argument block.
chan	is a channel number in the range 0-377 (octal).
cblk	is the address of the five-word user memory block where the channel status information is to be stored.

Request Format:

R0→area:

5	chan
cblk	

Errors:

Code	Explanation
0	The channel specified is not currently associated with any files; that is, a previous .LOOKUP on the channel was never done.
1	The file was opened via .ENTER, or is a magtape or cassette file, and a .SAVESTATUS is illegal.

Example:

```
        .TITLE  SAVEST.MAC
;ONE OF THE MORE COMMON USES OF .SAVESTATUS AND .REOPEN IS TO CONSOLIDATE
;ALL DIRECTORY ACCESS MOTION AND CODE AT ONE PLACE IN THE PROGRAM. ALL
;FILES NECESSARY ARE OPENED AND THEIR STATUS SAVED, THEN THEY ARE RE-OPENED
; ONE AT A TIME AS NEEDED. USR SWAPPING CAN BE MINIMIZED BY LOCKING IN THE
;USR, DOING .LOOKUP'S AS NEEDED, USING .SAVESTATUS TO SAVE THE FILE DATA,
;AND THEN UNLOCKING THE USR. IN THIS EXAMPLE THREE INPUT
;FILES ARE SPECIFIED IN THE COMMAND STRING;THESE ARE THEN PROCESSED ONE AT A TIME.
        .MCALL  .CSIGEN,.SAVESTATUS,.REOPEN
        .MCALL  .READ,.EXIT

START:  MOV     #AREA,R5
        .CSIGEN #DSPACE,#DEXT   ;GET INPUT FILES

        MOV     R0,BUFF         ;SAVE POINTER TO FREE CORE

        .SAVESTATUS R5,#3,#BLUCK1 ;SAVE FIRST INPUT FILE
        .SAVESTATUS R5,#4,#BLOCK2 ;SAVE SECOND FILE
        .SAVESTATUS R5,#5,#BLOCK3 ;SAVE THIRD FILE


        MOV     #BLOCK1,R4
PROCESS: .REOPEN R5,#0,R4       ;REOPEN FILE ON
                                ;CHANNEL 0

        .READ   R5,#0,BUFF,COUNT,BLOCK ;PROCESS FILE ON CHANNEL 0

DONE:   ADD     #12,R4          ;POINT TO NEXT SAVESTATUS BLOCK
        CMP     R4,#BLOCK3      ;LAST FILE RROCESSED?
        BLOS    PROCESS         ;NO - DO NEXT
        .EXIT

BLOCK1: .WORD   0,0,0,0,0       ;MEMORY BLOCKS FOR
BLOCK2: .WORD   0,0,0,0,0       ;SAVESTATUS INFORMATION
BLOCK3: .WORD   0,0,0,0,0
AREA:   .BLKW   10

BUFF:   .WORD   0
BLOCK:  .WORD   0
COUNT:  .WORD   256.

DEXT:   .WORD   0,0,0,0
DSPACE=.
        .END    START
```

.SCCA

2.4.48 .SCCA

The .SCCA request provides the following functions:

1. Inhibition of CTRL/C abort
2. Indication that a double CTRL/C was initiated at the keyboard.
3. Ability to distinguish between single and double CTRL/C commands.

This request intercepts and temporarily inhibits a console CTRL/C command, preventing the job from being aborted. CTRL/C characters are placed in the input ring buffer and are treated as normal control characters without specific system functions. The request requires a status word address that is used to report double CTRL/C input sequences. Bit 15 of the status word is set when consecutive CTRL/C characters are detected. The program must clear the bit.

There are two cautions to observe when using .SCCA. First, the request can cause CTRL/C to appear in the terminal input stream, and therefore the program must provide a way to handle it. Second, the request makes it impossible to terminate program loops from the console, and therefore it should be used only in thoroughly tested, reliable programs. When .SCCA causes an interminable program loop, the system must be halted and re-bootstrapped.

A .SCCA request with a status word address of zero disables the intercept and re-enables CTRL/C system action.

Macro Call: .SCCA area,addr

where:

area	is the address of a two-word parameter block.
addr	is the address of a terminal status word (an address of 0 re-enables the CTRL/C command).

Request Format:

R0 → area:

35	0
addr	

Errors:

None.

Example:

This example intercepts CTRL/C characters with the .SCCA request.

```
.TITLE  SCCA.MAC
        .MCALL  .SCCA,.PRINT,.TTYIN,.TTYOUT,    .EXIT

JSW     = 44                            ;JOB STATUS WORD
TTSPC$  = 10000                         ;SPECIAL MODE BIT
CTRL.C  = 3                             ;ASCII CTRL/C

START:: MOV     #SCCA,R1                ;R1 -> CTRL/C INDICATOR
        .SCCA   #AREA,R1                ;INTERCEPT CTRL/C
        BIS     #TTSPC$,@#JSW           ;SET SPECIAL MODE
        CLR     @R1                     ;CLEAR INDICATOR
        .PRINT  #HELLO                  ;TELL USER WE'RE RUNNING
10$:    .TTYIN                          ;READ A CHAR
        CMPB    R0,#CTRL.C              ;DID WE GET AN CTRL/C?
        BEQ     20$                     ;BRANCH IF SO
        .TTYOUT                         ;WRITE A CHAR
        BR      10$                     ;LOOP

20$:    .TTYOUT                         ;PRINT THE CTRL/C
        .PRINT  #CTRLC1                 ;TELL USER ABOUT IT
30$:    TST     @R1                     ;DOUBLE CTRL/C?
        BNE     40$                     ;BRANCH IF SO
        .TTYIN                          ;READ A CHARACTER
        .TTYOUT                         ;WRITE A CHARACTER
        BR      30$                     ;LOOP UNTIL DOUBLE CTRL/C STRUCK

40$:    .SCCA   #AREA,#0                ;LET MONITOR HANDLE CTRL/C'S
        .PRINT  #CTRLC2                 ;INDICATE DOUBLE CTRL/C STRUCK
        .EXIT                           ; AND QUIT.

SCCA:   .BLKW   1                       ;CTRL/C INTERRUPT WORD
AREA:   .BLKW   2                       ;EMT AREA
HELLO:  .ASCII  \SCCA EXAMPLE PROGRAM\<15><12>
        .ASCIZ  \TYPE CTRL/C PLEASE\
CTRLC1: .ASCII  \THANK YOU, CTRL/C WAS DETECTED\<15><12>
        .ASCII  \TYPE <RET>, FOLLOWED BY DOUBLE \
        .ASCIZ  \CTRL/C TO ABORT PROGRAM\
        .EVEN
CTRLC2: .ASCIZ  \DOUBLE CTRL/C STRUCK\

        .END    START
```

.SDAT/.SDATC/.SDATW

2.4.49 .SDAT/.SDATC/.SDATW (FB and XM Only)

These requests are used in conjunction with the .RCVD/.RCVDW/.RCVDC calls to allow message transfers between a foreground job and a background job under the FB or XM monitors. .SDAT transfers can be considered similar to .WRITE requests in which data transfer is not to a peripheral, but from one job to another. Additional I/O queue elements should be allocated for buffered I/O operations in .SDAT and .SDATC requests (see .QSET).

.SDAT

Macro Call: .SDAT area,buf,wcnt

where:

- area — is the address of a five-word EMT argument block.
- buf — is the buffer address of the beginning of the message to be transferred.
- wcnt — is the number of words to transfer.

Request Format:

R0 → area:

25	0
(unused)	
buf	
wcnt	
1	

Errors:

Code	Explanation
0	No other job exists.

Example:

See the example following .SDATW.

.SDATC

Macro Call: .SDATC area,buf,wcnt,crtn

where:

- area — is the address of a five-word EMT argument block.
- buf — is the buffer address of the beginning of the message to be transferred.
- wcnt — is the number of words to transfer.
- crtn — is the address of the completion routine to be entered when the message has been transmitted.

Request Format:

R0 → area:		
	25	0
	(unused)	
	buf	
	wcnt	
	crtn	

Errors:

Code	Explanation
0	No other job exists.

Example:

See the example following .SDATW.

.SDATW

Macro Call: .SDATW area,buf,wcnt

where:

- area — is the address of a five-word EMT argument block.
- buf — is the buffer address of the beginning of the message to be transferred.
- wcnt — is the number of words to transfer.

Request Format:

R0 → area:		
	25	0
	(unused)	
	buf	
	wcnt	
	0	

Errors:

Code	Explanation
0	No other job exists.

Example:

```
.TITLE  SDATWF.MAC
;THIS IS THE FOREGROUND HALF OF THE SDAT.MAC EXAMPLE.
;ONCE THE DATA IS RECEIVED, A MESSAGE INDICATING A POSITIVE ACKNOWLEDGEMENT
;IS SENT TO THE BACKGROUND.
        .MCALL  .SDATW,.RCVDW,.EXIT,.PRINT

START:  MOV     #AREA,R5
        .RCVDW  R5,#BUFR1,#176
        BCS     NJERR
        .SDATW  R5,#BUFR2,#2
        BCS     NJERR
        .EXIT
AREA:   .BLKW   5
NJERR:  .PRINT  #NJMSG
        .EXIT
NJMSG:  .ASCIZ  /NO BACKGROUND JOB?/
        .EVEN
BUFR1:  .BLKW   200
BUFR2:  .ASCII  /YES./
        .EVEN
        .END    START

.TITLE  SDAT.MAC
;IN THIS EXAMPLE, THE JOB FIRST SENDS A MESSAGE INTERROGATING THE OTHER
;JOB ABOUT THE STATUS OF AN OPERATION, AND THEN LOOKS FOR AN ACKNOWLEDGEMENT
;FROM THE JOB.
        .MCALL  .SDAT,.RCVD,.MWAIT,.PRINT,.EXIT

START:
        MOV     #AREA,R5            ;SET UP EMT BLOCK
        .SDAT   R5,#SBUFF,#MLGTH    ;ASK HIM A QUESTION
        BCS     NOJOB               ;NO OTHER JOB AROUND!

                                    ;MISCELLANEOUS PROCESSING

        .RCVD   R5,#BUFF2,#20.      ;RECEIVE 20 DECIMAL WORDS
        .MWAIT                      ;WAIT FOR ACKNOWLEDGE.
        MOV     #BUFF2+2,R1         ;POINT TO ACTUAL ANSWER.
        CMPB    (R1)+,#'Y           ;IS FIRST WORD Y FOR YES?
        BNE     PRNEG               ;NEGATIVE ACKNOWLEDGE
        .PRINT  #POSACK

        .EXIT

PRNEG:  .PRINT  #NEGACK             ;NEGATIVE ON OUR INQUIRY
        .EXIT
SBUFF:  .ASCII  /IS THE REQUIRED PROCESS GOING?/
MLGTH=.-SBUFF
BUFF2:  .WORD   0                   ;ACTUAL LENGTH IS HERE
        .BLKW   20.                 ;SPACE FOR 20. WORDS

NOJOB:  .PRINT  #NJMSG
        .EXIT
NEGACK: .ASCIZ  /NEGATIVE ACKNOWLEDGE/
POSACK: .ASCIZ  /POSITIVE ACKNOWLEDGE/
NJMSG:  .ASCIZ  /NO JOB/
        .EVEN
AREA:   .BLKW   10.

        .END    START
```

2.4.50 .SETTOP

The .SETTOP request allows the user program to request that a new address be specified as a program's upper limit. The monitor determines whether this address is legal and whether or not a memory swap is necessary when the USR is required. For instance, if the program specified an upper limit below the start address of USR (normally specified in offset 266), no swapping is necessary, as the USR is not overlaid. If .SETTOP from the background specifies a high limit greater than the address of the USR and a SET USR NOSWAP command has not been given, a memory swap is required. Section 2.2.5 gives details on determining where the USR is in memory and how to optimize the .SETTOP.

Careful .SETTOP usage provides a significant improvement in the performance of the user program. The following outline is a sample. Several of the system supplied programs use a similar approach.

A .SETTOP is done to the high limit of the code in a program before buffers or work areas are allocated. If the program is aborted, minimal writing of the user program occurs. However, the program is allowed to be restarted successfully.

A user command line is now read through .CSISPC or .GTLIN. An appropriate USR swap address is set in location 46. Successive .DSTATUS, .SETTOP and .FETCH requests are performed to load necessary device handlers. This attempts to keep the USR resident as long as possible during this procedure.

Buffers and work areas are allocated as needed, being sure to issue appropriate .SETTOP requests to account for their size. Frequently, a .SETTOP of #-2 is performed to request all available memory to be given to the program. This can be more useful than keeping the USR resident.

If the process has a well defined closing phase, another .SETTOP can be issued to cause the USR to become resident again to close files (the user should remember to set location 46 to zero if this is done, so that the USR again swaps in the normal area).

The program is now ready to cycle at the start of this procedure.

On return from .SETTOP, both R0 and the word at location 50 (octal) contain the highest memory address allocated for use. If the job requested an address higher than the highest address legal for the requesting job, the address returned is the highest legal address for the job rather than the requested address.

When doing a final exit from a program, the monitor writes the program to the file SWAP.SYS and then reads in the KMON. A .SETTOP #0 at exit time prevents the monitor from swapping out the program before reading in the KMON, thus saving time. This procedure is especially useful on a diskette system when indirect command files are used to run a sequence of programs.

Macro Call: .SETTOP addr

where: addr is the address of the highest word of the free area desired.

Notes:

1. A program should never do a .SETTOP and assume that its new upper limit is the address it requested. It must always examine the returned contents of R0 or location 50 to determine its actual high address.

2. It is imperative that the value returned in R0 or location 50 be used as the absolute upper limit. If this value is exceeded, vital parts of the monitor can be destroyed.

Errors:

None.

Example:

```
.TITLE  SETTOP.MAC
;THIS IS AN EXAMPLE IN TWO PARTS. THE FIRST INDICATES HOW A SMALL
;BACKGROUND JOB (I.E.,ONE WITH FREE SPACE BETWEEN ITSELF AND THE USR)
;CAN BE ASSURED OF RESERVING SPACE UP TO BUT NOT INCLUDING THE USR.
;THIS IN EFFECT GIVES THE JOB ALL THE SPACE IT CAN WITHOUT CAUSING
;THE USR TO BECOME NON-RESIDENT.
;THE SECOND PART INDICATES HOW TO ALWAYS RESERVE THE MAXIMUM AMOUNT
;OF SPACE BY MAKING THE USR NON-RESIDENT.
        .MCALL  .SETTOP,.EXIT,.GVAL

START:
;PART 1
RMON    = 54                            ;POINTER TO START OF RESIDENT
USRLOC  = 266                           ;OFFSET FROM RESIDENT TO POINTER
                                        ;WHERE USR WILL START.

        .GVAL   #AREA,#USRLOC           ;R0 -> USR
        TST     -(R0)                   ;POINT TO HIGHEST WORD NOT IN USR
        .SETTOP                         ;AND ASK FOR IT
        MOV     R0,HICORE               ;R0 CONTAINS THE HIGH ADDRESS
                                        ;THAT WAS RETURNED.

;PART 2
        .SETTOP #-2                     ;IF WE ASK FOR A VALUE GREATER
                                        ;THAN START OF RESIDENT, WE
                                        ;WILL GET BACK THE ABSOLUTELY
                                        ;HIGHEST USABLE ADDRESS.
        MOV     R0,HICORE               ;THAT IS OUR LIMIT NOW

        .EXIT
HICORE: .WORD   0
AREA:   .BLKW   2                       ;EMT AREA BLOCK
        .END    START
```

If a SET USR NOSWAP command is executed, the USR cannot be made non-resident. In this case, in both 1 and 2 above, R0 would return a value just below the USR.

Caution should be used concerning technique 1, above. If the background program is so large that the USR is normally positioned over part of it, the high limit value returned by the .SETTOP can actually be lower than the program's original high limit determined at LINK time. The USR is then resident, with a portion of the user program destroyed. When the USR is resident, that portion of the user program is swapped out (no longer in memory) but it is reloaded automatically when the USR is no longer required.

.SFPA

2.4.51 .SFPA

.SFPA allows users with floating point hardware to set trap addresses to be entered when a floating point exception occurs. If no user trap address is specified and a floating point (FP) exception occurs, a ?MON-F-FPU trap occurs, and the job is aborted.

Macro Call: .SFPA area,addr

where: area is the address of a two-word EMT argument block.

addr is the address of the routine to be entered when an exception occurs.

Request Format:

R0 → area:	30	0
	addr	

Notes:

1. If the address argument is 0, user floating point routines are disabled and the fatal ?MON-F-FPU trap error is produced.

2. In the FB environment, an address value of 1 indicates that the FP registers should be switched when a context switch occurs, but no user traps are enabled. This allows both jobs to use the FP unit. An address of 1 to the SJ monitor is equivalent to an address of 0.

3. When the user routine is activated, it is necessary to re-execute an .SFPA request, as the monitor inhibit user traps when any one is serviced. It does this to prevent a possible infinite loop from being set up by repeated FP exceptions.

4. If the FP11 is being used, the instruction STST -(SP) is executed by the monitor before entering the user's trap routine. Thus, the trap routine must pop the two status words off the stack before doing an RTI. The program can tell if FP hardware is available by examining the configuration word in the monitor.

Errors:

None.

Example:

```
.TITLE  SFPA.MAC
;THIS EXAMPLE SETS UP A USER FP TRAP ADDRESS.
        .MCALL  .SFPA,.EXIT

START:

        .SFPA   #AREA,#FPTRAP

        .EXIT

FPTRAP:

        MOV     R0,-(SP)        ;R0 USED BY .SFPA
        .SFPA   #AREA,#FPTRAP
        MOV     (SP)+,R0        ;RESTORE R0
        RTI

AREA:   .BLKW   10
        .END    START
```

.SPFUN

2.4.52 .SPFUN

This request is used with cassette and magtape handlers to do device-dependent functions, such as rewind and backspace, on those devices. It can be used with diskettes and some disks to allow reading and writing of absolute sectors. This request can determine the size of a volume mounted in a particular device unit for RX02 diskettes, RK06/07 disks, and RL01 disks.

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn]

where:

- area — is the address of a six-word EMT argument block.
- chan — is a channel number in the range 0 to 377 (octal).
- func — is the numerical code of the function to be performed.
- buf — is the buffer address; this parameter must be set to zero if no buffer is required.
- wcnt — is defined in terms of the device handler associated with the specified channel and in terms of the specified special function code.
- blk — is also defined in terms of the device handler associated with the specified channel and in terms of the specified special function code.
- crtn — is the entry point of a completion routine. If left blank, 0 is automatically inserted. This value is the same as for .READ, .READC and .READW.

  0 = wait I/O (.READW)
  1 = real time (.READ)

  Value >500 = completion routine

Request Format:

R0 → area:

32	chan
blk	
buf	
wcnt	
func	377
crtn	

The chan, blk and wcnt arguments are the same as those defined for .READ/.WRITE requests. They are only required when doing a .WRITE with extended record gap to magnetic tape. If the crtn argument is left blank, the requested operation completes before control returns to the user program. CRTN=1 is equivalent to executing a .READ or .WRITE in that the function is initiated and returns immediately to the user program. A .WAIT on the channel ensures that the operation is completed. The crtn argument is a completion routine address to be entered when the operation is complete.

The available functions and their function codes are:

Function	MT	CT
Forward to last file		377
Forward to last block		376
Forward to next file		375
Forward to next block		374
Rewind to load point	373	373
Write file gap		372
Write EOF	377	
Forward 1 record	376	
Backspace 1 record	375	
Write	371	
Read	370	
Write with extended file gap	374	
Offline rewind	372	

Function	DX	DM	DY	DL
Read	377	377	377	377
Write	376	376	376	376
Write with deleted data mark	375		375	
Force a read by the handler of the bad block replacement table from block 1 of the disk.		374		374
Return device size		373	373	373

To use the .SPFUN request, the handler must be in memory and a channel associated with a file via a .LOOKUP request.

Other device specific .SPFUN requests are included in Chapter 1 with the appropriate device in the device handler section (section 1.4).

For the RK06/07 handler (DM), special function codes 377 and 376 require the buffer size to be one word larger than necessary for the data. The first word of the buffer contains the error information returned as a result of the .SPFUN request. The data transferred as a result of the read or write request is found in the second and following words of the buffer. The error codes and information are as follows:

Code	Meaning
100000	If the I/O operation is successful
10020	If a bad block is detected (BSE error)
100001	If an ECC error is corrected
100002	If an error recovered on retry
100004	If an error recovered through an offset retry
100010	If an error recovered after recalibration
1774xx	If an error did not recover

Errors:

Code	Explanation
0	Attempt to read or write past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

```
        .TITLE  SPFUN.MAC
;THE FOLLOWING EXAMPLE REWINDS A CASSETTE AND WRITES OUT A 256-WORD BUFFER
;AND THEN A FILE GAP.
        .MCALL  .FETCH,.LOOKUP,.SPFUN,.WRITW
        .MCALL  .EXIT,.PRINT'.WAIT,'.CLOSE

START:
        .FETCH  #HSPC,#CT               ;GET A HANDLER
        BCS     FERR                    ;FETCH ERROR
        .LOOKUP #AREA,#4,#CT            ;LOOK IT UP ON CHANNEL 4
        BCS     LKERR                   ;LOOKUP ERROR
        .SPFUN  #AREA,#4,#373,#0 ;REWIND SYNCHRONOUSLY
        BCS     SPERR                   ;AN ERROR OCCURRED.
        MOV     #3,R5                   ;COUNT
                                        ;BLOCK 0.
        .WRITW  #AREA,#4,#BUFF,#256.,BLK
        BCS     WTERR
        .SPFUN  #AREA,#4,#372,#0,,#1 ;ASYNCHRONOUS FILE GAP
        .PRINT  #DONE
        .WAIT   #4                      ;WAIT FOR DONE
        .CLOSE  #4                      ;CLOSE THE FILE
        .EXIT

AREA:   .BLKW   10
FERR:   .PRINT  #FMSG
        .EXIT
LKERR:  .PRINT  #LKMSG
        .EXIT
SPERR:  .PRINT  #SPMSG
        .EXIT
WTERR:  .PRINT  #WTMSG
        .EXIT
DONE:   .ASCIZ  /ALL DONE/
FMSG:   .ASCIZ  /FETCH?/
LKMSG:  .ASCIZ  /FILE?/
SPMSG:  .ASCIZ  /SPECIAL FUNCTION ERROR/
WTMSG:  .ASCIZ  /WRITE ERROR/
        .EVEN
CT:     .RAD50  /CT /
        .WORD   0,0,0
BUFF:   .BLKW   256.
BLK:    .WORD   0
HSPC=.
        .END    START
```

.SPND/.RSUM

2.4.53 .SPND/.RSUM (FB and XM Only)

The .SPND/.RSUM requests allow a job to control execution of its mainstream code (that code which is not executing as a result of a completion routine). .SPND suspends the mainstream and allows only completion routines (for I/O and mark time requests) to run. .RSUM from one of the completion routines resumes the mainstream code. These functions enable a program to wait for a particular I/O or mark time request by suspending the mainstream and having the selected event's completion routine issue a .RSUM. This differs from the .WAIT request, which suspends the mainstream until all I/O operations on a specific channel have completed.

.SPND

Macro Calls: .SPND

.RSUM

Request Formats:

R0 =	1	0
R0 =	2	0

Notes:

1. The monitor maintains a suspension counter for each job. This counter is decremented by .SPND and incremented by .RSUM. A job is actually suspended only if this counter is negative. Thus, if a .RSUM is issued before a .SPND, the latter request returns immediately.

2. A program must issue an equal number of .SPNDs and .RSUMs.

3. A .RSUM request from the mainstream code increments the suspension counter.

4. A .SPND request from a completion routine decrements the suspension counter, but does not suspend the mainstream. If a completion routine does a .SPND, the mainstream continues until it also issues a .SPND, at which time it is suspended and requires two .RSUMs to proceed.

5. Since a .TWAIT is simulated in the monitor using suspend and resume, a .RSUM issued from a completion routine without a matching .SPND can cause the mainstream to continue past a timed wait before the entire time interval has elapsed.

6. A .SPND or .RSUM, like most other programmed requests, can be issued from within a user-written interrupt service routine if the .INTEN/.SYNCH sequence is followed. All notes referring to .SPND/.RSUM from a completion routine also apply to this case.

Errors:

None.

Example:

```
.TITLE  SPND.MAC
;IN THIS EXAMPLE, THE PROGRAM STARTS A NUMBER OF READ OPERATIONS
;AND SUSPENDS ITSELF UNTIL AT LEAST TWO OF THEM ARE COMPLETE.
        .MCALL  .SPND,.RSUM,.READC,.EXIT,.LOOKUP
        .MCALL  .PRINT,.WAIT

START:
        .LOOKUP #AREA,#2,#FILE2
        BCS     1$
        .LOOKUP #AREA,#3,#FILE3
        BCS     1$
        .LOOKUP #AREA,#4,#FILE4
        BCC     3$
1$:     .PRINT  #2$
        .EXIT
2$:     .ASCIZ  /LOOKUP ERROR/
        .EVEN
3$:

        MOV     #2,RSVCTR       ;WAIT FOR 2 COMPLETIONS
        MOV     #AREA,R5
        .READC  R5,#2,#BUF1,COUNT1,#CROUTN,BLOK1
        BCS     ERROR
        .READC  R5,#3,#BUF2,COUNT2,#CROUTN,BLOK2
        BCS     ERROR
        .READC  R5,#4,#BUF3,COUNT3,#CROUTN,BLOK3
        BCS     ERROR
        .SPND

        .WAIT   #2
        .WAIT   #3
        .WAIT   #4
        .EXIT
CROUTN: ASL     R1              ;DOUBLE CHANNEL # FOR INDEXING

        INC     DONEFL(R1)      ;R1=CHANNEL THAT IS DONE
                                ;SET A FLAG SAYING SO.
        ROR     R0              ;ANY ERRORS?
        ADC     ERRFLG(R1)      ;IF CARRY SET, SET ERROR FLAG FOR CHANNEL
        DEC     RSVCTR          ;ARE WE THE SECOND TO FINISH?
        BNE     1$              ;NO
        .RSUM                   ;YES, START UP
1$:     RTS     PC

ERROR:  .PRINT  #RDMSG
        .EXIT
RDMSG:  .ASCIZ  /READ ERROR/
        .EVEN
AREA:   .BLKW   10
RSVCTR: .WORD   0
COUNT1: .WORD   256.
COUNT2: .WORD   256.
COUNT3: .WORD   256.
BLOK1:  .WORD   0
BLOK2:  .WORD   0
BLOK3:  .WORD   0
FILE2:  .RAD50  /DK TEST2 TMP/
FILE3:  .RAD50  /DK TEST3 TMP/
FILE4:  .RAD50  /DK TEST4 TMP/
DONEFL: .WORD   0,0,0
ERRFLG: .WORD   0,0,0
BUF1:   .BLKW   256.
BUF2:   .BLKW   256.
BUF3:   .BLKW   256.

        .END    START
```

.SRESET

2.4.54 .SRESET

The .SRESET (software reset) request performs the following functions:

1. Dismisses any device handlers that were brought into memory via a .FETCH call. Handlers loaded via the keyboard monitor LOAD command remain resident, as does the system device handler.

2. Purges any currently open files. Files opened for output with .ENTER are never made permanent.

3. Reverts to using only 16 (decimal) I/O channels. Any channels defined with .CDFN are discarded. A .CDFN must be reissued to open more than 16 (decimal) channels after a .SRESET is performed.

4. Resets the I/O queue to one element. A .QSET must be reissued to allocate extra queue elements.

5. Clears the completion queue of any completion routines.

Macro Call .SRESET

Errors:

None.

Example:

```
.TITLE  SRESET.MAC
;IN THIS EXAMPLE, .SRESET IS USED PRIOR TO CALLING THE CSI TO ENSURE
;THAT ALL HANDLERS ARE REMOVED FROM MEMORY AND THE CSI IS STARTED WITH
;A FREE HANDLER AREA. IF THE .SRESET HAD NOT BEEN PERFORMED PRIOR TO THE
;SECOND CALL OF CSIGEN, IT IS POSSIBLE THAT THE SECOND COMMAND STRING
;WOULD LOAD A HANDLER OVER ONE THAT THE MONITOR THOUGHT WAS RESIDENT FROM
;THE FIRST COMMAND LINE.
        .MCALL  .CSIGEN,.SRESET,.EXIT

START:  .CSIGEN #DSPACE,#DEXT,#0 ;GET COMMAND STRING
        MOV     R0,BUFFER       ;R0 POINTS TO FREE MEMORY


DONE:   .SRESET                 ;RELEASE HANDLERS, DELETE
                                ;TENTATIVE FILES
        BR      START           ;AND REPEAT PROGRAM.

DEXT:   .WORD   0,0,0,0         ;NO DEFAULT EXTENSIONS
BUFFER: 0
DSPACE=.                        ;START OF HANDLER AREA.


        .END    START
```

.SYNCH

2.4.55 .SYNCH

This macro call enables the user program to perform monitor programmed requests from within an interrupt service routine. Code following the .SYNCH call executes at processor priority 0 and in the issuing job's context. Unless a .SYNCH is used, issuing programmed requests from interrupt routines is not supported by the system and should not be performed. .SYNCH, like .INTEN, is not an EMT monitor request, but rather a subroutine call to the monitor.

Macro Call: .SYNCH area[,pic]

where: area — is the address of a seven-word block that the user must set aside for use by .SYNCH. Note that the argument, area, represents a special seven-word block used by .SYNCH. This is not the same as the regular area argument used by many other programmed requests. The user must not confuse the two; he should set up a unique seven-word block specifically for the .SYNCH request. The seven-word block appears as:

Word 1	RT-11 maintains this word; its contents should not be altered by the user.
Word 2	The current job's number. This can be obtained by a .GTJB call.
Word 3	Unused.
Word 4	Unused.
Word 5	R0 argument. When a successful return is made from .SYNCH, R0 contains this argument.
Word 6	Must be -1.
Word 7	Must be 0.

pic — is an optional argument that enables the .SYNCH macro to produce position independent code for use by device drivers.

Note:

.SYNCH assumes that the user has not pushed anything on the stack between the .INTEN and .SYNCH calls. This rule must be observed for proper operation.

Errors:

The monitor returns to the location immediately following the .SYNCH if the .SYNCH was rejected. The routine is still unable to issue programmed requests, and R4 and R5 are available for use. Errors returned are due to one of the following causes:

1. Another .SYNCH that specified the same seven-word block is still pending.

2. An illegal job number was specified in the second word of the block. The only currently legal job numbers are 0 and 2.

3. If the job has been aborted or for some reason is no longer running, the .SYNCH fails.

Normal return is to the word after the error return with the routine in user state and thus allowed to issue programmed requests. R0 contains the argument that was in word 5 of the block. R0 and R1 are free to be used without having to be saved. R4 and R5 are not free, and do not contain the same information they contained before the .SYNCH request. A long time can elapse before the program returns from a .SYNCH request since all interrupts must be serviced before the main program can continue. Exit from the routine should be done via an RTS PC.

Example:

```
.TITLE  SYNCH.MAC
        .MCALL  .GTJB,.INTEN,.WRITC,.SYNCH,.EXIT,.PRINT
START:  MOV     #JOB,R5         ;OUTPUT OF .GTJB GOES HERE
        .GTJB   #AREA,R5        ;GET JOB NUMBER
        MOV     (R5),SYNBLK+2   ;STORE THE JOB NUMBER INTO SYNCH BLOCK
                                ;IN HERE WE SET UP INTERRUPT
                                ;PROCESSING, AND START UP THE
                                ;INTERRUPTING DEVICE.

INTRPT: .INTEN  5               ;GO INTO SYSTEM STATE
                                ;RUN AT LEVEL FIVE
                                ;INTERRUPT PROCESSING --
                                ;NOTHING CAN GO ON STACK
        .SYNCH  #SYNBLK         ;TIME TO WRITE A BUFFER
        BR      SYNFAIL         ;SYNCH BLOCK IN USE
;RETURN HERE AT PRIORITY 0.  NOTE: .SYNCH DOES RTI

        .WRITC  #AREA,CHAN,BUFF,WCNT,#CRTN1,BLK
                                ;WRITE A BUFFER
        BCS     WTFAIL          ;FAILED SOMEHOW

                                ;RE-INITIALIZE FOR MORE
        RTS     PC              ;INTERRUPTS AND EXIT
SYNBLK: .WORD   0
        .WORD   0               ;JOB NUMBER
        .WORD   0
        .WORD   0
        .WORD   5               ;R0 CONTAINS 5 ON SCCESSFUL
                                ;SYNCH
        .WORD   -1,0            ;SET UP FOR MONITOR
SYNFAIL:

WTFAIL: MOV     #MSG2,R0
TELLEM: .PRINT
        .EXIT
MSG2:   .ASCIZ  /WRITE FAIL/
        .EVEN
AREA:   .BLKW   5
JOB:    .BLKW   5               ;BUFFER FOR .GTJB
CRTN1:

        RTS     PC
CHAN:   .WORD   0               ;CHANNEL #
BUFF:   .BLKW   256.
WCNT:   .WORD   0               ;WORD COUNT
BLK:    .WORD   0               ;BLOCK #

        .END    START
```

2.4.56 .TLOCK (FB and XM Only)

.TLOCK is used in an FB environment to attempt to gain ownership of the USR. It is similar to .LOCK in that if successful, the user job returns with the USR in memory (it is identical to .LOCK in the SJ monitor). However, if a job attempts to .LOCK the USR while the other job is using it, the requesting job is suspended until the USR is free. With .TLOCK, if the USR is not available, control returns immediately with the C bit set to indicate the .LOCK request failed.

The .TLOCK request allows the job to continue running, with only one sub-job affected. With a .LOCK request, all sub-jobs are automatically suspended, and the other job in the system runs.

Macro Call: .TLOCK

Request Format:

R0 =	7	0

Errors:

Code	Explanation
0	USR is already in use by another job.

Example:

```
.TITLE  TLOCK.MAC
;IN THIS EXAMPLE, THE USER PROGRAM NEEDS THE USR FOR A SUB-JOB IT IS
;EXECUTING. IF IT FAILS TO GET THE USR IT SUSPENDS THAT SUB-JOB AND
;RUNS ANOTHER SUB-JOB. THIS TYPE OF PROCEDURE IS USEFUL TO SCHEDULE SEVERAL
;SUB-JOBS WITHIN A BACKGROUND OR FOREGROUND PROGRAM.
        .MCALL  .TLOCK,.LOOKUP,.UNLOCK,.EXIT,.PRINT

START:

        .TLOCK                          ;GET THE USR
        BCS     SUSPND                  ;FAILED, SUSPEND SUB-JOB
        .LOOKUP #AREA,#4,#J1NAM         ;LOOKUP A FILE
        BCS     LKERR
        .UNLOCK                         ;RELEASE USR

        .EXIT

SUSPND: JSR     PC,SPSJOB               ;SUSPEND SUB-JOB
        JSR     PC,SCHED                ;AND SCHEDULE NEXT USER
        BR      START                   ;LOOP

AREA:   .BLKW   10
J1NAM:  .RAD50  /DK TEST1 TMP/
LKERR:  .PRINT  #LKMSG
        .EXIT
LKMSG:  .ASCIZ  /LOOKUP ERROR/
        .EVEN
SPSJOB: RTS     PC
SCHED:  RTS     PC

        .END    START
```

.TRPSET

2.4.57 .TRPSET

.TRPSET allows the user job to intercept traps to 4 and 10 instead of having the job aborted with a ?MON-F-Trap to 4 or ?MON-F-Trap to 10 message. If .TRPSET is in effect when an error trap occurs, the user-specified routine is entered. The sense of the carry bit on entry to the routine determines which trap occurred: carry bit clear indicates a trap to 4; carry bit set indicates a trap to 10. The user routine should exit with an RTI instruction. Traps to 4 can also be caused by user stack overflow on some processors. These traps are not intercepted by the .TRPSET request but they do cause job abort and a printout of the message ?MON-F-Stack overflow (in the SJ monitor) or ?MON-F-Trap to 4 (in the FB and XM monitors).

Macro Call: .TRPSET area,addr

where: area — is the address of a two-word EMT argument block.

addr — is the address of the user's trap routine. If an address of 0 is specified, the user's trap interception is disabled.

Request Format:

R0 → area:		
	3	0
	addr	

Notes:

It is necessary to reissue a .TRPSET request whenever an error trap occurs and the user routine is entered. The monitor inhibits servicing user traps prior to entering the first user trap routine. Thus, if a trap should occur from within the user's trap routine, an error message is generated and the job is aborted. The last operation the user routine should perform before an RTI is to reissue the .TRPSET request.

In the XM monitor, traps dispatched to a user program by .TRPSET execute in user mode. They appear as interrupts of the user program by a synchronous trap operation. Programs that intercept errors traps by trying to steal the trap vectors must be carefully designed to handle two cases accurately: programs that are virtual jobs and programs that are privileged jobs.

If the program is a virtual job, the stolen vector is in user virtual space that is not mapped to kernel vector space. The proper method is to use .TRPSET; otherwise interception attempts fail and the monitor continues to handle traps to 4 and 10.

If the program is a privileged job, it is mapped to the kernel vector page. The user can steal the error trap vectors from the monitor, but the benefits of doing so must be carefully evaluated in each case. Trap routines run in the mapping mode specified by bits 14 and 15 of the trap vector PS word. With both bits set to 0, kernel mode is set. However, kernel mapping is not always equivalent to user mapping, particularly when extended memory is being used. With both PS word bits set to 1, user mode is set, and the trap routine executes in user mapping.

Errors:

None.

Example:

```
.TITLE  TRPSET.MAC
;IN THIS EXAMPLE, A USER TRAP ROUTINE IS SET AND, WHEN THE TRAP OCCURS,
;THE USER TRAP ROUTINE PRINTS AN APPROPRIATE ERROR MESSAGE.
        .MCALL  .TRPSET,.EXIT,.PRINT

START:

        .TRPSET #AREA,#TRPLOC
        MOV     #101,R0         ;SET TO PRODUCE A TRAP
        TST     (R0)+           ;THIS WILL TRAP TO 4.
        .WORD   67              ;THIS WILL TRAP TO 10.
        .EXIT


TRPLOC: MOV     R0,-(SP)        ;R0 USED BY EMTS
        BCS     1$              ;C SET = TRAP TO 10.
        .PRINT  #TRP4           ;TRAP TO 4
        BR      2$
1$:     .PRINT  #TRP10          ;TRAP TO 10
2$:     .TRPSET #AREA,#TRPLOC   ;RESET TRAP ADDRESS
        MOV     (SP)+,R0        ;RESTORE R0
        RTI


AREA:   .BLKW   10
TRP4:   .ASCIZ  /TRAP TO 4/
TRP10:  .ASCIZ  /TRAP TO 10/
        .EVEN

        .END    START
```

.TTYIN/.TTINR

2.4.58 .TTYIN/TTINR

These requests are used to transfer characters from the console terminal to the user program. The character thus obtained appears right-justified (even byte) in R0. The user can cause the characters to be returned in R0, or R0 and other locations.

The expansion of .TTYIN is:

```
EMT 340
BCS .-2
```

The expansion of .TTINR is:

```
EMT 340
```

If no characters or lines are available when an EMT 340 is executed, return is made with the carry bit set. The implication of these calls is that .TTYIN causes a tight loop waiting for a character/line to appear, while the user can either wait or continue processing using .TTINR.

If the carry bit is set when execution of the .TTINR request is completed, it indicates that no character was available; the user has not yet typed a valid line. Under the FB or XM monitor, .TTINR does not return the carry bit set unless bit 6 of the job status word (JSW) was on when the request was issued (see below).

There are two modes of doing console terminal input. This is governed by bit 12 of the job status word. If bit 12 = 0, normal I/O is performed. In this mode, the following conditions apply:

1. The monitor echoes all characters typed.
2. CTRL/U and the DELETE key perform line deletion and character deletion, respectively.
3. A carriage return, line feed, CTRL/Z, or CTRL/C must be struck before characters on the current line are available to the program. When carriage return is typed, characters on the line typed are passed one-by-one to the user program; both carriage return and line feed are passed to the program.

If bit 12 = 1, the console is in special mode. The effects are:

1. The monitor does not echo characters typed except for CTRL/C and CTRL/O.
2. CTRL/U and the DELETE key do not perform special functions.
3. Characters are immediately available to the program.
4. No ALTMODE conversion is done.

In special mode, the user program must echo the characters received. However, CTRL/C and CTRL/O are acted on by the monitor in the usual way. Bit 12 in the JSW must be set by the user program. This bit is cleared when the program terminates.

CTRL/F and CTRL/B are not affected by the setting of bit 12. The monitor always acts on these characters (unless the SET TT NOFB command is issued).

CTRL/S and CTRL/Q are intercepted by the monitor (unless, under the FB or XM monitor, the SET TT NOPAGE command is issued).

Under the FB or XM monitor, if a terminal input request is made and no character is available, job execution is blocked until a character is ready. This is true for both .TTYIN and .TTINR, and for both normal and special modes. If a program requires execution to continue and the carry bit to be returned, it must set bit 6 of the JSW (location 44) before the .TTINR request. Bit 6 is cleared when a program terminates.

NOTE

The .TTYIN request does not support (track) indirect files. If this function is desired, the .GTLIN request must be used. .TTYIN cannot get a character from an indirect command file.

Macro Calls: .TTYIN char

.TTINR

where: char is a pointer to the location where the character in R0 is to be stored. If the character is specified, the character is in R0 and the address is pointed to by the character. If the character is not specified, the character is in R0.

Errors:

Code	Explanation
0	No characters available in ring buffer.

Example:

Refer to the example following the description of .TTYOUT/.TTOUTR.

.TTYOUT/.TTOUTR

2.4.59 .TTYOUT/.TTOUTR

These requests cause a character to be transmitted from R0 to the console terminal. The difference between the two requests, as in the .TTYIN/.TTINR requests, is that if there is no room for the character in the monitor's buffer, the .TTYOUT request waits for room before proceeding, while the .TTOUTR does not wait for room and the character in R0 is not output.

If the carry bit is set when execution of the .TTOUTR request is completed, it indicates that there is no room in the buffer and that no character was output. Under the FB or XM monitor, .TTOUTR normally does not return the carry bit set. Instead, the job is blocked until room is available in the output buffer. If a job requires execution to continue and the carry bit to be returned, it must turn on bit 6 of the job status word (location 44) before issuing the request.

The .TTINR and .TTOUTR requests have been supplied as a help to those users who do not need to suspend program execution until a console operation is complete. With these modes of I/O, if a no-character or no-room condition occurs, the user program can continue processing and try the operation again at a later time.

NOTE

If a foreground job leaves bit 6 set in the JSW, any further foreground .TTYIN or .TTYOUT requests cause the system to lock out the background. Note also that each job in the foreground/background environment has its own JSW, and therefore can be in different terminal modes independently of the other job.

Macro Calls: .TTYOUT char

.TTOUTR

where: char — is the location containing the character to be loaded in R0 and printed. If not specified, the character in R0 is printed. Upon return from the request, R0 still contains the character.

Errors:

Code	Explanation
0	Output ring buffer full.

Example:

```
.TITLE  TTINR.MAC
;SIMILAR TO TTYIN.MAC, BUT RATHER THAN WAITING FOR THE USER TO TYPE
;SOMETHING AT INLOOP OR WAIT FOR THE OUTPUT BUFFER TO HAVE AVAILABLE
;SPACE AT OUTLOOP, THE ROUTINE HAS BEEN RECODED USING .TTINR AND .TTOUTR.
        .MCALL  .TTYIN,.TTYOUT
        .MCALL  .TTINR,.TTOUTR,.EXIT
JSW     = 44
START:  MOV     #BUFFER,R1          ;POINT R1 TO BUFFER
        CLR     R2                  ;CLEAR CHARACTER COUNT
        BIS     #100,@#JSW          ;WE REALLY WANT CARRY SET
INLOOP: .TTINR                      ;GET CHAR FROM TERMINAL
        BCS     NOCHAR              ;NONE AVAILABLE
CHRIN:  MOVB    R0,(R1)+            ;PUT CHAR IN BUFFER
        INC     R2                  ;INCREASE COUNT
        CMPB    R0,#12              ;WAS LAST CHAR = LF?
        BNE     INLOOP              ;NO-GET NEXT
        MOV     #BUFFER,R1          ;YES-POINT R1 TO BUFFER
OUTLOOP: MOVB   (R1),R0             ;PUT CHAR IN R0
        .TTOUTR                     ;TYPE IT
        BCS     NOROOM              ;NO ROOM IN OUTPUT BUFFER
CHROUT: DEC     R2                  ;DECREASE COUNT
        BEQ     START               ;DONE IF COUNT=0
        INC     R1                  ;BUMP BUFFER POINTER
        BR      OUTLOOP             ;AND TYPE NEXT
NOCHAR:


        .TTINR                      ;PERIODIC CHECK FOR
                                    ;CHARACTER AVAILABILITY
        BCC     CHRIN               ;GOT ONE
                    .
                    .
                    .

        BR      NOCHAR

NOROOM:

        MOVB    (R1),R0             ;PERIODIC ATTEMPT TO TYPE
                                    ;CHARACTER
        .TTOUTR
        BCC     CHROUT              ;SUCCESSFUL
                                    ;CODE TO BE EXECUTED WHILE WAITING


        .REPT   10
        NOP
        .ENDR

        BIC     #100,@#JSW          ;MUST CLEAR THIS BIT
                                    ;SO HANG WHILE
                                    ;WAITING FOR ROOM.
        .TTYOUT (R1)                ;PUT CHAR
        BIS     #100,@#JSW          ;RESTORE NO-WAIT
        BR      CHROUT



BUFFER: .BLKW   100.
        .END    START
```

.TWAIT

2.4.60 .TWAIT (FB and XM Only)

The .TWAIT request suspends the user's job for an indicated length of time. .TWAIT requires a queue element, and thus, should be considered when the .QSET request is executed.

Macro call: .TWAIT area,time

where: area — is the address of a two-word EMT argument block.

time — is a pointer to two words of time (high-order first, low-order second), expressed in ticks.

Request Format:

R0 → area:

24	0
time	

Notes:

Since a .TWAIT is simulated in the monitor using suspend and resume, a .RSUM issued from a completion routine without a matching .SPND can cause the mainstream to continue past a timed wait before the entire time interval has elapsed. In addition, a .TWAIT issued within a completion routine is ignored by the monitor, since it would block the job from ever running again.

The unit of time for this request is clock ticks, which can be 50 Hz or 60 Hz, depending on the local power supply. This must be kept in mind when the time interval is specified.

Errors:

Code	Explanation
0	No queue element was available.

Example:

```
.TITLE  TWAIT.MAC
;.TWAIT CAN BE USED IN APPLICATIONS WHERE A PROGRAM MUST BE ONLY
;ACTIVATED PERIODICALLY. THIS EXAMPLE WILL 'WAKE UP' EVERY TEN SECONDS
;TO PERFORM A TASK, AND THEN SLEEP AGAIN. FOR EXAMPLE PURPOSES ONLY A
;COUNT OF TEN CYCLES IS MAXIMUM.
        .MCALL  .TWAIT,.QSET,.EXIT,.PRINT
GO:     .QSET   #QAREA,#7       ;SET UP 7 EXTRA ELEMENTS
        CLR     COUNT           ;MAX COUNT
START:  MOV     #EMTLST,R0      ;SET R0 TO THE ARG. BLOCK
        .TWAIT                  ;GO TO SLEEP FOR 10 SECONDS
        BCS     NOQ             ;NO QUEUE ELEMENT?
        JSR     PC,TASK         ;DO SOMETHING HERE
        INC     COUNT           ;BUMP COUNTER
        CMP     #10.,COUNT      ;AT MAX ?
        BNE     START           ;NO-GO AGAIN
        .EXIT                   ;EXIT
QAREA:  .BLKW   7*7             ;SPACE FOR 7 ELEMENTS
EMTLST: .BYTE   0,24
        .WORD   TIME
TIME:   .WORD   0,10.*60.       ;10 SECOND INTERVALS
TASK:                           ;SOME GENERALIZED USER
                                ;HERE.

        INC     MPTR
        BIT     #1,MPTR
        BEQ     1$
        .PRINT  #MSG
        RTS     PC
1$:     .PRINT  #MSG1
        RTS     PC
COUNT:  .WORD   0
MPTR:   .WORD   0
MSG:    .ASCIZ  /TICK/
MSG1:   .ASCIZ  /TOCK/
        .EVEN
NOQ:    .EXIT
        .END    GO
```

.WAIT

2.4.61 .WAIT

The .WAIT request suspends program execution until all input/output requests on the specified channel are completed. The .WAIT request combined with the .READ/.WRITE requests makes double-buffering a simple process.

.WAIT also conveys information back through its error returns. An error is returned if either the channel is not currently open or if the last I/O operation resulted in a hardware error.

In an FB system, executing a .WAIT when I/O is pending causes that job to be suspended and the other job to run, if possible.

Macro Call: .WAIT chan

Request Format:

R0 =	0	chan

Errors:

Code	Explanation
0	Channel specified is not open.
1	Hardware error occurred on the previous I/O operation on this channel.

Example:

For an example of .WAIT used for I/O synchronization, see the examples for the .WRITE/.WRITC/.WRITW requests.

```
.TITLE  WAIT.MAC
;AN EXAMPLE OF THE USE OF .WAIT FOR ERROR DETECTION IS ITS USE IN
;CONJUNCTION WITH .CSIGEN TO DETERMINE WHICH FILE FIELDS IN THE COMMAND
;STRING HAVE BEEN SPECIFIED. FOR EXAMPLE, A PROGRAM SUCH
;AS MACRO MIGHT USE THE FOLLOWING CODE TO DETERMINE IF A LISTING
;FILE IS DESIRED.
        .MCALL  .WAIT,.CSIGEN,.EXIT

START:

        .CSIGEN #DSPACE,#DEXT,#0 ;PROCESS COMMAND STRING
        .WAIT   #0              ;CHECK FOR FILE IN FIRST FIELD
        BCS     NOBINARY        ;NO BINARY DESIRED
NOBINARY:

        .WAIT   #1              ;CHECK FOR LISTING SPECIFICATION
        BCS     NOLISTING       ;NO LISTING DESIRED
NOLISTING:

        .WAIT   #3              ;CHECK FOR INPUT FILE OPEN
        BCS     ERROR           ;NO INPUT FILE

ERROR:  .EXIT

DEXT:   .RAD50  /MAC/
        .RAD50  /OBJ/
        .RAD50  /LST/
        .WORD   0
DSPACE=.
        .END    START
```

.WRITE/.WRITC/.WRITW

2.4.62 .WRITE/.WRITC/.WRITW

Note that in the case of .WRITE and .WRITC, additional queue elements should be allocated for buffered I/O operations (see .QSET).

.WRITE

The .WRITE request transfers a specified number of words from memory to the specified channel. Control returns to the user program immediately after the request is queued.

Macro Call: .WRITE area,chan,buf,wcnt,blk

where:	area	is the address of a five-word EMT argument block.
	chan	is a channel number in the range 0-377 (octal).
	buf	is the address of the memory buffer to be used for output.
	wcnt	is the number of words to be written.
	blk	is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:

R0→area:		
	11	chan
	blk	
	buf	
	wcnt	
	1	

Notes:

See the note following .WRITW.

Errors:

Code	Explanation
0	Attempted to write past end-of-file.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to the examples following .WRITW.

.WRITC

The .WRITC request transfers a specified number of words from memory to a specified channel. Control returns to the user program immediately after the request is queued. Execution of the user program continues until the .WRITC is complete, then control passes to the routine specified in the request. When an RTS PC is encountered in the completion routine, control returns to the user program.

Macro Call: .WRITC area,chan,buf,wcnt,crtn,blk

where:	area	is the address of a five-word EMT argument block.

chan	is a channel number in the range 0 to 377 (octal).
buf	is the address of the memory buffer to be used for output.
wcnt	is the number of words to be written.
crtn	is the address of the completion routine to be entered (see Section 2.2.8).
blk	is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:

RC→area:

11	chan
blk	
buf	
wcnt	
crtn	

Notes:

See the note following .WRITW.

When entering a .WRITC completion routine the following are true:

1. R0 contains the contents of the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer. The data can be unreliable.

2. R1 contains the octal channel number of the operation. This is useful when the same completion routine is to be used for several different transfers.

3. Registers R0 and R1 are available for use by the routine, but all other registers must be saved and restored. Data cannot be passed between the main program and completion routines in any register or on the stack.

Errors:

Code	Explanation
0	End-of-file on output. Tried to write outside limits of file.
1	Hardware error occurred.
2	Specified channel is not open.

Example:

Refer to the examples following .WRITW.

.WRITW

The .WRITW request transfers a specified number of words from memory to the specified channel. Control returns to the user program when the .WRITW is complete.

Macro Call: .WRITW area,chan,buf,wcnt,blk

where:

- **area** is the address of a five-word EMT argument block.
- **chan** is a channel number in the range 0-377 (octal).
- **buf** is the address of the buffer to be used for output.
- **wcnt** is the number of words to be written. The number must be positive.
- **blk** is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update blk before it is used again. See Chapter 1 for the significance of the block number for line printers, paper tape readers, etc.

Request Format:

R0→area:		
	11	chan
	blk	
	buf	
	wcnt	
	0	

NOTE

Upon return from any .WRITE, .WRITC or .WRITW programmed request, R0 contains the number of words requested if the write is to a sequential-access device (for example, magtape). If the write is to a random-access device (disk or DECtape), R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. The shortened word count is returned in R0. Note that the write is done and a completion routine, if specified, is entered, unless the request cannot be partially filled (shortened word count = 0).

Errors:

Code	Explanation
0	Attempted to write past EOF.
1	Hardware error.
2	Channel was not opened.

Examples:

Each of the following examples is a simple program to duplicate a paper tape. They illustrate RT-11's three types of .READ/.WRITE requests.

In the first example, .READW and .WRITW are used. The I/O is completely synchronous, with each request retaining control until the buffer is filled (or emptied).

```
.TITLE  READW.MAC
        .MCALL  .FETCH,.READW,.WRITW
        .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT
ERRBYT=52

START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FERR            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
        BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   .READW  R5,R3,#BUFF,#256.,R1 ;READ ONE BLOCK
        BCS     RDERR
        .WRITW  R5,R4,#BUFF,#256.,R1 ;WRITE THAT BLOCK
        BCS     WTERR
        INC     R1              ;BUMP BLOCK. NOTE: THIS IS
                                ;NOT NECESSARY FOR NON-FILE
                                ;DEVICES IN GENERAL. IT IS
                                ;USED HERE AS AN EXAMPLE OF
                                ;A GENERAL TECHNIQUE.
        BR      LOOP            ;KEEP GOING
RDERR:  TSTB    @#ERRBYT                ;ERROR. IS IT EOF?
        BEQ     1$              ;YES
        .PRINT  #RDMSG          ;NO, HARD READ ERROR
        .EXIT
1$:     .CLOSE  R3              ;CLOSE INPUT AND OUTPUT
        .CLOSE  R4
        .EXIT                   ;AND EXIT.
WTERR:  .PRINT  #WTMSG
        .EXIT
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
FERR:   .PRINT  #FMSG           ;ERROR ACTIONS GO HERE. IT IS
        .EXIT                   ;GENERALLY UNDESIRABLE TO
ENERR:  .PRINT  #EMSG           ;EXECUTE A HALT OR RESET
        .EXIT                   ;INSTRUCTION ON ERROR.
LKERR:  .PRINT  #LMSG
        .EXIT
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
RDMSG:  .ASCIZ  /READ ERROR?/
WTMSG:  .ASCIZ  /WRITE ERROR?/
        .EVEN
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

The same routine can be coded using .READ and .WRITE as follows. The .WAIT request is used to determine if the buffer is full or empty prior to its use.

```
.TITLE  READ.MAC
        .MCALL  .FETCH,.READ,.WRITE
        .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT

ERRBYT=52

START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FERR            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
        BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   .READ   R5,R3,#BUFF,#256.,R1 ;READ A BUFFER
        BCS     RDERR
        .WAIT   R3              ;WAIT FOR BUFFER
        BCS     IOERR           ;ERROR HERE IS HARD ERROR
        .WRITE  R5,R4,#BUFF,#256.,R1 ;WRITE TTHE BUFFER
        BCS     IOERR           ;I/O ERROR
        INC     R1
        BR      LOOP            ;KEEP GOING
RDERR:  TSTB    @#ERRBYT                ;ERROR. IS IT EOF?
        BNE     IOERR           ;NO, HARD ERROR
        .CLOSE  R3              ;CLOSE INPUT AND OUTPUT
        .CLOSE  R4
        .EXIT                   ;AND EXIT.
IOERR:  .PRINT  #IOMSG          ;NO, HARD READ ERROR
        .EXIT
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
FERR:   .PRINT  #FMSG           ;ERROR ACTIONS GO HERE. IT IS
        .EXIT                   ;GENERALLY UNDESIRABLE TO
ENERR:  .PRINT  #EMSG           ;EXECUTE A HALT OR RESET
        .EXIT                   ;INSTRUCTION ON ERROR.
LKERR:  .PRINT  #LMSG
        .EXIT
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
IOMSG:  .ASCIZ  "I/O ERROR?"
WTMSG:  .ASCIZ  /WRITE ERROR?/
        .EVEN
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

.READ and .WRITE are also often used for double-buffered I/O. The basic double-buffering algorithm for input is:

	Action		Explanation
	READ	BUFFER 1	Fill BUFFER 1
LOOP:	WAIT	BUFFER 1	Wait for BUFFER 1 to fill
	READ	BUFFER 2	Start filling BUFFER 2
	USE	BUFFER 1	Process BUFFER 1 while BUFFER 2 fills
	WAIT	BUFFER 2	Wait for BUFFER 2 to fill
	READ	BUFFER 1	Start filling BUFFER 1
	USE	BUFFER 2	Process BUFFER 2 while BUFFER 1 fills
	BR	LOOP	

Correspondingly, the basic double-buffering algorithm for output is:

	Action		Explanation
	FILL	BUFFER 1	Prepare BUFFER 1 for output
LOOP:	WRITE	BUFFER 1	Start emptying BUFFER 1
	FILL	BUFFER 2	Fill BUFFER 2 while BUFFER 1 empties
	WAIT	BUFFER 1	Wait for BUFFER 1 to empty
	WRITE	BUFFER 2	Start emptying BUFFER 2
	FILL	BUFFER 1	Fill BUFFER 1 while BUFFER 2 empties
	WAIT	BUFFER 2	Wait for BUFFER 2 to empty
	BR	LOOP	

The following example duplicates a paper tape by using the .READC and .WRITC requests and completion routines. After the first read, the completion routines control the remaining I/O.

```
.TITLE  WRITC2.MAC
        .MCALL  .FETCH,.READC,.WRITC
        .MCALL  .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT

ERRBYT=52

START:  .FETCH  #HSPACE,#TTNAME ;GET TT HANDLER
        BCS     FLNK            ;TT NOT AVAILABLE
        MOV     R0,R2           ;R0 HAS NEXT FREE LOCATION
        .FETCH  R2,#PCNAME      ;GET PC HANDLER
FLNK:   BCS     FERR            ;NOT AVAILABLE
        MOV     #AREA,R5        ;EMT ARGUMENT AREA
        CLR     R4              ;R4 IS OUTPUT CHANNEL; 0
        MOV     #1,R3           ;R3 IS INPUT CHANNEL ;1
        .ENTER  R5,R4,#PCNAME   ;ENTER THE FILE
        BCS     ENERR           ;SOME ERROR IN ENTER
        .LOOKUP R5,R3,#TTNAME   ;LOOKUP FILE ON CHANNEL 1
        BCS     LKERR           ;ERROR IN LOOKUP
        CLR     R1              ;USE R1 AS BLOCK NUMBER
LOOP:   CLR     DFLG            ;CLEAR DONE/ERROR FLAG
        .READC  R5,R3,#BUFF,#256.,#RDCOMP,R1 ;READ ONE BLOCK
        BCS     EOF             ;NO ERROR WILL HAPPEN HERE
1$:     TST     DFLG            ;DONE FLAG SET?
        BEQ     1$              ;NO, WAIT FOR IT TO BE SET.
        BMI     IOERR           ;YES, BUT HARD ERROR OCCURRED
EOF:    .CLOSE  R3              ;CLOSE INPUT AND OUTPUT CHANNELS
        .CLOSE  R4
        .EXIT                   ;ALL DONE
.ENABL  LSB
RDCOMP: ROR     R0              ;IF BIT 0 SET
        BCS     RWERR           ;AN ERROR OCCURRED.
        .WRITC  #AREA,#0,#BUFF,#256.,#WRCOMP,BLKN ;WRITE THAT BLOCK
        BCC     2$              ;ERROR HERE IS HARDWARE
RWERR:  MOV     #-1,DFLG        ;FLAG THE ERROR
2$:     RTS     PC
WRCOMP: ROR     R0
        BCS     RWERR           ;HARDWARE ERROR
        INC     BLKN            ;BUMP BLOCK NUMBER.
        .READC  #AREA,#1,#BUFF,#256.,#RDCOMP,BLKN
        BCC     3$              ;NO ERROR
        TSTB    @#ERRBYT                ;EOF?
        BNE     RWERR           ;NO, HARD  ERROR
        INC     DFLG            ;SAY WE'RE DONE
3$:     RTS     PC
.DSABL  LSB
FERR:   MOV     #FMSG,R0        ;ERROR ACTIONS GO HERE. IT IS
        BR      TYPIT           ;GENERALLY UNDESIRABLE TO
ENERR:  MOV     #EMSG,R0        ;EXECUTE A HALT OR RESET
        BR      TYPIT           ;INSTRUCTION ON ERROR.
IOERR:  MOV     #IOMSG,R0
        BR      TYPIT
LKERR:  MOV     #LMSG,R0
TYPIT:  .PRINT
        .EXIT
.NLIST BEX
FMSG:   .ASCIZ  /NO DEVICE?/
EMSG:   .ASCIZ  /ENTRY ERROR?/
LMSG:   .ASCIZ  /LOOKUP ERROR?/
IOMSG:  .ASCIZ  "I/O ERROR?"
.LIST BEX
.EVEN
DFLG:   .WORD   0
TTNAME: .RAD50  /TT /           ;NOTE THAT TT NEEDS NO FILE NAME
        .WORD   0               ;FILE NAME NEED ONLY BE 0.
PCNAME: .RAD50  /PC /
        .WORD   0
BLKN:   .WORD   0               ;BLOCK NUMBER
AREA:   .BLKW   10
BUFF:   .BLKW   256.
HSPACE=.
        .END    START
```

The following example incorporates the .LOOKUP, .READW, and .CLOSE requests. The program opens the file RT11.MAC on the system device, SY:, for input on channel 0. The first block is read and the file is then closed.

```
.TITLE  WRIT4.MAC
        .MCALL  .CLOSE,.LOOKUP
        .MCALL  .PRINT,.EXIT,.READW,.FETCH

START:  MOV     #LIST,R5                ;EMT ARGUMENT LIST POINTER
        CLR     R4                      ;BLOCK NUMBER
        CLR     R3                      ;CHANNEL #
        .FETCH  #CORADD,#FPTR           ;FETCH DEVICE HANDLER
        BCC     2$
        MOV     #FETMSG,R0              ;FETCH ERROR
1$:     .PRINT                          ;PRINT ERROR MESSAGE
        .EXIT
2$:     .LOOKUP R5,R3,#FPTR             ;LOOKUP FILE ON CHANNEL 0
        BCC     3$
        MOV     #LKMSG,R0               ;PRINT FAILURE MESSAGE
        BR      1$
3$:     .READW  R5,R3,#BUFF,#256.,R4 ;READ ONE BLOCK
        BCC     4$
        MOV     #RDMSG,R0               ;READ ERROR
        BR      1$
4$:     .CLOSE  R3                      ;CLOSE THE CHANNEL
        .EXIT

LIST:   .BLKW   5                       ;LIST FOR EMT CALLS
FPTR:   .RAD50  /SY RT11  MAC/  ;RAD50 OF FIEL NAME,DEVICE
FETMSG: .ASCIZ  /FETCH FAILED/  ;ASCII MESSAGES
LKMSG:  .ASCIZ  /LOOKUP FAILED/
RDMSG:  .ASCIZ  /READ FAILED/
        .EVEN
CORADD: .BLKW   2000                    ;SPACE FOR LARGEST HANDLERS
BUFF=.
        .END    START
```

2.5 CONVERTING VERSION 1 MACRO CALLS TO VERSION 3

As mentioned in the introduction of this chapter, RT-11 Version 3 and later releases support a slightly modified format for system macro calls compared to Version 1. This section details the conversion process from the Version 1 format to Version 3.

2.5.1 Macro Calls Requiring No Conversion

Version 1 macro calls that need no conversion are:

.CSIGN	.RCTLO
.CSISPC	.RELEAS
.DATE	.SETTOP*
.DSTATUS	.SRESET
.EXIT	.TTINR**
.FETCH	.TTOUTR
.HRESET	.TTYIN
.LOCK	.TTYOUT
.PRINT	.UNLOCK
.QSET	

*Provided location 50 is examined for the maximum value.

**Except in FB or XM systems.

2.5.2 Macro Calls That Can Be Converted

The following Version 1 macro calls can be converted:

```
.CLOSE          .RENAME
.DELETE         .REOPEN
.ENTER          .SAVESTATUS
.LOOKUP         .WAIT
.READ           .WRITE
```

The general format of the ..V1.. macro is:

```
.PRGREQ chan,arg1,arg2,...argn
```

In this form, chan is an integer between 0 and 17 (inclusive), and is not a general assembler argument. The channel number is assembled into the EMT instruction itself. The arguments arg1-argn are always pushed either into R0 or on the stack.

The ..V2.. equivalent of the above call is:

```
.PRGREQ area,chan,arg1,....argn
```

In the ..V2.. call, the chan argument can be any legal assembler argument; it need not be in the range 0 to 17 (octal), but should be in the range 0-377 (octal). Area points to a memory list where the arguments arg1...argn will go.

As an example, consider a .READ request in both forms:

```
V1:       .READ 5,#BUFF,#256.,BLOCK

V2:       .READ #AREA,#5,#BUFF,#256.,BLOCK
                .
                .
                .
AREA:     .WORD 0    ;CHANNEL/FUNCTION CODE HERE
          .WORD 0    ;BLOCK NUMBER HERE
          .WORD 0    ;BUFFER ADDRESS HERE
          .WORD 0    ;WORD COUNT HERE
          .WORD 0    ;A 1 GOES HERE.
```

Thus, the difference in the calls is that in Version 2 the channel number becomes a legal assembler argument and the area argument has been added.

Following is a complete list of the conversions necessary for each of the EMT calls. Both the Version 1 and Version 2 formats are given. In Version 3, this function is performed automatically. Note that parameters inside square brackets, [], are optional parameters. Refer to the appropriate section in this chapter for more details on each request.

Version	Programmed Request
V1:	.DELETE chan,dblk
V2:	.DELETE area,chan,dblk[,count]
V1:	.LOOKUP chan,dblk
V2:	.LOOKUP area,chan,dblk[,count]
V1:	.ENTER chan,dblk[,length]
V2:	.ENTER area,chan,dblk[,length[,count]]
V1:	.RENAME chan,dblk
V2:	.RENAME area,chan,dblk
V1:	.SAVESTAT chan,cblk
V2:	.SAVESTAT area,chan,cblk
V1:	.REOPEN chan,cblk
V2:	.REOPEN area,chan,cblk
V1:	.CLOSE chan
V2:	.CLOSE chan
V1:	.READ/.READW chan,buff,wcnt,blk
V2:	.READ/.READW area,chan,buff,wcnt,blk
V1:	.READC chan,buff,wcnt,crtn,blk
V2:	.READC area,chan,buff,wcnt,crtn,blk
V1:	.WRITE/.WRITW chan,buff,wcnt,blk
V2:	.WRITE/.WRITW area,chan,buff,wcnt,blk
V1:	.WRITC chan,buff,wcnt,crtn,blk
V2:	.WRITC area,chan,buff,wcnt,crtn,blk
V1:	.WAIT chan
V2:	.WAIT chan

Important features to keep in mind for Version 3 calls are:

1. Version 3 calls require the area argument, which points to the area where the other arguments will be (unless R0 already points to it and the first word is set up).

2. Enough memory space must be allocated to hold all the required arguments.

3. The chan argument must be a legal assembler argument, not just an integer between 0-17 (octal).

4. Blank fields are permitted in the Version 3 calls. Any field not specified (left blank) is not modified in the argument block.

CHAPTER 3

EXTENDED MEMORY

3.1 INTRODUCTION

The RT-11 operating system is the single-user system for the PDP-11 family of computers. As such, RT-11 has never supported more than 28K words of memory. Extended memory support has been reserved for the multi-tasking systems, since multi-tasking is the usual method for utilizing a large memory space. In such systems, many tasks are run simultaneously, but each task is limited to 32K words or less because of the virtual addressing limitation imposed by the 16-bit word size and the byte addressing capabilities of the PDP-11. However, users of both types of systems encounter the same addressing limitation and have to apply one of several techniques for effectively extending the available logical addressing space.

Two of the standard methods of extending a program are overlaying and chaining. In overlaying, a program is broken into pieces called segments and assembled separately. The segments are then linked together with an overlay handler. When a segment of code is referenced that is not resident, the overlay handler reads the referenced segment into memory, overlaying another segment not currently needed as specified at link time. Communication between segments must be through the root segment of the program, which is never overlayed.

Chaining of programs is most effective when the program can be broken into several completely independent functions that can communicate through a data file. An example of this is the use of a separate program to produce a cross reference listing in RT-11. The MACRO assembler chains to CREF and passes the name of a temporary file containing the necessary symbol data. CREF produces its listing from the file and then chains back to MACRO. These techniques are effective in extending logical addressing space, but they have disadvantages and may not suit a particular application. Overlaying can increase execution time if a great deal of overlaying occurs during program execution. Segmenting may not be applicable. The use of virtual disk arrays can considerably slow down array processing. What is needed is a means of address extension that makes use of the full memory capabilities of the PDP-11.

RT-11 offers as a SYSGEN option the ability to increase the amount of memory it supports from 28K words to 124K words. This optional monitor (extended memory, XM) is a superset of the FB monitor and extends the memory support capability of RT-11 beyond the 28K-word restriction imposed by the 16-bit address size of unmapped PDP-11 processors. The XM monitor is based on the FB monitor and is functionally equivalent to it. The XM monitor offers a set of programmed requests to extend a program's effective logical addressing space that is a subset of similar requests offered on other PDP-11 systems.

The XM monitor software architecture makes it unnecessary for the user to have a detailed knowledge of the PDP-11 memory management hardware. In a mapped system, the user does not need to know where a program resides in physical memory. Mapping, the process of associating program segments with available physical memory, is transparent to the user and is accomplished by the memory management hardware. When a program addresses a location, the memory management unit determines the location's actual physical address in memory. The programmed requests use the memory management hardware to perform address mapping at a higher level that is visible to and controlled by the user. Programs developed on an unmapped system will run on a mapped system. This applies to system programs and user programs. They are called privileged, or compatibility jobs. However, programs that must use the extended memory monitor will not run on an unmapped system. These programs are called virtual jobs. Privileged jobs are not restricted from using the extended memory programmed requests. If they do so, however, they must run on a mapped system under the XM monitor.

The address space extension programmed requests supplied with XM provide the advanced or system programmer with controlled access to extended memory. Through these requests, the program can allocate a region of extended memory for its use and can map selected portions of its virtual address space to portions of that region. A single segment of address space can be mapped into several successive segments of memory, providing an effective extension of the logical address space of the program. The use made of extended memory depends on the application, and can include such uses as resident overlays, buffers, or data arrays.

The remaining sections of this chapter emphasize the use of the programmed requests and their associated parameters, arguments and data structures.

3.2 THE LANGUAGE AND CONCEPTS OF RT-11 EXTENDED MEMORY SUPPORT

Understanding the language and terminology of extended memory is essential to effective program utilization of this feature. Following is a list of terms with their definitions that provides the programmer with the necessary vocabulary:

1. Address Space - The set of addresses available to a program while it is running in a specific processor mode. (RT-11 supports the kernel and user modes of PDP-11 memory management hardware.) The virtual address space is that set of addresses available to a program in a particular mode. The physical address space for the mode is the set of physical addresses to which the virtual addresses are mapped. In general, the kernel and user modes operate in the same virtual address space but possibly in different physical address spaces.

2. Block - A unit of memory. The memory management unit deals in units of 32 words.

3. Dynamic Region - A region in extended memory created by a program at run time through an allocation request.

4. Extended Memory - Memory having a physical address greater than 28K.

5. Kernel Mode - One of the modes of the memory management unit hardware. It is the mapping mode for RMON and the USR. Contrast with user mode.

6. Low Memory - Memory having a physical address in the range 0-28K words.

7. Mapping - The process of associating a virtual address with a physical memory location accomplished by the memory management hardware.

8. Memory Management Fault - An error in an extended memory operation caused by referencing an address not within the program's virtual address space, and indicated by an error message returned by the monitor and displayed at the console terminal.

9. Mode - The memory management unit provides a separate set of relocation registers for use in each of its modes. The mode is specified by bits (15 and 14) in the PS word.

   00 = Kernel
   11 = User

   RT-11 uses kernel mode for monitor and USR operations, and user mode for user programs. The keyboard monitor (KMON) also runs in user mode.

10. Page - A collection of continuous memory addresses mapped by a single relocation register. The 32K word virtual address space is divided into eight 4K word sections, called pages. The lowest address in each page is a whole multiple of 4096. The length of the page is some whole multiple of 32 words ranging from 1 through 128 units. Thus, a page can vary in size from 32 to 4096 words, in 32 word increments.

11. Page Address Register - A memory management unit register containing the base address or relocation constant associated with a page. The memory management unit has 16 page address registers: two groups of eight registers (one register per 4K page). One group is associated with each of the two processor modes (user and kernel).

12. Page Descriptor Register - A memory management unit register containing information associated with a page. This includes the page length, the expansion direction, and the access key. The RT-11 system uses 16 of these registers; eight for user and eight for kernel mode.

13. Physical Address - The hardware address of a specific memory location. The XM monitor supports memory with a physical address between 0 and 124K words.

14. Program Logical Address Space - Program logical address space is the range of effective memory space available to a program. Normally it is limited to the 32K words of virtual address space. It can be extended by overlaying or by using the memory extension capability of the XM monitor.

15. Program Virtual Address Space - Program virtual address space is the 32K (32,768 words) address space accessible to a program determined by the 16-bit word size of the PDP-11 processors.

16. Region - A contiguous segment of physical memory.

17. Static Region - A fixed region of physical memory located in the 0-28K word area. It is created when the program is loaded and it contains the program's base segment. This region cannot be altered by program requests. This region has an identifier of 0.

18. User Mode - One of the modes of the memory management unit hardware. It is the mapping mode for user jobs and KMON. Contrast with kernel mode.

19. Virtual Address - A 16-bit address (0-177777). Under the XM monitor, the memory management unit relocates this address to produce the physical address of the memory location that is to be accessed. (Under the SJ and FB monitors, the virtual address and the physical address of a memory location are the same.)

20. Window - A segment of program virtual address space that begins on a 4K boundary, and can vary in size from 32 to 28K words.

3.3 RT-11 EXTENDED MEMORY FUNCTIONAL DESCRIPTION

The RT-11 software architecture provides programmed requests in the XM monitor that perform the following operations:

1. Divide virtual memory into address windows
2. Allocate regions in extended memory
3. Map the virtual windows to areas within the allocated regions

These three operations are prerequisite to accessing any location in extended memory (above 28K). The first two operations can be performed in any order, but both must be performed before the third operation can take place. A brief description of each operation follows.

3.3.1 Creating Virtual Address Windows

The PDP-11 memory management hardware divides virtual memory into eight pages of 4K (4096) words. The pages are numbered 0 to 7 (see Figure 3-1) corresponding to eight relocation registers. The XM monitor divides virtual memory address space into windows. A window is a segment of address space of any size that must begin on a 4K address boundary. There can be any number of windows up to a maximum of eight (0 to 7). The maximum of eight windows is a compromise between monitor size (seven words per window control block) and allowing enough windows for the user to define eight 4K windows. Windows are similar to overlay segments in that there can be any number of overlay segments, but only one or two are in memory at any given time. Any number of windows can be defined (eight actively defined at a time), but all windows do not have to be mapped at the same time. For example, a multi-user application could segment memory as indicated in Figure 3-2 (example 1). In this figure, the virtual address space is divided linearly. The interpreter remains mapped, but the window containing the user data area is mapped to successive segments of the region. The extended memory region in the example occupies 96K words, which is the largest possible region. If each user is to have a 12K-word data area, as the example shows, there can be up to eight users "sharing" the interpreter at one time. Another example of window usage involves defining several parallel windows of various sizes (see Figure 3-2, example 2) that overlay the same portion of virtual address space.

The size and base of a window is specified by a window definition block supplied by the programmer. Each actively defined window requires a window definition block. The mapping requests must reference the definition block that contains the window specifications, mapping parameters and status information.

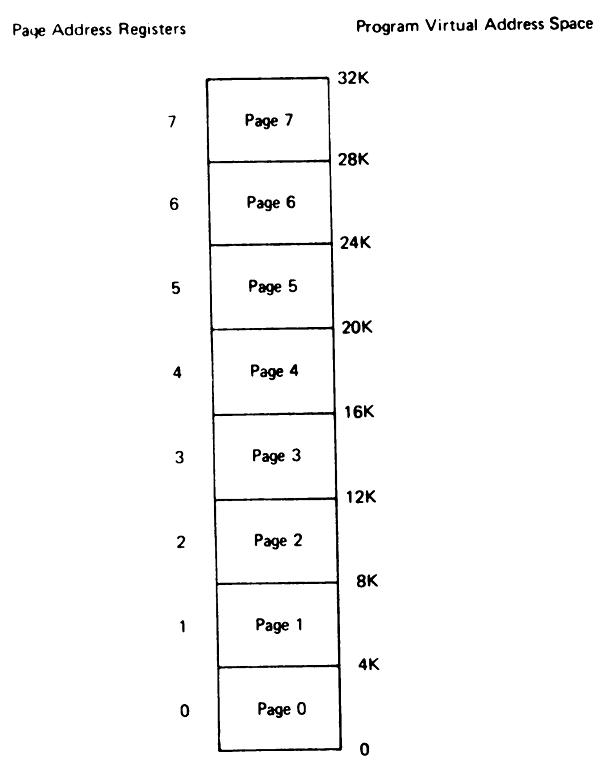


Figure 3-1 Page Address Register Assignments to Program Virtual Address Space Pages

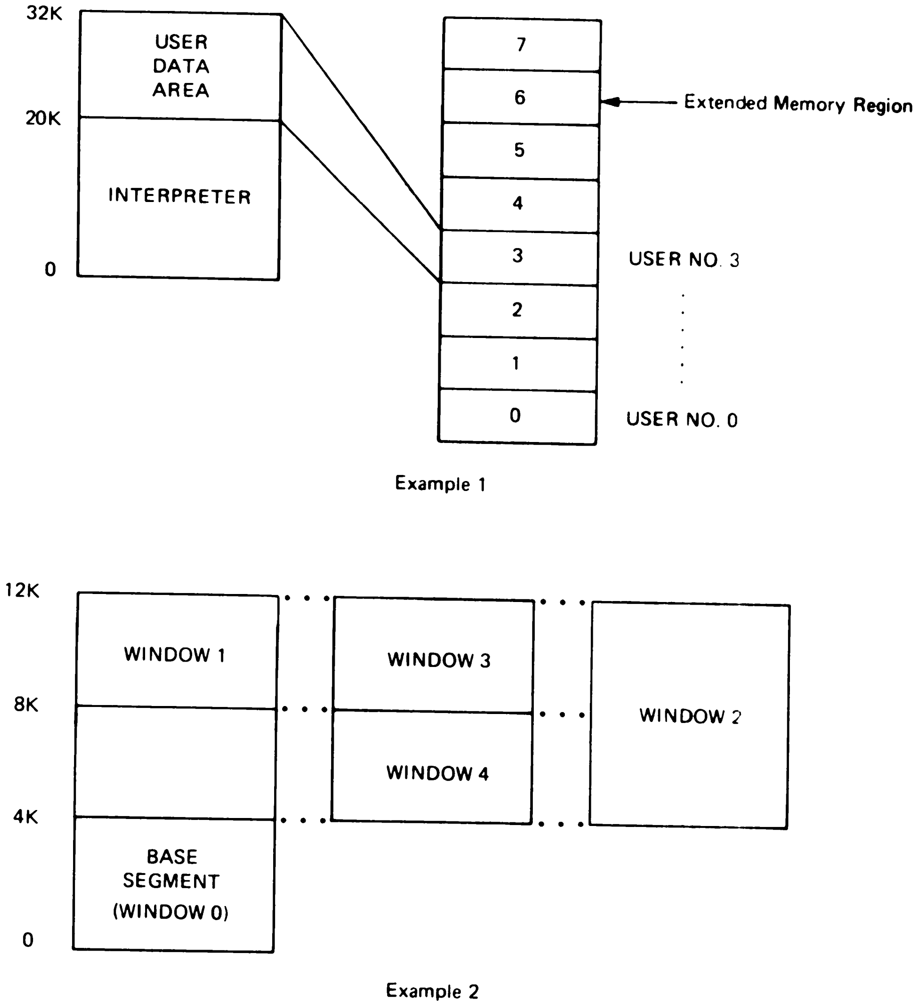


Figure 3-2 Examples of Window Creation

A window's identification is a number from 0 to 7 that is an index to the window's corresponding window block. The address window identified by 0 is a static window and cannot be changed by programmed requests. This window is automatically created and mapped into the static region by the monitor for virtual programs. Every virtual program contains one static window that maps the program's base segment. The base segment is mapped into its corresponding allocated static region of physical memory when the R or FRUN request is executed.

When a program uses extended memory programmed requests, the program views the relationship between virtual and logical address space in terms of windows and regions. Unless a virtual address is part of an existing address window, the address cannot reference a physical memory location. Similarly, a window can be mapped only to an area that is part of an existing region (see Figure 3-3).

However, privileged jobs (discussed in Section 3.3.4.3) usually have all 32K of virtual address space mapped to the lower 28K and the I/O page. The window 0 concept does not apply to privileged jobs.

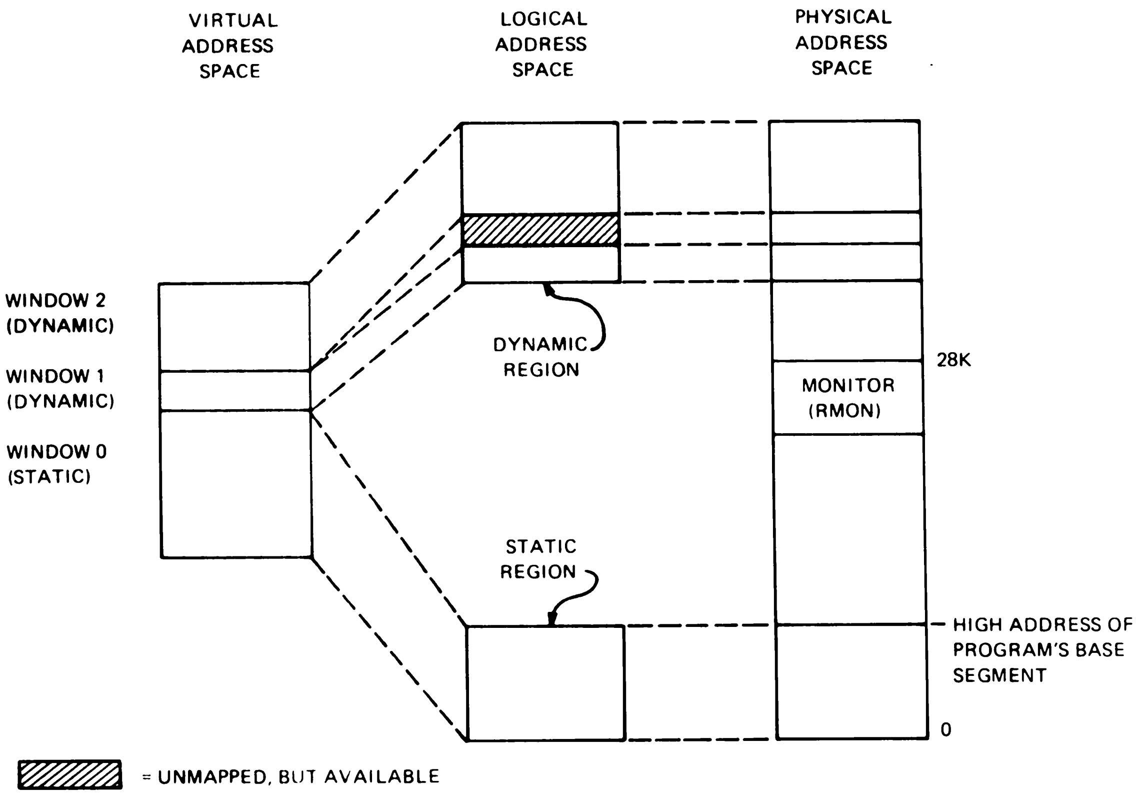


Figure 3-3 Relationship of Windows and Regions

Consider, for example, the case where a program requires two workspace areas (see Figure 3-4): one of 6K words and the other 8K words. The program's base segment requires 8K words. Then, the virtual address space is divided into three windows as follows:

1. Static window, window 0, of 8K words for the base segment

2. Dynamic window of 6K words for workspace area 1

3. Dynamic window of 8K words for workspace area 2

Note that the defined windows overlap page address registers. Window 1 uses page address registers 2 and 3 while window 2 uses registers 4 and 5. Note further that window 1 is only 6K words in size and a discontinuity exists in the program's virtual address space between 14K and 16K. References made to an address in the 14K-16K range cause a memory management fault as long as this discontinuity exists.

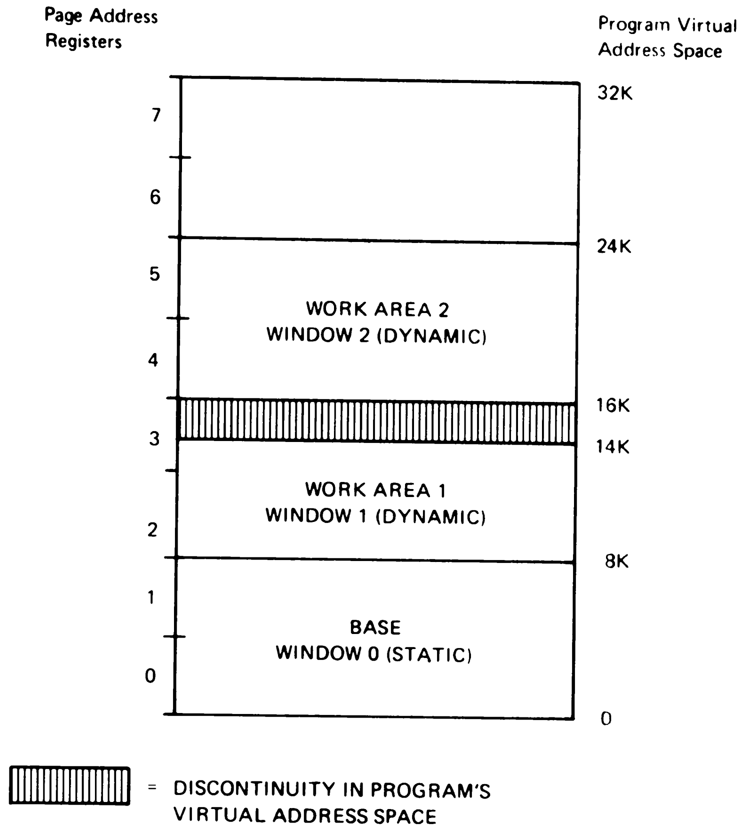


Figure 3-4 Defining Windows for Mapping

This area of undefined virtual address space is produced by the memory management hardware restriction that all windows must begin on a 4K virtual address boundary. In this case, the discontinuity can be avoided by reversing windows 1 and 2. In other situations a linker option can be used to round the window up to a 4K multiple to avoid discontinuities.

Once a program has defined the current windows and regions, it can issue programmed requests to perform operations such as the following:

- Map a window to all or part of a region.
- Unmap a window from one region in order to map it to another region.
- Unmap a window from one part of a region in order to map it to another part of the same region.

3.3.2 Allocating and Deallocating Regions in Extended Memory

Another operation that must be performed before the user can access extended memory is the allocation of dynamic regions. The monitor provides programmed requests that allocate or deallocate dynamic regions. A user program can have up to three of these dynamic regions allocated at any one time. These regions are located in extended memory and do not include the program's base (or static) region located in the lower 28K of memory. The size of a dynamic region can range up to 96K words in 32 word increments. This convention allows the size to be specified in 16 bits and assures that the regions always begin on a 32-word boundary. When a region is created, a unique region identifier is returned by the monitor and is retained in a 3-word region definition block described later in this chapter. Any subsequent programmed request referring to this region must use the region identification code supplied by the monitor. The current window-to-region mapping assignments determine what part of the program's logical address space can be accessed at any given time. Figure 3-5 illustrates created regions that compose a program's logical address space at a discrete time. Since these are dynamic regions and can be allocated and deallocated several times, the logical address space can increase or decrease in size as a function of the controlling program.

Dynamic region deallocation is also accomplished through programmed requests. When a dynamic region is deallocated (static regions cannot be allocated or deallocated), the extended memory area is returned to the monitor's free list where it can be used by other jobs. At the time a region is deallocated, all windows still mapped to the region are automatically unmapped.

3.3.3 Mapping Windows to Regions

Once the regions and address windows have been defined, the initialization work is complete. The final step in accessing extended memory is to connect the windows in virtual memory to the defined regions of physical memory. This process is referred to as "mapping." As stated earlier, the actual mapping operation is a hardware-implemented function performed by the memory management unit. After software has set up the necessary parameters in descriptor blocks, groups of registers in the memory management hardware relocate the user program address references from virtual to physical memory (see Figure 3-6). It must be understood that the user program cannot directly access extended memory without first mapping a portion of virtual addressing space to the desired portion of physical memory.

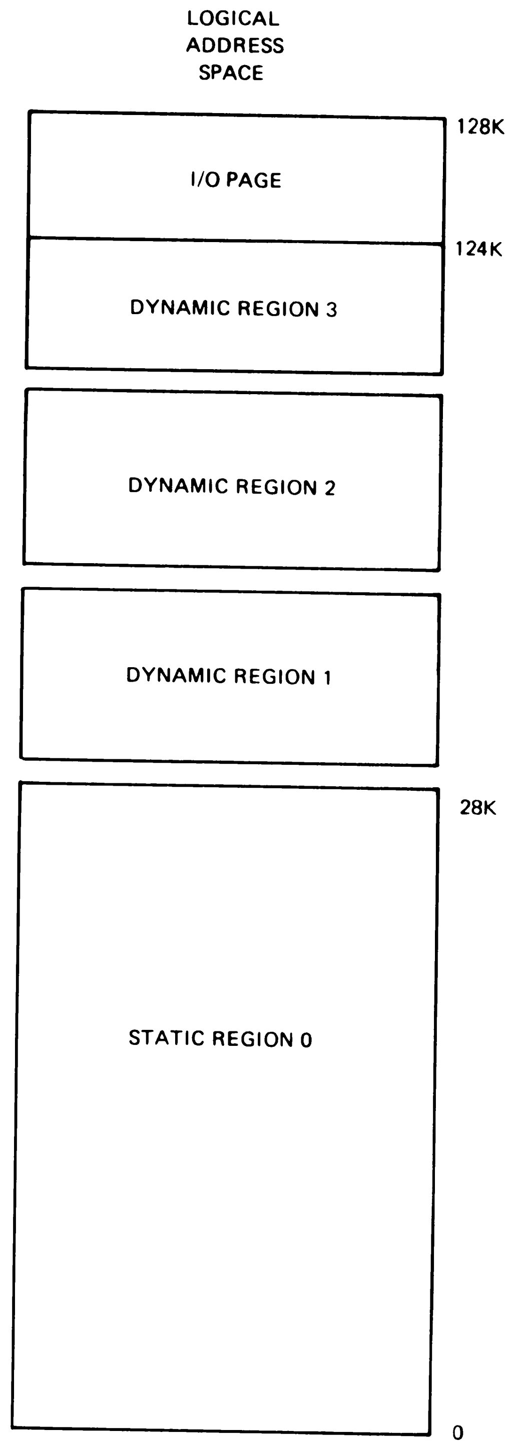


Figure 3-5 Regions Created In Extended Memory

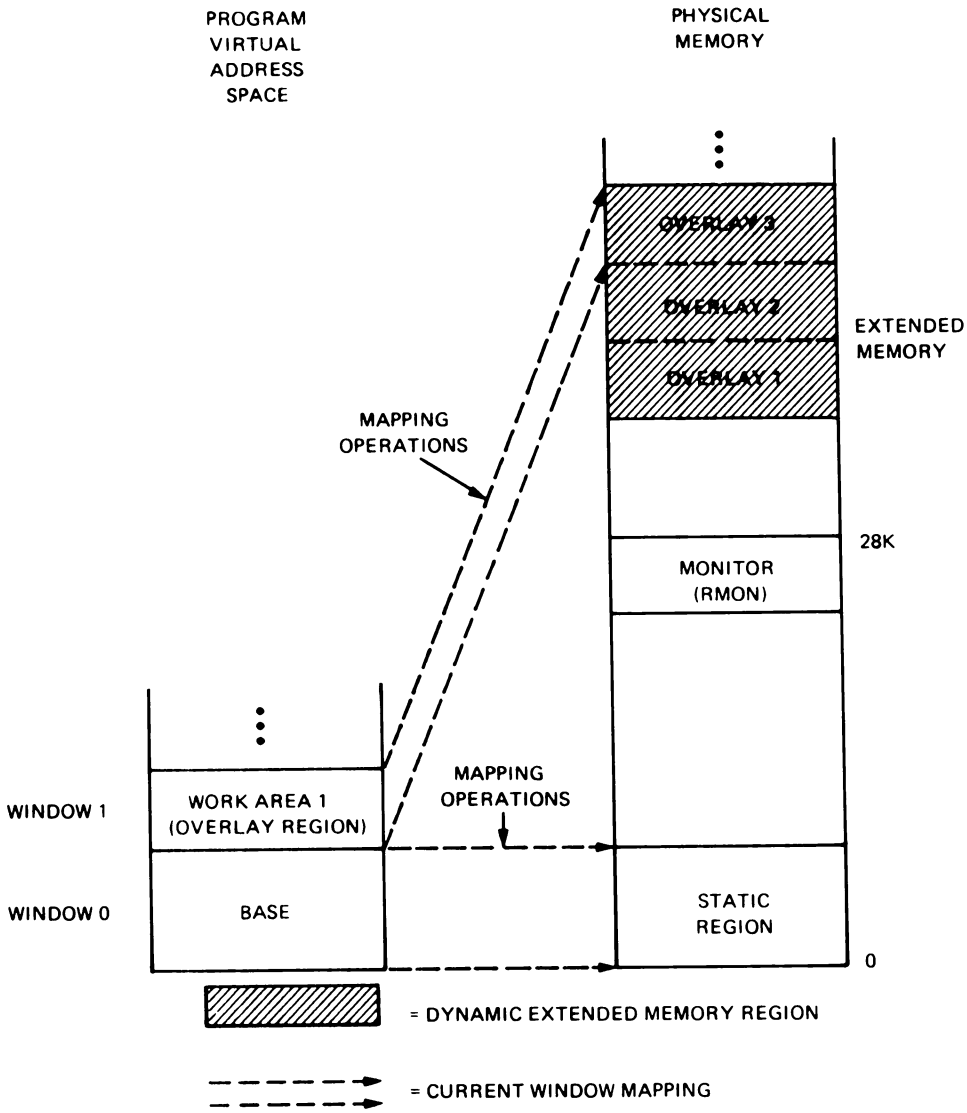


Figure 3-6 Typical Mapping Relationship

The concept of extended memory can be summarized as follows:

1. The user program deals in virtual memory addresses limited to a 16-bit addressing word.

2. A virtual memory address is relocated to an 18-bit physical address capable of accessing 128K words of physical memory.

3. A window of virtual memory can be mapped to successive segments of physical memory by changing offset values through programmed requests.

There are two ways that windows can be mapped to regions. One is to map the window after the creation of that window through a .CRAW (CReate Address Window) programmed request that also performs an implied .MAP programmed request. Under this condition, a window is established and mapped with a single .CRAW request and an additional programmed request is avoided. However, when mapping previously defined windows, the .MAP programmed request must be used. This request can use the same window definition block that was used in the .CRAW mapping operation to map the associated window into a specified region. An offset into the region must be specified. If the window overlaps the end of the region, the system maps as much of the window as fits in the region.

A window can be unmapped by the .UNMAP programmed request. When a window is unmapped in this manner, for a virtual job, that portion of the program's virtual address space becomes undefined. Further attempts to access this unmapped virtual address space result in a memory management fault.

When a window is unmapped by the .UNMAP programmed request for a privileged job (Section 3.3.4.3) the original mapping arrangment is restored.

For both virtual and privileged jobs, an implicit unmapping operation is performed whenever an existing mapped window is remapped to another region or another part of the same region.

3.3.4 Mapping in the Foreground and Background Modes

Extended memory support is available for foreground and background jobs. Both jobs can use extended memory simultaneously but allocated memory regions are dedicated and cannot be shared by jobs. Memory layout for the XM monitor and the types of mapping that can occur are discussed in the following sections.

3.3.4.1 **Monitor Loading and Memory Layout** - The locations of various system components in the XM monitor are very similar to those in the FB monitor. The monitor is bootstrapped into the high end of the lower 28K of memory (see Figure 3-7). The resident monitor (RMON) executes in kernel mode and maps the lower 28K of memory and the I/O page. The kernel vector space is the lower 256 words of physical memory below the background job. The USR runs mapped in kernel mode and is always memory resident. KMON is a privileged background job and runs in user mode, with the same mapping used by the resident monitor.

3.3.4.2 **Virtual Mapping** - This type of mapping provides the full 32K of virtual space for applications that do not need privileged access to the monitor and I/O page. All of the virtual memory space is available to foreground and background jobs. If the virtual mapping configuration is desired, it must be specified at the time the program is loaded into memory. This is done by setting bit 10 of the JSW in location 44 of the system communication area. The user must do this with an .ASECT at assembly time or with a patch prior to run time. The partition where the job is installed is mapped starting at user virtual address 0. The first 500 bytes are the virtual vector and system communication area for the job. Window 0 maps from virtual 0 to the program's high address. Any remaining address space from the program's high address up to 32K is available for mapping into extended memory. Region 0 is defined to include the area from physical location 0 to program partition high limit.

When a virtual foreground job is loaded, it is installed below the resident monitor. The foreground job is mapped to appear as a virtual background job. The program is linked at a default base of 1000 and the region from 0 to 500 is the system communication area and pseudo vector space for the foreground. The job header (impure area) is located just below the foreground job but is not mapped. Unused address space above the foreground job's high limit can be used to define windows so that the job can access extended memory.

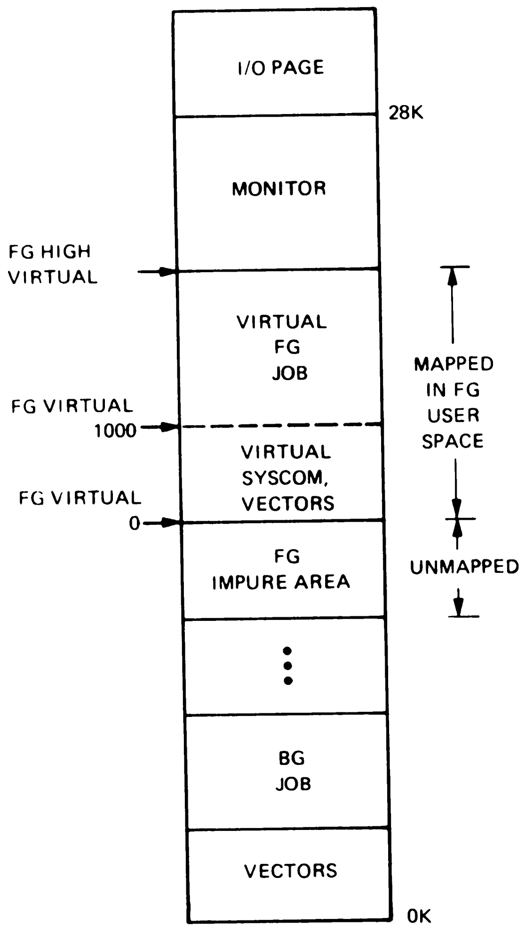


Figure 3-7 Memory Map with Virtual Foreground Job Installed

3.3.4.3 **Privileged or Compatibility Mapping** - This type of mapping is the default mapping that provides compatibility between the FB monitor and the XM monitor (see Figure 3-8). This mapping arrangement gives free access to vectors, monitor and the I/O page. It also is the default mode under which all RT-11 system programs run. In the privileged or compatibility mapping arrangement, the program is normally mapped to the lower 28K of memory plus the I/O page. However, provisions are made through programmed requests for windows to be created and mapped to regions allocated in extended memory so that the effective program space can be increased beyond the virtual address space. In this arrangement, when a window is created and mapped, the default privileged mapping for this space set up by the monitor is temporarily unmapped and the address space is mapped through the window definition block to the new region of memory. Then when the window is unmapped, that address space is returned to the privileged mapping. This type of mapping is particularly important for the user who desires to include interrupt service routines in the system. Interrupt service routines must run in kernel mode. They depend on privileged mapping being identical to kernel mapping.

The privileged job requiring access to the monitor, vectors, and I/O page is limited in the amount of virtual address space it has available to map to extended memory. The user must select portions of the address space that can be borrowed for memory extension operations. If user interrupt service routines are part of the program, the vectors, I/O page, user interrupt service routines, and possibly the monitor must remain mapped at all times that an interrupt can occur.

3.3.4.4 **Context Switching of Virtual and Privileged Jobs** - The two types of jobs (privileged and virtual) are context switched. When the monitor switches between a virtual and a privileged job, it saves context information about the job it switches out, and restores context information about the job it switches in. When a job is switched out, the contents of the memory management mapping registers for the job are not saved. User programs should not manipulate these registers directly because their contents are lost when context switching occurs. The monitor restores job mapping solely from the window and region definition blocks.

When a virtual job is switched in, the monitor disables all user mapping and scans the job's window definition and mapping data. The monitor maps only that 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. Any attempt to access an unmapped address causes a memory management fault. Any unused portions of virtual address space remain unmapped and discontinuities can appear. The virtual job can use the unmapped space by allocating a region in extended memory and mapping to that region.

When a privileged job is switched in, the monitor first sets up the job's user mapping to be identical to kernel mapping (the lower 28K of physical memory and the I/O page). Next, the monitor scans the job's window definition and mapping data. If no windows had been defined at the time the job was switched out, the default kernel mapping remains. If windows had been defined and mapped, those mappings selectively replace the default kernel mapping for the privileged job.

NOTE

> User programs should never attempt to access the memory management unit mapping registers directly. These registers should always be addressed through the appropriate programmed requests.

3.3.5 I/O to Extended Memory

The monitor supports I/O within a job's virtual address space regardless of the physical location of the data buffers. However, the buffers must be in a segment of logical space currently mapped at the time a .READ or .WRITE request is issued. The buffers must also be physically contiguous, which implies that they be completely within an address window. This restriction is necessary because I/O buffer addresses are specified as virtual addresses in the program. The monitor converts the virtual address to an internal physical address representation when the programmed request is executed. This process allows the user program to unmap the buffers on a .READ/.WRITE or .READC/.WRITEC request upon return from the programmed request. Note however, that completion routines must remain mapped until the transfer is complete.

3.4 SUMMARY OF PROGRAMMED REQUESTS

This section briefly describes each of the RT-11 extended memory programmed requests and its associated data structures, arguments and parameters. For convenience, the following requests are ordered by functions and alphabetized within these functional groupings.

Window Requests

.CRAW - Create an address window (3.4.1.3)
.ELAW - Eliminate an address window (3.4.1.4)

Region Requests

.CRRG - Create a region (3.4.2.3)
.ELRG - Eliminate a region (3.4.2.4)

Mapping Requests

.GMCX - Get mapping status (3.4.3.1)
.MAP - Map an address window (3.4.3.2)
.UNMAP - Unmap an address window (3.4.3.3)

The extended memory programmed requests are individually capable of performing a number of separate actions. For example, a single Create an Address Window (.CRAW) request can unmap and eliminate conflicting address windows, create a new window, and then map the new window to a specified region. The complexity of the requests requires a special means of communication between the user program and the RT-11 monitor. The communication is achieved through data structures that:

1. Allow the program to specify which options it wants the monitor to perform.

2. Permit the monitor to provide the job or program with details about the outcome of the requested actions.

Two types of data structures, the region definition block and the window definition block, are used by the requests to provide information to the XM monitor and to receive information from it. Every extended memory programmed request uses one of these structures as its communication area between the job and the monitor. Each issued request includes in the programmed request parameter block a pointer to the appropriate definition block. Values stored by the job in a block define or modify the requested operation. After the monitor has carried out the specified operation, it returns values in various locations within the block to describe the actions taken and to provide the program with information useful for subsequent operations.

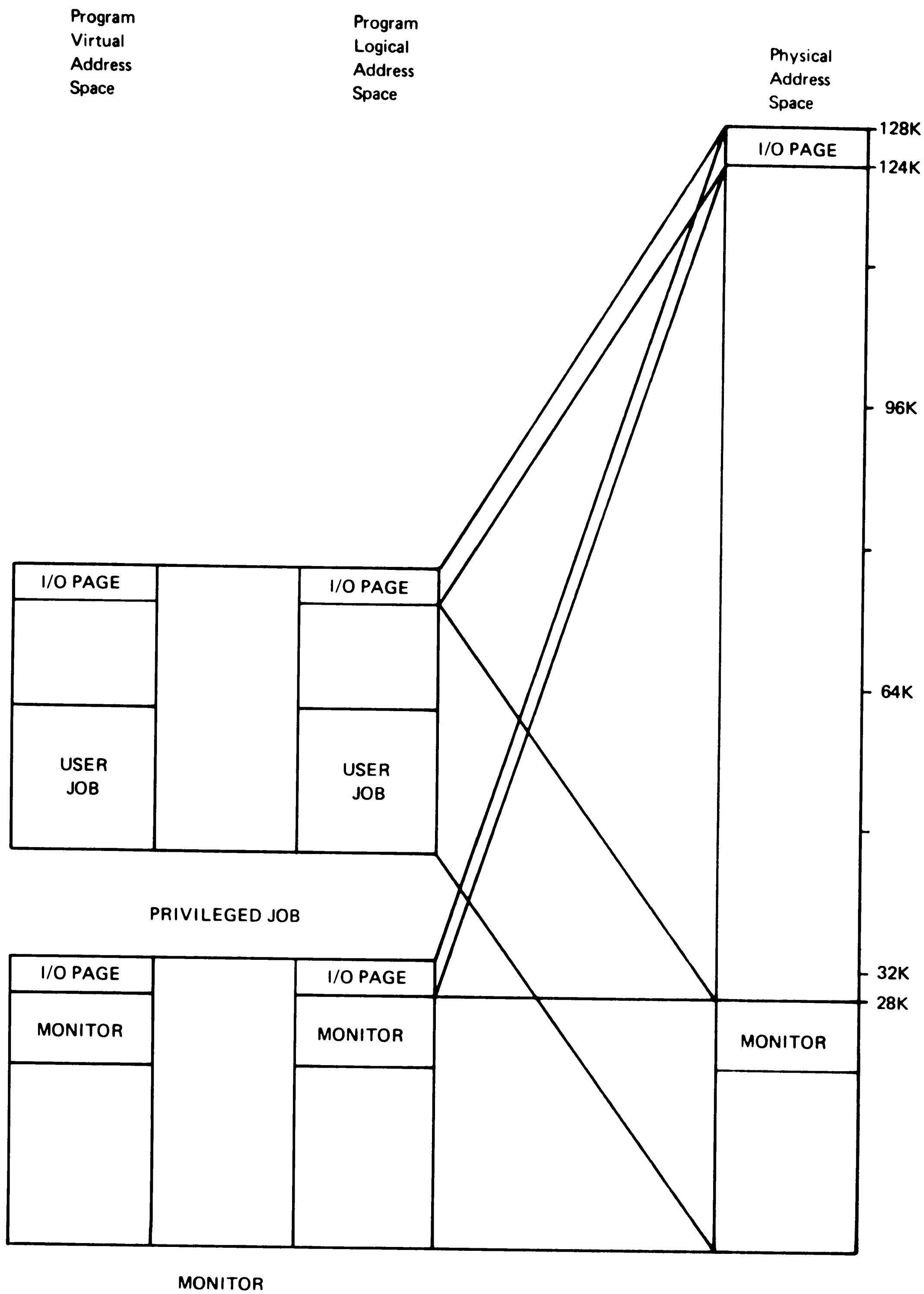


Figure 3-8 RT-11 Privileged Mapping

3.4.1 Programmed Requests to Manipulate Windows

All programmed requests described in this section have a common user data structure, called a window definition block, which is used to store information for the XM monitor and to receive information from it. To use these programmed requests, the window definition block must be defined and set up according to the rules explained in the following section.

3.4.1.1 Window Definition Block

The group of programmed requests to manipulate windows must specify a pointer to the window definition block. The window definition block (see Figure 3-9) is used to define a window and store the returned window identification. It can be created at assembly time by the macro, .WDBBK.

The format of the window definition block is a seven-word block as shown in Figure 3-9.

SYMBOLIC OFFSET	BLOCK FORMAT		BYTE OFFSET
W.NID W.NAPR	BASE PAR	WINDOW ID	0
W.NBAS	VIRTUAL BASE ADDRESS (BYTES)		2
W.NSIZ	WINDOW SIZE (32W BLOCKS)		4
W.NRID	REGION ID		6
W.NOFF	OFFSET IN REGION (32W BLOCKS)		10
W.NLEN	LENGTH TO MAP (32W BLOCKS)		12
W.NSTS	WINDOW STATUS WORD		14

Figure 3-9 Window Definition Block

The first three words are used to establish the window and contain the following information:

W.NID is a one-byte window identifier code returned by the monitor. This identifier must be used in mapping requests involving this window.

W.NAPR is a one-byte value supplied by the user specifying the starting virtual address of the window. Windows must start on a 4K virtual address boundary. The one-byte value is a digit in the range 0 through 7. The digit is the page address register corresponding to the desired 4K virtual address (see Table 3-1). Refer to Figures 3-1 and 3-4, which illustrate the page address registers.

Table 3-1
Virtual Address Boundaries

Starting Virtual Address (Octal)	Page Address Register (W.NAPR)
0 (0K words)	0
20000 (4K words)	1
40000 (8K words)	2
60000 (12K words)	3
100000 (16K words)	4
120000 (20K words)	5
140000 (24K words)	6
160000 (28K words)	7

NOTE

A value of 0 should not be used in W.NAPR since 0 cannot be specified in a .CRAW request.

W.NBAS is the base virtual address of this window, returned by the monitor. This information is redundant with the W.NAPR field, but is provided for user convenience and as a check on the window specification.

W.NSIZ is a one-word value supplied by the user specifying the size of the window in 32-word blocks. If it is not a multiple of 4K words, a discontinuity occurs in the virtual address space, since the next window definition must start on a 4K boundary.

The remaining fields of the window definition block are provided for mapping the window to a region. This same window block can be used with the .MAP request, and since the .CRAW request returns window data in its proper place, an extra operation is avoided. The region-specific data fields returned by the monitor to the window block are as follows:

W.NRID is the region identifier for the region to be mapped, as returned by the .CRRG request.

W.NOFF is the offset into the region at which to start mapping the window, in blocks of 32 words.

W.NLEN is the length of the window to map to the region, in 32-word blocks. If it is 0, the entire window is mapped, or as much as will fit into the region. If W.NLEN is specified, that length portion of the window is mapped. The actual length of the window mapped is returned in W.NLEN.

In addition to creating a window, the .CRAW request is capable of creating a window and then mapping that window to a region by specifying the proper W.NSTS field as described below.

W.NSTS is the window status word.

The window status bits are defined as follows:

Input

WS.MAP Map the window to the specified region after creating it, thus saving an explicit .MAP request.

Output

WS.CRW Address window was successfully created.

WS.UNM One or more windows were unmapped to create and map this window.

WS.ELW One or more windows were eliminated.

3.4.1.2 **Using Macros to Generate Window Definition Blocks** - There are two macros used to generate window definition blocks. The macro .WDBDF defines the offsets and status word bits for the window definition block. The second macro, .WDBBK, actually creates a window definition block. When creating a window definition block with the .WDBBK macro, the offset and status word definitions are automatically supplied because the .WDBBK macro invokes the .WDBDF macro. Hence, the programmer does not need to specify the .WDBDF macro when a .WDBBK is being used. The .WDBBK macro has the following form:

```
.WDBBK wnapr,wnsiz[,wnrid,wnoff,wnlen,wnsts]
```

where:

wnapr is the page address register supplied by the user specifying the starting virtual address of the window (see Table 3-1).

wnsiz specifies the size of the window, in 32-word blocks.

wnrid is the region identifier for the region to be mapped.

wnoff is the offset into the region at which to start mapping the window, in 32-word blocks.

wnlen is the length of the window to map to the region, in 32-word blocks. A value of 0 maps as much of the window as possible.

wnsts is the window status word.

When it is desired only to define the offsets and status bits the .WDBDF macro is invoked by the following call:

```
.WDBDF
```

When this macro is invoked, the following symbols are defined:

1. Window Definition Block Offsets

```
W.NID  = 0
W.NAPR = 1
W.NBAS = 2
W.NSIZ = 4
W.NRID = 6
W.NOFF = 10
W.NLEN = 12
W.NSTS = 14
```

2. Window Definition Block Byte Size

```
W.NLGH = 16
```

3. Window Definition Block Status Word Bits

```
WS.CRW = 100000
WS.UNM =  40000
WS.ELW =  20000
WS.MAP =    400
```

To illustrate the use of these macros to create a window definition block, consider the following example:

A window definition block is to be created defining a window that is 76 decimal blocks long (76 x 32, or 2432 decimal words long) beginning at virtual address 20K, or 120000 octal. Page address register 5 is used.

The defined window is to be mapped to a region beginning 50 decimal blocks (1600 decimal words) from the base of the region. The portion of the region mapped is to be equal to the length of the window or the length remaining in the region, whichever is smaller.

Macro Call: .WDBBK 5,76.,,50.,,WS.MAP

Expands to:

```
        .BYTE  0,5      ;window ID = 0, to be
                        ;returned by monitor.
                        ;window starts at 20K,
                        ;uses address register 5.
        .WORD  0        ;base virtual address of
                        ;window, to be returned
                        ;by monitor.
        .WORD  76.      ;window size in 32-word
                        ;blocks.
        .WORD  0        ;region ID, to be returned
                        ;by .CRRG request
                        ;into the region definition block.
        .WORD  50.      ;offset into region, in 32-word
                        ;blocks, at which to start
                        ;mapping the window.
        .WORD  0        ;length of window to map.
                        ;0 = map as much as possible.
                        ;actual length mapped is
                        ;returned here.
        .WORD  400      ;window status word; 400
                        ;causes .CRAW to also map.
```

Note that setting up the window definition block does not in itself create the window. The .CRRG request must be issued to create the region and return the region ID to the region definition block. If the .CRAW request is to perform an implied .MAP, the region ID must be moved from the region definition block to the window definition block. Then the .CRAW request must be issued to create the window.

.CRAW

3.4.1.3 **Create an Address Window (.CRAW)** - This request defines a virtual address window and optionally maps it into a physical memory region. Mapping occurs if the user has set the WS.MAP bit in the last word of the window definition block. Since the window must start on a 4K boundary, the program only has to specify the page address register to use and the window size in 32-word increments. If the new window being defined overlaps previously defined windows (except window 0, the static window reserved for the virtual program's base segment), the previously defined windows are eliminated before the new window is created.

Macro Call: .CRAW area[,addr]

where:

area is the address of a two-word argument block as indicated below.

36	2
addr	

addr is the address of the window definition block (see Section 3.4.1.1). This argument is optional if the user has filled in the second word of the area argument block with the address pointer.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error codes are contained in error byte 52.

Codes: 0 - Indicates a window alignment error. The window is too large or W.NAPR is greater than 7.

1 - Indicates that no window control blocks are available. The user should eliminate a window first or redefine the division of virtual space into windows so that no more than seven are required.

.ELAW

3.4.1.4 **Eliminate an Address Window (ELAW)** - This request eliminates a defined window. An implied unmapping of the window occurs when its definition block is eliminated.

Macro Call: .ELAW area[,addr]

where:

area   is the address of a two-word argument block as indicated below:

36	3
addr	

addr   is the address of the window definition block for the window to be eliminated.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error code is contained in error byte 52 (refer to Section 3.5 for explanation).

Code: 3 - Indicates an illegal window identifier was specified.

3.4.2 Programmed Requests to Manage Extended Memory Regions

As in the case of the programmed requests to manipulate windows (section 3.4.1), all programmed requests in this section also have a common user data structure, the region definition block. To use these programmed requests, the region definition block must be defined and set up according to the rules and syntax explained in the following section.

3.4.2.1 **Region Definition Block** - The programmed requests to manage extended memory regions must specify a pointer to the region definition block. The region definition block is a three-word block describing the region and having the format shown in Figure 3-10.

SYMBOLIC OFFSET	BLOCK FORMAT	BYTE OFFSET
R.GID	REGION ID	0
R.GSIZ	REGION SIZE (32W BLOCKS)	2
R.GSTS	REGION STATUS WORD	4

Figure 3-10 Region Definition Block

The words contain the following information:

R.GID — is a unique region identifier returned by the monitor. This identifier must be used when referring to the region in other program requests.

R.GSIZ — is the size of the dynamic region, in 32-word blocks, specified by the user.

R.GSTS — is the region status word. The region status bits are defined as follows:

- RS.CRR = 1 — if region was successfully created.
- RS.UNM = 1 — if one or more windows were unmapped as a result of eliminating this region.
- RS.NAL = 1 — if the region specified was not actually allocated at this time.

3.4.2.2 **Using Macros to Generate Region Definition Blocks** - There are two macros used to generate region definition blocks. The first macro, .RDBDF, defines the offsets and status word bits for the region definition blocks. This macro is invoked with the following call:

```
.RDBDF
```

When the macro is invoked, the following symbols are defined:

1. Region Definition Block Offsets

   R.GID = 0

   R.GSIZ = 2

   R.GSTS = 4

2. Region Definition Block Byte Size

   R.GLGH = 6

3. Region Status Word Bits

   RS.CRR = 100000

   RS.UNM = 40000

   RS.NAL = 20000

The second macro, .RDBBK, actually creates the region definition block. The .RDBBK macro has the following form:

```
.RDBBK    rgsiz
```

where:

rgsiz — is the size of the dynamic region, in 32-block words, specified by the user.

When the region definition block is created with the .RDBBK macro, the region definition block offsets and status word are automatically defined. Therefore, the programmer only needs to specify .RDBBK and this macro automatically invokes .RDBDF.

For example, consider the following case. A region of 102 decimal blocks (3264 decimal words) is to be allocated.

The .RDBBK macro sets up the region definition block.

```
                RGADR:         .RDBBK    #102.

Expands to:     RGADR:         .WORD  0      ;region ID=0, to be
                                             ;returned by the
                                             ;monitor.
                               .WORD 102.    ;size of the region
                                             ;in 32-word blocks.
                               .WORD 0       ;region status word.
```

.CRRG

3.4.2.3 **Create a Region (.CRRG)** - The .CRRG request directs the monitor to allocate a dynamic region in physical memory for use by the current requesting program. Symbolically, this request is defined as follows:

Macro Call: .CRRG area [,addr]

where:

area is the address of a two-word argument block as indicated below:

36	0
addr	

addr is the address of the region definition block for the region to be created.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error codes are contained in error byte 52.

Codes:

6 - Indicates no region control blocks available. A region must be eliminated.

7 - Indicates a region of the requested size cannot be created. The size of the largest available region is returned in R0.

10 - Indicates an illegal region size specification. Requests of 0 size and >96K words are illegal.

.ELRG

3.4.2.4 **Eliminate a Region (.ELRG)** - The .ELRG request directs the monitor to eliminate a dynamic region in physical memory and return it to the free list where it can be used by the other jobs.

Macro Call: .ELRG area [,addr]

where:

area — is the address of a two-word argument block as indicated below:

36	1
addr	

addr — is the address of the region definition block for the region to be eliminated. Windows mapped to this region are unmapped. The static region (region 0) cannot be eliminated.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete and the following error code is contained in error byte 52.

Code: 2 - Indicates an illegal region identifier was specified.

3.4.3 Mapping Requests

The mapping requests explained in this section map virtual address windows into dynamic regions in extended memory. To perform this function, the rules and syntax described in the following sections must be followed.

.GMCX

3.4.3.1 **Mapping Status (.GMCX)** - The .GMCX request returns the mapping status of a specified window. Status is returned in the window definition block, and can be used in a subsequent mapping operation. Since the .CRAW request permits combined window creation and mapping operations, it allows entire windows to be changed by modifying certain fields of the window definition block.

The .GMCX request modifies the following fields of the window definition block:

1. W.NAPR - the base page address register of the window

2. W.NBAS - the window base virtual address

3. W.NSIZ - the window size in 32-word blocks

If the window whose status is requested is mapped to a region, the .GMCX request modifies the following additional fields in the window definition block:

1. W.RID - the region identifier

2. W.NOFF - the offset value into the region

3. W.NLEN - the actual length of the mapped window

4. W.NSTS - the state of the WS.MAP bit is set to 1 in the window status word.

Macro Call: .GMCX area[,addr]

where:

area is the address of a two-word argument block as indicated below:

36	6
addr	

addr is the address of the window definition block where the specified window's status is returned.

Errors:

When errors are detected during the execution of this request, the C bit is set after execution is complete, and the following error code is contained in error byte 52.

Code: 3 - Indicates an illegal window identifier was specified.

.MAP

3.4.3.2 **Map a Window (.MAP)** - The .MAP request maps a previously defined address window into a dynamic region of extended memory or into the static region in the lower 28K. If a window is already mapped to a region, an implicit unmapping operation is performed.

Macro Call: .MAP area[,addr]

where:

area is the address of a two-word argument block as indicated below:

36	4
addr	

addr — is the address of the window definition block containing a description of the window to be mapped and the region to be mapped to (see Section 3.4.1.1).

Errors:

When errors are detected during the execution of this request, the C bit is set and the following error codes are contained in error byte 52.

Codes: 2 - Indicates an illegal region identifier was specified.

3 - Indicates an illegal window identifier was specified.

4 - Indicates the specified window was not mapped.

.UNMAP

3.4.3.3 **Unmap a Window (.UNMAP)** - The .UNMAP request unmaps a window and flags that portion of the program's virtual address space as being inaccessible. When an unmap operation is performed for a virtual job, attempts to access the unmapped address space cause a memory management fault. For a privileged job, the default (kernel) mapping is restored when a window is unmapped.

Macro Call: .UNMAP area[,addr]

where:

area — is the address of a two-word argument block as indicated below:

36	5
addr	

addr — is the address of the window control block that describes the window to be unmapped.

Errors:

When errors are detected during the execution of this request, the C bit is set and the following error codes are contained in error byte 52.

Codes: 3 - Indicates an illegal window identifier was specified.

5 - Indicates the specified window was not mapped.

3.5 SUMMARY OF STATUS AND ERROR MONITORING

The XM monitor performs error checking and status monitoring. All extended memory programmed requests generate error codes as indicated in Table 3-2. When errors are detected, the C bit is set on return from the program request, and the error code is returned in error byte

52. In addition to the error codes, two status words are provided to log the status of the requested operations. After completing the requested operation, the monitor sets appropriate bits in the region status word or the window status word (depending on the type of request) to indicate what actions were taken. These status words were discussed in conjunction with the window and region definition blocks. Table 3-3 provides a convenient summary of the byte 52 error codes and status word bits.

3.6 USER INTERRUPT SERVICE ROUTINES WITH THE XM MONITOR

There are three restrictions to using user interrupt service routines with the XM monitor. Such routines can only be used within a privileged job, they must be resident in the lower 28K words of memory, and they must be permanently mapped while they are active.

Care must be used in locating buffers and in setting up vectors for these routines. When an interrupt occurs, the interrupt vector is always taken from kernel space. In XM, kernel space always maps the lower 28K words of memory and the I/O page. The contents of the interrupt vector are placed in the PC and PS, causing the interrupt service routine to execute in the mapping mode specified in the PS of the interrupt vector.

It is possible to execute an interrupt service routine in either mode: kernel or user. However, due to protection mechanisms in the mapping hardware, it is impossible to go from user mode to kernel mode when dismissing an interrupt with an RTI instruction. Consequently, if an interrupt service routine is executed in user mode, it is impossible to return to kernel mode. This guarantees a system crash if the interrupt has interrupted the monitor. Therefore, all interrupt service routines must be serviced in kernel mode (that is, the high byte of the second word of the vector pair must be zero). The interrupt service routine will then execute in kernel mode. This is normally no problem, since privileged job mapping defaults to kernel mode. Thus, old programs that ran under RT-11 version 2C or earlier versions should function properly.

Privileged jobs can also use the memory extension programmed requests. However, the portion of user virtual memory mapped to extended memory at the time of the interrupt is not accessible to the interrupt service routine. This is why the interrupt service routine must use addresses that are permanently mapped in the lower 28K words of memory.

Table 3-2
Extended Memory Error Codes

Error Code	REQUESTS							CAUSE
	.CRAW	.CRRG	.ELAW	.ELRG	.GMCX	.MAP	.UNMAP	
0	x							Window alignment error. Window is too large or W.NAPR is greater than 7, or illegally specified window.
1	x							Attempt to create more than eight windows. Unmap a window first or redefine the division of virtual space into windows.
2				x		x		Illegal region identifier specified.
3			x		x	x	x	Illegal window identifier specified.
4						x		Invalid region offset/window size combination.
5							x	Specified window was not mapped.
6		x						Attempt to create more than four regions. A region must be eliminated.
7		x						A region of the requested size cannot be created. The size of the largest available region is returned in R0.
10		x						Illegal region size specification. A size of 0 or a size greater than 96K words was requested.

Table 3-3
Extended Memory Status Words

Status Word	Symbolic Name	Bit Name	Bit Number	Input/Output*	Bit (octal)	Comments/Definition
Region status word	R.GSTS	RS.CRR	15	Output	100000	Set to 1 for successful region allocation.
		RS.UNM	14	Output	40000	Set to 1 if one or more windows were unmapped as a result of eliminating a region.
		RS.NAL	13	Output	20000	Set to 1 if region specified was not allocated at this time.
Window status word	W.NSTS	WS.CRW	15	Output	100000	Set to 1 if address window was successfully created.
		WS.UNM	14	Output	40000	Set to 1 if one or more windows were unmapped by a .CRAW or a .MAP request.
		WS.ELW	13	Output	20000	Set to 1 if one or more windows were eliminated in a .CRAW request.
		WS.MAP	8	Input	400	Set to 1 if a window is to be mapped in a .CRAW request.

* Input by user or output by monitor.

3.7 EXAMPLE PROGRAM

This section provides a complete and detailed MACRO listing of a sample program that uses all the RT-11 extended memory programmed requests.

```
        .TITLE  XMCOPY
        .NLIST  BEX
        .MCALL  .UNMAP,.ELRG,.ELAW
        .MCALL  .CRRG,.CRAW,.MAP,.PRINT,.EXIT
        .MCALL  .RDBBK,.WDBBK,.TTYOUT,.WDBDF,.RDBDF
        .MCALL  .WRITW,.READW,.CLOSE,.CSIGEN
        JSW     =       44
        J.VIRT  =       2000    ;VIRTUAL BIT IN THE JSW
        ERRBYT  =       52
        APR     =       1
        APR1    =       2
        BUF     =       WDB+W.NBAS  ;GET THE VIRTUAL ADDRESS
        BUF1    =       WDB1+W.NBAS ;GET SECOND BUFFER
                                    ; VIRTUAL ADDRESS
        CORSIZ  =       4096.
        PAGSIZ  =       CORSIZ/256.
        WRNID1  =       WDB1    +       W.NRID
        WRNID   =       WDB     +       W.NRID

        .ASECT
      . = JSW
        .WORD   J.VIRT          ;MAKE THIS A VIRTUAL JOB
        .PSECT

;************************************************
;* XM MONITOR EXAMPLE -- OPENS INPUT FILE AND   *
;* WRITES TO OUTPUT FILE USING 4K BUFFERS IN    *
;* EXTENDED MEMORY.FILES ARE VERIFIED AFTER     *
;* COPYING,TWO 4K BUFFERS IN EXTENDED MEMORY    *
;* ARE USED IN THE VERIFICATION.                *
;************************************************
        .WDBDF          ;CREATE WINDOW DEFINITION BLOCK SYMBOLS
        .RDBDF          ;CREATE REGION DEFINITION BLOCK SYMBOLS

START::
5$:     .CSIGEN #ENDCRE,#DEFLT,#0
        BCS     5$
        .CRRG   #CAREA,#RDB     ;CREATE REGION
        BCC     10$
        JSR     PC,ERROR        ;ERROR REPORT IT
10$:    MOV     RDB,WRNID       ;MOVE REGION ID
        .CRAW   #CAREA,#WDB     ;CREATE WINDOW
        BCC     20$             ;NO ERROR
        JSR     PC,ERROR
20$:    .MAP    #CAREA,#WDB-    ;MAP WINDOW
        BCC     30$             ;NO ERROR
        JSR     PC,ERROR        ;REPORT ERROR
30$:    CLR     R1              ;COUNT REG.
READ:   .READW  #RAREA,#3,BUF,#CORSIZ,R1
        BCC     LOOP            ;NO ERROR
        JSR     PC,ERROR
LOOP:   CMP     R0,#CORSIZ      ;SHORT READ
        BNE     CLOSE           ;CLOSE FILE,SHORT READ
WRITE:  .WRITW  #RAREA,#0,BUF,#CORSIZ,R1
        BCC     ADDIT           ;NO ERROR
        JSR     PC,ERROR
ADDIT:: ADD     #PAGSIZ,R1      ;GO GET NEXT BLOCK
        BR      READ            ;READ LOOP
```

```
CLOSE:: MOV     R0,R2             ;SAVE NUMBER OF WORDS
        .WRITW  #RAREA,#0,BUF,R2,R1 ;WRITE LAST BLOCK
        BCC     CHECK             ;NOW VERIFY DATA
        JSR     PC,ERROR          ;ERROR REPORT IT
CHECK:: .CRRG   #CAREA,#RDB1      ;CREATE A REGION
        BCC     35$               ;NO ERROR HERE
        JSR     PC,ERROR          ;REPORT ERROR
35$:    MOV     RDB1,WRNID1       ;GET REGION ID TO WINDOW

;***********************************************
;* EXAMPLE USING THE .CRAW REQUEST DOING AN    *
;* IMPLIED .MAP REQUEST.                       *
;***********************************************
        .CRAW   #CAREA,#WDB1      ;CREATE WINDOW USING
                                  ; IMPLIED .MAP
        BCC     VERIFY            ;CHECK THE DATA
        JSR     PC,ERROR          ;ERROR REPORT IT
VERIFY::        CLR     R1        ;COUNT REGISTER
GETBLK: .READW  #RAREA,#0,BUF,#CORSIZ,R1
        BCC     40$
        JSR     PC,ERROR
40$:    .READW  #RAREA,#3,BUF1,#CORSIZ,R1
        BCC     50$               ;NO ERROR
        JSR     PC,ERROR          ;ERROR REPORT IT
50$:    CMP     R0,#CORSIZ        ;IS IT A SHORT READ
        BNE     60$               ;NO IT WASNT
        MOV     #CORSIZ,R4        ;REGULAR BUFFER SIZE
        BR      65$               ;GO VERIFY DATA
60$:    MOVB    #-1,SFLAG         ;SET SHORT BUFFER FLAG
        MOV     R0,R4             ;GET SHORT BUFFER
65$:    MOV     BUF,R2            ;GET BUFFER ADDRESS
        MOV     BUF1,R3           ;GET NEXT BUFFER
70$:    CMP     (R2)+,(R3)+       ;VERIFY DATA
        BEQ     75$               ;DATA IS THE SAME
        JSR     PC,ERRDAT
75$:    DEC     R4                ;ARE WE FINISHED
        BNE     70$               ;NO WE ARENT
        ADD     #PAGSIZ,R1        ;GET NEXT PAGE
        TSTB    SFLAG             ;HAS SHORT BUFFER BEEN READ
        BMI     ENDIT             ;YES IT HAS
        BR      GETBLK
ENDIT:: .CLOSE  #0                ;CLOSE CHAN 0
        .CLOSE  #3                ;CLOSE CHANNEL 3
        .PRINT  #ENDPRG
        .EXIT
;***********************************************
;* EXAMPLES SHOWING THE COMPLEMENTS OF THE .MAP *
;* .CRRG, AND THE .CRAW REQUESTS.               *
;***********************************************
        .ELRG   #CAREA,#RDB       ;ELIMIMATE A REGION,
                                  ; IMPLIES ELAW AND UNMAP
        .UNMAP  #CAREA,#WDB1      ;UNMAP WINDOW
        .ELAW   #CAREA,#WDB1      ;ELIMINATE A WINDOW
        .ELRG   #CAREA,#RDB1      ;ELIMINATE A REGION
ERROR:  .PRINT  #ERR
        .EXIT
ERRDAT: .PRINT  #ERRBUF
        .EXIT
RDB:    .RDBBK  CORSIZ/32.        ;DEFINE REGION
WDB:    .WDBBK  APR,CORSIZ/32.
RDB1:   .RDBBK  CORSIZ/32.        ;DEFINE SECOND REGION
WDB1:   .WDBBK  APR1,CORSIZ/32.,0,0,CORSIZ/32.,WS.MAP
CAREA:  .BLKW   2
RAREA:  .BLKW   6
```

```
ENDPRG: .ASCIZ  /END OF EXAMPLE/
ERR:    .ASCIZ  /ERROR IN REQUEST/
ERRBUF: .ASCIZ  /ERROR IN DATA/
        .EVEN
DEFLT:  .RAD50  /MAC/
SFLAG:  .BLKB   0               ;SHORT BUFFER FLAG
        .EVEN
ENDCRE  =       .+2
        .END    START
```

3.8 EXTENDED MEMORY RESTRICTIONS

There are some restrictions that the user of RT-11 extended memory support must be aware of. Some restrictions are physical in nature and imposed by hardware limitations. These restrictions are generally discussed in the descriptions of the applicable programmed requests. Other restrictions are on the use of the system facilities and are discussed below:

1. Device handlers to be used under the XM monitor must be loaded into memory through the keyboard monitor LOAD command before they can be used. User interrupt service routines are not supported for virtual jobs.

2. Some programmed requests are restricted when used with the XM monitor. The requests and their restrictions are as follows:

Programmed Request	Restriction
.CDFN	The channel area specified must be entirely in the lower 28K of physical memory.
.QSET	The queue element space specified must lie entirely in the lower 28K of physical memory, and space must be allowed for 10 words per queue element.
.CNTXSW	This request is not available for virtual jobs. There is no need to context switch the system communication area.
.SETTOP	This request returns the high limit for the job. This address is always within the lower 28K of physical memory. .SETTOP does not reflect any mapping to extended memory that may be in effect.

3.9 SUMMARY AND HIGHLIGHTS OF RT-11 EXTENDED MEMORY SUPPORT

This section gives the highlights and summarizes the basic operations of RT-11 extended memory support. Since this is a new and also complex concept, this section is provided as an aid to understanding the material in this chapter. More than one reading of this chapter is necessary to fully understand its contents.

The following material can be used to review the basic operations and features, and subsequent readings of the chapter can be keyed to amplify this abbreviated discussion.

3.9.1 Extended Memory Prerequisites

The following hardware and software components must be incorporated into the RT-11 operating system to utilize the extended memory feature. The system cannot be bootstrapped without these components.

1. Memory management unit
2. XM monitor and handlers
3. Extended instruction set (EIS)

3.9.2 What Is Extended Memory Support?

Extended memory support is the technique of extending the addressing capability of the RT-11 system beyond its limitation of 32K words imposed by the 16-bit PDP-11 processor word.

3.9.3 How Is Extended Memory Support Implemented?

Extended memory support is implemented through hardware and software.

Hardware

1. Memory management unit

Software

1. XM monitor and handlers
2. User Data Structures
   a. Window Definition Block
   b. Region Definition Block
3. Programmed Requests
   a. .CRAW
   b. .ELAW
   c. .CRRG
   d. .ELRG
   e. .MAP
   f. .UNMAP
   g. .GMCX

3.9.4 How To Use Extended Memory Programmed Requests

This section briefly outlines the various steps involved in using the programmed requests and macros to set up extended memory.

1. Create a region definition block by invoking the macro .RDBBK, or define parameters and set up a region definition block by invoking the macro .RDBDF.

2. Create the necessary regions in extended memory by executing the .CRRG request for each region. A region is eliminated by the .ELRG request.

3. Create a window definition block by invoking the macro .WDBBK, or define parameters and set up a window definition block by invoking the macro .WDBDF.

4. For each window to be created, move the region ID (R.GID, returned by the monitor from .CRRG) from the region definition block into the window definition block. (Move it to W.NRID). This procedure links the window and the region together, but does not map the window to the region.

5. Create the necessary windows in the virtual address space, 0-28K, by executing the .CRAW request for each window to be created. A window is eliminated by the .ELAW request.

6. Map the window to the desired region by executing the .CRAW or .MAP request. A window is unmapped by the .UNMAP request or implicitly unmapped by another .MAP request.

3.9.5 Operational Characteristics of Extended Memory Support

1. The two types of user programs are virtual and privileged.

   a. Virtual provides more address space for mapping to extended memory. It is selected by setting a bit of the JSW before program execution.

   b. Privileged is the default mapping that is compatible with SJ and FB monitors. In this mapping arrangement, the low 28K words of memory and the I/O page are mapped to simulate the non-extended memory environment.

2. The two operating modes are kernel and user.

   a. RMON and the USR run in kernel mode.

   b. KMON and user jobs run in user mode.

CHAPTER 4

SYSTEM SUBROUTINE LIBRARY

4.1 INTRODUCTION

The RT-11 FORTRAN system subroutines are a collection of FORTRAN-callable routines that allow a FORTRAN user to utilize various features of RT-11 foreground/background (FB) and single-job (SJ) monitors. There are no FORTRAN routines to manipulate extended memory under the extended memory (XM) monitor. SYSF4 also provides utility functions, a complete character string manipulation package, and a two-word integer support. This collection of routines is usually placed in a default system library, which is an object module library file called SYSLIB.OBJ. This library file is the default library that the linker uses to resolve undefined globals and is resident on the system device (SY:). The concatenated set of FORTRAN-callable routines is in a file called SYSF4.OBJ. Section 4.1.5 describes how to make these routines into a library.

The user of SYSF4 should be familiar with Chapter 2 of this manual. Chapter 4 assumes that FORTRAN users are familiar with the PDP-11 FORTRAN Language Reference Manual and the RT-11/RSTS/E FORTRAN IV User's Guide.

The following are some of the functions provided by SYSF4:

- Complete RT-11 I/O facilities, including synchronous, asynchronous, and event-driven modes of operation. FORTRAN subroutines can be activated upon completion of an input/output operation.
- Timed scheduling of asynchronous subjobs (completion routines). This feature is standard in FB and XM, and optional in the SJ monitor.
- Complete facilities for interjob communication between foreground and background jobs (FB and XM only).
- FORTRAN interrupt service routines.
- Complete timer support facilities, including timed suspension of execution (FB and XM only), conversion of different time formats, and time of day information. These timer facilities support either 50- or 60-cycle clocks.
- All auxiliary input/output functions provided by RT-11, including the capabilities of opening, closing, renaming, creating, and deleting files from any device.
- All monitor-level informational functions, such as job partition parameters, device statistics, and input/output channel statistics.
- Access to the RT-11 Command String Interpreter (CSI) for accepting and parsing standard RT-11 command strings.
- A character string manipulation package supporting variable-length character strings.
- INTEGER*4 support routines that allow two-word integer computations.

SYSF4 allows the FORTRAN user to write almost all application programs completely in FORTRAN with no assembly language coding. Assembly language programs can also utilize SYSF4 routines (see Section 4.1.3).

4.1.1 Conventions and Restrictions

In general, the SYSF4 routines were written for use with RT-11 V2 or later and FORTRAN IV V1B or later versions. The use of this SYSF4 package with prior versions of RT-11 or FORTRAN leads to unpredictable results.

Programs using IPEEK, IPOKE, IPEEKB, IPOKEB, and/or ISPY to access FORTRAN, monitor, hardware, or other system specific addresses are not guaranteed to run under future releases or on different configurations. Suitable care should be taken with this type of coding to document precisely the use of these access functions and to check a referenced location's usage against the current documentation.

The following must be considered when coding a FORTRAN program that uses SYSF4.

1. Various functions in the SYSF4 package return values that are of type integer, real, and double precision. If the user specifies an IMPLICIT statement that changes the defaults for external function typing, he must explicitly declare the type of those SYSF4 functions that return integer or real results. Double precision functions must always be declared to be type DOUBLE PRECISION (or REAL*8). Failure to observe this requirement leads to unpredictable results.

2. All names of subprograms external to the routine being coded that are being passed to scheduling calls (such as ISCHED, ITIMER, IREADF, etc.) must be specified in an EXTERNAL statement in the FORTRAN program unit issuing the call.

3. Certain arguments (noted as such in the individual routine descriptions) to SYSF4 calls must be located in such a manner as to prohibit the RT-11 USR (User Service Routine) from swapping over them at execution time. If the section OTS$I is not 2K words in length, a program using SYSF4 calls can malfunction because the USR can swap over data to be passed to the USR. This should be rare, but if it occurs, making the USR resident through a SET USR NOSWAP command before starting the job or using the linker's /BOUNDARY option to have OTS$O start at 11000 (octal) eliminates the problem.

   FORTRAN IV version 2 uses .PSECTs to collect code and data into appropriate areas of memory. If RT-11 USR is needed and is not resident, it swaps over a FORTRAN program starting at the symbol OTS$I for 2K words of memory.

4. Quoted-string literals are useful as arguments of calls to routines in the SYSF4 package, notably the character string routines. These literals are allowed in subroutine and function calls.

5. Certain restrictions apply to completion or interrupt routines; see Section 4.2.1 for these restrictions.

4.1.2 Calling SYSF4 Subprograms

SYSF4 subprograms are called in the same manner as user-written subroutines. SYSF4 includes both FUNCTION subprograms and SUBROUTINE subprograms. FUNCTION subprograms receive control by means of a function reference, as:

```
i = function name ([arguments])
```

SUBROUTINE subprograms are invoked by means of a CALL statement; that is,

```
CALL subroutine name [(arguments)]
```

All routines in SYSF4 can be called as FUNCTION subprograms if the return value is desired, or as SUBROUTINE subprograms if no return value is desired. For example, the LOCK subroutine can be referenced as either:

```
CALL LOCK

  or

I = LOCK()
```

Note that routines that do not explicitly return function results produce meaningless values if they are referenced as functions. In the following descriptions, the more common usage (function or subroutine) is shown.

4.1.3 Using SYSF4 with MACRO

The calling sequence is standard for all subroutines, including user-written FORTRAN subprograms and assembly language subprograms. SYSF4 routines can be used with MACRO programs by passing control to the SYSF4 routine with the following instruction:

```
JSR     PC,routine
```

Register five points to an argument list having the following format:

R5 →	undefined	# of arguments
	address of arg. #1	
	address of arg. #2	
	. . .	
	address of arg. #n	

Control is returned to the calling program by use of the instruction:

```
RTS     PC
```

The following is an example of calling a SYSF4 function from an assembly language routine.

```
            .GLOBL JMUL         ;GLOBAL FOR JMUL

            .
            .
            .

            MOV #LIST,R5        ;POINT R5 TO ARG LIST
            JSR PC,JMUL         ;CALL JMUL
            CMP #-2,R0          ;CHECK FOR OVERFLOW
            BEQ OVRFL           ;BRANCH IF ERROR

            .
            .
            .

LIST:       .WORD 3             ;ARG LIST,3 ARGS
            .WORD OPR1          ;ADDR OF 1ST ARG
            .WORD OPR2          ;ADDR OF 2ND ARG
            .WORD RESULT        ;ADDR OF 3RD ARG
OPR1:       .WORD 100           ;LOW-ORDER VALUE OF 1ST ARG
            .WORD 0             ;HIGH-ORDER VALUE OF 1ST ARG
OPR2:       .WORD 10            ;LOW-ORDER VALUE OF 2ND ARG
            .WORD 10            ;HIGH-ORDER VALUE OF 2ND ARG
RESULT:     .BLKW 2             ;2-WORD RESULT (LOW ORDER, HIGH ORDER)
            .END
```

The following routines can be used only with FORTRAN:

GETSTR
IASIGN
ICDFN
IFETCH
IFREEC
IGETC
IGETSP
ILUN
INTSET
IQSET
IRCVDF
IREADF
ISCHED
ISDATF
ISPFNF
ITIMER
IWRITF
PUTSTR
SECNDS

User-written assembly language programs that call SYSF4 subprograms must preserve any pertinent registers before calling the SYSF4 routine and restore the registers, if necessary, upon return.

Function subprograms return a single result in the registers. The register assignments for returning the different variable types are:

Integer, Logical functions - result in R0

Real functions - high-order result in R0,
low-order result in R1

Double Precision functions - result in R0-R3, lowest order result in R3

Complex functions - high-order real result in R0,
low-order real result in R1,
high-order imaginary result in R2,
low-order imaginary result in R3

User-written assembly language routines that interface to the FORTRAN Object Time System (OTS) must be aware of the location of the RT-11 USR (User Service Routine). If a user routine requests a USR function (such as IENTER or LOOKUP), or if the USR is invoked by the FORTRAN OTS, the USR is swapped into memory if it is nonresident. The FORTRAN OTS is designed so that the USR can swap over it. User routines must be written to allow the USR to swap over them or must be located outside the region of memory into which the USR swaps. User interrupt service routines and completion routines, because of their asynchronous nature, must be further restricted to be located where the USR will not swap. The USR, if in a swapping state, will swap at the address specified in location 46 of the system communication area. If location 46 is 0, the USR will swap at the default USR swap location (shown in Figure 1-1). The USR occupies 2K words. Interrupt and completion routines (and their data areas) must not be located in this area. The best way to accomplish this is to examine the link map, determine whether the USR will swap over an assembly language or FORTRAN asynchronous routine, and, if so, change the order of object modules and libraries as specified to the linker. Continue this process until a suitable arrangement is obtained.

The order in which program sections are allocated in the executable program is controlled by the order in which they are first presented to the LINK utility. Applications that are sensitive to this ordering typically separate those sections that contain read-only information (such as executable code and pure data) from impure sections containing variables.

The main program unit of a FORTRAN program (normally the first object program in sequence presented to LINK) declares the following PSECT ordering:

Section Name	Attributes
OTS$I	RW, I, LCL, REL, CON
OTS$P	RW, D, GBL, REL, OVR
SYS$I	RW, I, LCL, REL, CON
USER$I	RW, I, LCL, REL, CON
$CODE	RW, I, LCL, REL, CON
OTS$O	RW, I, LCL, REL, CON
SYS$O	RW, I, LCL, REL, CON
$DATAP	RW, D, LCL, REL, CON
OTS$D	RW, D, LCL, REL, CON
OTS$S	RW, D, LCL, REL, CON
SYS$S	RW, D, LCL, REL, CON
$DATA	RW, D, LCL, REL, CON
USER$D	RW, D, LCL, REL, CON
.$$$$.	RW, D, GBL, REL, OVR
Other COMMON Blocks	RW, D, GBL, REL, OVR

The User Service Routine (USR) can swap over pure code, but must not be loaded over constants or impure data that can be passed as arguments to it.

The above 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.

Assembly-language routines used in applications sensitive to PSECT ordering should use the same program sections as output by the compiler for this purpose. This is, the programmer should place pure code and read-only data in section USER$I, and all impure storage in section USER$D. This ensures that the assembly-language routines will participate in the separation of code and data.

Note that the ordering of PSECTs in an overlay program follows the guidelines herein for each overlay segment (that is, the root segment will contain pure sections followed by impure, and each overlay segment will have a similar separation of pure and impure internal to its structure).

See the <u>RT-11/RSTS/E FORTRAN IV User's Guide</u> for more information.

To remove these restrictions, the user must make the USR resident either by specifying the /NOSWAP option to the FORTRAN command (when compiling a program to be run in the background of FB or XM, or under SJ) or by issuing the SET USR NOSWAP command before executing the program.

4.1.4 Running a FORTRAN Program in the Foreground

The FRUN monitor command must be modified to include various SYSF4 functions. The following formula allocates the needed space when running a FORTRAN program as a foreground job.

```
x = [1/2[44@+(33*N)+(R-136)+A*512)]]
```

The variables are defined as follows:

A = The number of files open at one time. If double buffering is used, A should be multiplied by 2.

N = The number of channels (logical unit numbers).

R = Maximum record length. The default is 136 characters.

This formula must be modified for SYSF4 functions as follows:

The IQSET function requires the formula to include additional space for queue elements (qleng) to be added to the queue:

```
x = [1/2[440+(33*N)+(R-136)+A*512)]]+[10*qleng]
```

The ICDFN function requires the formula to include additional space for the integer number of channels (num) to be allocated.

```
x = [1/2[440+(33*N)+(R-136)+A*512)]]+[6*num]
```

The INTSET function requires the formula to include additional space for the number of INTSET calls (INTSET) issued in the program.

```
x = [1/2[440+(33*N)+(R-136)+A*512]]+[25*INTSET]
```

Any SYSF4 calls, including INTSET, that invoke completion routines must include 64(decimal) words plus the number of words needed to allocate the second record buffer (default is 68(decimal) words). The length of the record buffer is controlled by the /R option to the FORTRAN compiler. If the /R option is not used, the allocation in the formula must be 136(decimal) words.

x = [1/2[440+(33*N)+(R-136)+A*512]]+[64+R/2]

If the /N option does not allocate enough space in the foreground on the initial call to a completion routine, the following message appears:

```
?ERR 0, NON-FORTRAN ERROR CALL
```

This message also appears if there is not enough free memory for the background job or if a completion routine in the single-job monitor is activated during another completion routine. In the latter case, the job aborts. The FB monitor should be used for multiple active completion routines.

4.1.5 Linking with SYSF4

SYSF4 is provided on the distribution media as a file of concatenated object modules (SYSF4.OBJ). If this file is linked directly with the FORTRAN program, all SYSF4 modules are included whether they are used or not. For example:

```
.LINK PROG,SYSF4
```

A library can be created by using the librarian to transform SYSF4 into a library file (SYSLIB.OBJ) as follows:

```
.LIBRARY/CREATE SYSLIB SYSF4
```

Normally the default system library file (SYSLIB.OBJ) also includes the appropriate FORTRAN runtime system routine.

When a library is used, only the modules called are linked with the program. For example:

```
.LINK PROG
```

To add the SYSF4 modules to the default library SYSLIB.OBJ, the following command should be used:

```
.LIBRARY/INSERT SYSLIB SYSF4
```

The following example links the object module EXAMPL.OBJ into a single memory image file EXAMPL.SAV and produces a load map file on LP:. The default system library (SYSLIB.OBJ), which contains the FORTRAN OTS routine, is searched for along with any routines that are not found in other object modules.

```
.LINK/MAP EXAMPL
```

4.2 TYPES OF SYSF4 SERVICES

Ten types of services are available to the user through SYSF4. These are:

1. File-oriented operations
2. Data transfer operations
3. Channel-oriented operations
4. Device and file specifications
5. Timer support operations
6. RT-11 service operations
7. INTEGER*4 support functions
8. Character string functions
9. Radix-50 conversion operations
10. Miscellaneous services

Table 4-1 alphabetically summarizes the SYSF4 subprograms in each of these categories. Those marked with an asterisk (*) are allowed only in a foreground/background environment, under either the FB or XM monitor.

Table 4-1
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
File-Oriented Operations		
CLOSEC	4.3.3	Closes the specified channel.
IDELET	4.3.20	Deletes a file from the specified device.
IENTER	4.3.23	Creates a new file for output.
IRENAM	4.3.41	Changes the name of the indicated file to a new name.
LOOKUP	4.3.70	Opens an existing file for input and/or output via the specified channel.
Data Transfer Functions		
GTLIN	4.3.11	Transfers a line of input from the console terminal or indirect file (if active) to the user program.
*IRCVD *IRCVDC *IRCVDF *IRCVDW	4.3.39	Receives data. Allows a job to read messages or data sent by another job in an FB environment. The four modes correspond to the IREAD, IREADC, IREADF, and IREADW modes.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Data Transfer Functions (cont.)		
IREAD	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of I/O.
IREADC	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers to the assembly language routine specified in the IREADC function call.
IREADF	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers to the FORTRAN subroutine specified in the IREADF function call.
IREADW	4.3.40	Transfers data via the specified channel to a memory buffer and returns control to the program only after the transfer is complete.
*ISDAT *ISDATC *ISDATF *ISDATW	4.3.45	Allows the user to send messages or data to the other job in an FB environment. The four modes correspond to the IWRITE, IWRITC, IWRITF, and IWRITW modes.
ITTINR	4.3.51	Inputs one character from the console keyboard.
ITTOUR	4.3.52	Transfers one character to the console terminal.
IWAIT	4.3.55	Waits for completion of all I/O on a specified channel. (Commonly used with the IREAD and IWRITE functions.)
IWRITC	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers to the assembly language routine specified in the IWRITC function call.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Data Transfer Functions (cont.)		
IWRITE	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of the I/O.
IWRITF	4.3.56	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers to the FORTRAN subroutine specified in the IWRITF function call.
IWRITW	4.3.56	Transfers data via the specified channel to a device and returns control to the user program only after the transfer is complete.
*MTATCH	4.3.72	Attaches a particular terminal in a multi-terminal environment
*MTDTCH	4.3.73	Detaches a particular terminal in a multi-terminal environment
*MTGET	4.3.74	Provides information about a particular terminal in a multi-terminal system.
*MTIN	4.3.75	Transfers characters from a specific terminal to the user program in a multi-terminal system.
*MTOUT	4.3.76	Transfers characters to a specific terminal in a multi-terminal system.
*MTPRNT	4.3.77	Prints a message to a specific terminal in a multi-terminal system.
*MTRCTO	4.3.78	Enables output to terminal by cancelling the effect of a previously typed CTRL/O.
*MTSET	4.3.79	Sets terminal and line characteristics in a multi-terminal system.
*MWAIT	4.3.80	Waits for messages to be processed.
PRINT	4.3.81	Outputs an ASCII string to the console terminal.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Channel-Oriented Operations		
ICDFN	4.3.15	Defines additional I/O channels.
*ICHCPY	4.3.16	Allows access to files currently open in the other job's environment.
*ICSTAT	4.3.19	Returns the status of a specified channel.
IFREEC	4.3.25	Returns the specified RT-11 channel to the available pool of channels for the FORTRAN I/O system.
IGETC	4.3.26	Allocates an RT-11 channel and marks it in use to the FORTRAN I/O system.
ILUN	4.3.29	Returns the RT-11 channel number with which a FORTRAN logical unit is associated.
IREOPN	4.3.42	Restores the parameters stored via an ISAVES function and reopens the channel for I/O.
ISAVES	4.3.43	Stores five words of channel status information into a user-specified array.
PURGE	4.3.82	Deactivates a channel.
Device and File Specifications		
IASIGN	4.3.14	Sets information in the FORTRAN logical unit table.
ICSI	4.3.18	Calls the RT-11 CSI in special mode to decode file specifications and options.
Timer Support Operations		
CVTTIM	4.3.5	Converts a two-word internal format time to hours, minutes, seconds, and ticks.
GTIM	4.3.9	Gets time of day.
ICMKT	4.3.17	Cancels an unexpired ISCHED, ITIMER, or MRKT request. (Valid for SJ monitors with timer support, a SYSGEN option.)

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Timer Support Operations (cont.)		
ISCHED	4.3.44	Schedules the specified FORTRAN subroutine to be entered at the specified time of day as an asynchronous completion routine. (Valid for SJ monitors with timer support, a SYSGEN option.)
ISLEEP	4.3.46	Suspends main program execution of the running job for a specified amount of time; completion routines continue to run. (Valid for SJ monitors with timer support, a SYSGEN option.)
ITIMER	4.3.49	Schedules the specified FORTRAN subroutine to be entered as an asynchronous completion routine when the time interval specified has elapsed. (Valid for SJ monitors with timer support, a SYSGEN option.)
*ITWAIT	4.3.53	Suspends the running job for a specified amount of time; completion routines continue to run.
*IUNTIL	4.3.54	Suspends the main program execution of the running job until a specified time of day; completion routines continue to run.
JTIME	4.3.67	Converts hours, minutes, seconds, and ticks into 2-word internal format time.
MRKT	4.3.71	Marks time; that is, schedules an assembly language routine to be activated as an asynchronous completion routine after a specified interval. (Valid for SJ monitors with timer support, a SYSGEN option.)
SECNDS	4.3.93	Returns the current system time in seconds past midnight minus the value of a specified argument.
TIMASC	4.3.98	Converts a specified two-word internal format time into an eight-character ASCII string.
TIME	4.3.99	Returns the current system time of day as an 8-character ASCII string.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
RT-11 Services		
CHAIN	4.3.2	Chains to another program (in the background job only).
*DEVICE	4.3.6	Specifies actions to be taken on normal or abnormal program termination, such as turning off interrupt enable on foreign devices, etc.
GTJB	4.3.10	Returns the parameters of this job.
IDSTAT	4.3.22	Returns the status of the specified device.
IFETCH	4.3.29	Loads a device handler into memory.
IQSET	4.3.37	Expands the size of the RT-11 monitor queue from the free space managed by the FORTRAN system.
ISPFN ISPFNC ISPFNF ISPFNW	4.3.47	Issues special function requests to various handlers, such as magtape. The four modes correspond to the IWRITE, IWRITC, IWRITF, the IWRITW modes.
*ITLOCK	4.3.50	Indicates whether the USR is currently in use by another job and performs a LOCK if the USR is available.
LOCK	4.3.69	Makes the RT-11 monitor User Service Routine (USR) permanently resident until an UNLOCK function is executed. A portion of the user's program is swapped out to make room for the USR if necessary.
RCHAIN	4.3.86	Allows a program to access variables passed across a chain.
RCTRLO	4.3.87	Enables output to the terminal by cancelling the effect of a previously typed CTRL/O, if any.
*RESUME	4.3.89	Causes the main program execution of a job to resume where it was suspended by a SUSPND function call.
SCCA	4.3.90	Intercepts a CTRL/C command initiated at the console terminal.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
RT-11 Services (cont.)		
SETCMD	4.3.99	Passes command lines to the keyboard monitor to be executed after the program exits.
*SUSPND	4.3.97	Suspends main program execution of the running job; completion routines continue to execute.
UNLOCK	4.3.102	Releases the USR if a LOCK was performed; the user program is swapped in if required.
INTEGER*4 Support Functions		
AJFLT	4.3.1	Converts a specified INTEGER*4 value to REAL*4 and returns the result as the function value.
DJFLT	4.3.7	Converts a specified INTEGER*4 value to REAL*8 and returns the result as the function value.
IAJFLT	4.3.13	Converts a specified INTEGER*4 value to REAL*4 and stores the result.
IDJFLT	4.3.21	Converts a specified INTEGER*4 value to REAL*8 and stores the result.
IJCVT	4.3.28	Converts a specified INTEGER*4 value to INTEGER*2.
JADD	4.3.57	Computes the sum of two INTEGER*4 values.
JAFIX	4.3.58	Converts a REAL*4 value to INTEGER*4.
JCMP	4.3.59	Compares two INTEGER*4 values and returns an INTEGER*2 value that reflects the signed comparison result.
JDFIX	4.3.60	Converts a REAL*8 value to INTEGER*4.
JDIV	4.3.61	Computes the quotient and remainder of two INTEGER*4 values.
JICVT	4.3.62	Converts an INTEGER*2 value to INTEGER*4.
JJCVT	4.3.63	Converts the two-word internal time format to INTEGER*4 format, and vice versa.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (Cont.)
Summary of SYSF4 Sbuprograms

Function Call	Section	Purpose
INTEGER*4 Support Functions (cont.)		
JMOV	4.3.64	Assigns an INTEGER*4 value to a variable.
JMUL	4.3.65	Computes the product of two INTEGER*4 values.
JSUB	4.3.66	Computes the difference between two INTEGER*4 values.
Character String Functions		
CONCAT	4.3.4	Concatenates two variable-length strings.
GETSTR	4.3.8	Reads a character string from a specified FORTRAN logical unit.
INDEX	4.3.30	Returns the location in one string of the first occurrence of another string .
INSERT	4.3.31	Replaces a portion of one string with another string.
ISCOMP	4.3.91	Compares two character strings.
IVERIF	4.3.103	Indicates whether characters in one string appear in another.
LEN	4.3.68	Returns the number of characters in a specified string.
PUTSTR	4.3.83	Writes a variable-length character string on a specified FORTRAN logical unit.
REPEAT	4.3.88	Concatenates a specified string with itself to provide an indicated number of copies and stores the resultant string.
SCOMP	4.3.91	Compares two character strings.
SCOPY	4.3.92	Copies a character string from one array to another.
STRPAD	4.3.95	Pads a variable-length string on the right with blanks to create a new string of a specified length.
SUBSTR	4.3.96	Copies a substring from a specified string.
TRANSL	4.3.100	Replaces one string with another after performing character modification.

* FB and XM monitors only.

(continued on next page)

Table 4-1 (cont.)
Summary of SYSF4 Subprograms

Function Call	Section	Purpose
Character String Functions (cont.)		
TRIM	4.3.101	Removes trailing blanks from a character string.
VERIFY	4.3.103	Indicates whether characters in one string appear in another.
Radix-50 Conversion Operations		
IRAD50	4.3.38	Converts ASCII characters to Radix-50, returning the number of characters converted.
R50ASC	4.3.84	Converts Radix-50 characters to ASCII.
RAD50	4.3.85	Converts six ASCII characters, returning a REAL*4 result that is the 2-word Radix-50 value.
Miscellaneous Services		
IADDR	4.3.12	Obtains the memory address of a specified entity.
IGETSP	4.3.27	Returns the address and size (in words) of free space obtained from the FORTRAN system.
INTSET	4.3.32	Establishes a specified FORTRAN subroutine as an interrupt service routine at a specified priority.
IPEEK	4.3.33	Returns the value of a word located at a specified absolute memory address.
IPEEKB	4.3.34	Returns the value of a byte located at a specified byte address.
IPOKE	4.3.35	Stores an integer value in an absolute memory location.
IPOKEB	4.3.36	Stores an integer value in a specified byte location.
ISPY	4.3.48	Returns the integer value of the word located at a specified offset from the beginning of the RT-11 resident monitor.

* FB and XM monitors only.

Routines requiring the USR (see Section 2.3) differ between the SJ and FB monitors. (The USR is always resident in the XM monitor.) The following functions require the use of the USR:

CLOSEC
GETSTR (only if first I/O operation on logical unit)
ICDFN (single job only)
GETLIN
ICSI
IDELET
IDSTAT
IENTER
IFETCH
IQSET
IRENAM
ITLOCK (only if USR is not in use by the other job)
LOCK (only if USR is in a swapping state)
LOOKUP
PUTSTR (only if first I/O operation on logical unit)

Certain requests require a queue element taken from the same list as the I/O queue elements. These are:

IRCVD/IRCVDC/IRCVDF/IRCVDW
IREAD/IREADC/IREADF/IREADW
ISCHED
ISDAT/ISDATC/ISDATF/ISDATW
ISLEEP
ISPFN/ISPFNC/ISPFNF/ISPFNW
ITIMER
ITWAIT
IUNTIL
IWRITC/IWRITE/IWRITF/IWRITW
MRKT
MWAIT

4.2.1 Completion Routines

Completion routines are subprograms that execute asynchronously with a main program. A completion routine is scheduled to run as soon as possible after the event for which it has been waiting has completed (such as the completion of an I/O transfer, or the lapsing of a specified time interval). All completion routines of the current job have higher priority than other parts of the job; therefore, once a completion routine becomes runnable because of its associated event, it interrupts execution of the job and continues to execute until it relinquishes control. See Figure 1-2, in Chapter 1.

Completion routines are handled differently in the SJ and the FB monitors. In SJ, completion routines are totally asynchronous and can interrupt one another. In FB (and XM), completion routines do not interrupt each other but are queued and made to wait until the correct job is running. (For further information on completion routines, see Sections 2.2.8 and 4.1.4).

A FORTRAN completion routine can have a maximum of two arguments:

SUBROUTINE crtn [(iarg1,iarg2)]

where: iarg1 is equivalent to R0 on entry to an assembly language completion routine.

iarg2 is equivalent to R1 on entry to an assembly language completion routine.

If an error occurs in a completion routine or in a subroutine at completion level, the error handler traces back normally through to the original interruption of the main program. Thus the traceback is shown as though the completion routine were called from the main program and lets the user know where the main program was executing if a fatal error occurs.

Certain restrictions apply to completion routines (those routines that are activated by the following calls:)

INTSET
IRCVDC
IRCVDF
IREADC
IREADF
ISCHED
ISDATC
ISDATF
ISPFNC
ISPFNF
ITIMER
IWRITC
IWRITF
MRKT

These restrictions are:

1. The first subroutine call that references a FORTRAN completion routine must be issued from the main program.

2. No channels can be allocated (by calls to IGETC) or freed (by calls to IFREEC) from a completion routine. Channels to be used by completion routines should be allocated and placed in a COMMON block for use by the routine.

3. The completion routine cannot perform any call that requires the use of the USR, such as LOOKUP and IENTER. See Section 4.2 for a list of SYSF4 functions that call the USR.

4. Files to be operated upon in completion routines must be opened and closed by the main program. There are, however, no restrictions on the input or output operations that can be performed in the completion routine. If many files must be made available to the completion routine, they can be opened by the main program and saved for later use (without tying up RT-11 channels) by the ISAVES call. The completion routine can later make them available by reattaching the file to a channel with an IREOPN call.

5. FORTRAN subprograms are reusable but not reentrant. A given subprogram can be used many times as a completion routine or as a routine in the main program, but a subprogram executing as main program code does not work properly if it is interrupted at the completion level. This restriction applies to all subprograms that can be invoked at the completion level and can be active at the same time in the main program.

6. Only one completion function should be active at any time under the single-job monitor (see Section 4.1.4).

7. Assembly language completion routines must be exited via an RTS PC.

8. FORTRAN completion routines must be exited by execution of a RETURN or END statement in the subroutine.

4.2.2 Channel-Oriented Operations

An RT-11 channel being used for input/output with SYSF4 must be allocated in one of the following two ways:

1. The channel is allocated and marked in use to the FORTRAN I/O system by a call to IGETC and is later freed by a call to IFREEC.

2. An ICDFN call is issued to define more channels (up to 256). All channels numbered greater than 17 (octal) can be freely used by the programmer; the FORTRAN I/O system uses only channels 0 through 17 (octal).

Channels must be allocated in the main program routine or its subprograms, not in routines that are activated as the result of I/O completion events or ISCHED or ITIMER calls.

4.2.3 INTEGER*4 Support Functions

INTEGER*4 variables are allocated two words of storage. INTEGER*4 values are stored in two's complement representation. The first word (lower address) contains the low-order part of the value, and the second word (higher address) contains the sign and the high-order part of the value. The range of numbers supported is -2^31+1 to 2^31-1.

Note that this format differs from the 2-word internal time format that stores the high-order part of the value in the first word and the low-order part in the second. The JJCVT function (Section 4.3.63) is provided for conversion between the two internal formats.

Integer and real arguments to subprograms are indicated in the following manner in this chapter.

i = INTEGER*2 arguments
j = INTEGER*4 arguments
a = REAL*4 arguments
d = REAL*8 arguments

When the DATA statement is used to initialize INTEGER*4 variables, it must specify both the low- and high-order parts. The following example only initializes the first word.

```
INTEGER*4 J
DATA J/3/
```

The correct way to initialize an INTEGER*4 variable to a constant (such as, 3) is shown below:

```
INTEGER*4 J
INTEGER*2 I(2)
EQUIVALENCE (J,I)
DATA I/3,0/        !INITIALIZE J TO 3
```

If initializing an INTEGER*4 variable to a negative value (such as -4), the high-order (second word) part must be the continuation of the two's complement of the low-order part. For example:

```
INTEGER*4 J
INTEGER*2 I(2)
EQUIVALENCE (J,I)
DATA I/-4,-1/      !INITIALIZE J TO -4
```

The following form is suitable for INTEGER*4 arguments to subprograms:

```
INTEGER*2 J(2)
DATA J/3,0/        !LOW-ORDER,HIGH-ORDER
```

4.2.4 Character String Functions

The SYSF4 character string functions and routines provide variable-length string support for RT-11 FORTRAN. SYSF4 calls are provided to perform the following character string operations:

- Read character strings from a specified FORTRAN logical unit (GETSTR).
- Write character strings to a specified FORTRAN logical unit (PUTSTR).
- Concatenate variable-length strings (CONCAT).
- Return the position of one string in another (INDEX).
- Insert one string into another (INSERT).
- Return the length of a string (LEN).
- Repeat a character string (REPEAT).
- Compare two strings (SCOMP).
- Copy a character string (SCOPY).
- Pad a string with rightmost blanks (STRPAD).
- Copy a substring from a string (SUBSTR).
- Perform character modification (TRANSL).
- Remove trailing blanks (TRIM).
- Verify the presence of characters in a string (VERIFY).

Strings are stored in LOGICAL*1 arrays that are defined and dimensioned by the FORTRAN programmer. Strings are stored in these arrays as one character per array element plus a zero element to indicate the current end of the string (ASCIZ format).

The length of a string can vary at execution time, ranging from zero characters in length to one less than the size of the array that stores the string. The maximum size of any string is 32767 characters. Strings can contain any of the 7-bit ASCII characters except null (0), since the null character is used to mark the end of the string. Bit 7 of each character must be cleared (0); therefore, the valid characters are those whose decimal representations range from 1 to 127, inclusive.

The ASCII code used in this string package is the same as that employed by FORTRAN for A-type FORMAT items, ENCODE/DECODE strings, and object-time FORMAT strings. ASCIZ strings in the form used by these routines are generated by the FORTRAN compiler whenever quoted strings are used as arguments in the CALL statement. Note that a null string (a string containing no characters) can be represented in FORTRAN by a variable or constant of any type that contains the value zero, or by a LOGICAL variable or constant with the .FALSE. value.

The SYSF4 user should ensure that a string never overflows the array that contains it by being aware of the length of the string result produced by each routine. In many routines where the resultant string length can vary or is difficult to determine, an optional integer argument can be specified to the subroutine to limit the length. In the sections describing the character string routines, this argument is called len. The length of an output string is limited to the value specified for len plus one (for the null terminator); therefore the array receiving the result must be at least len plus one elements in size.

The optional argument err can be included when len is specified. Err is a logical variable that should be initialized by the FORTRAN program to the .FALSE. value. If a string function is given the arguments len and err, and len is actually used to limit the length of the string result, then err is set to the .TRUE. value. If len is not used to truncate the string, err is unchanged; that is, it remains .FALSE..

Arguments len and err are normally optional arguments. The argument len can appear alone; however, len must appear if err is specified. The err argument should be used for GETSTR and PUTSTR.

Several routines use the concept of character position. Each character in a string is assigned a position number that is one greater than the position of the character immediately to its left. The first character in a string is in position one.

4.2.4.1 Allocating Character String Variables

- A one-dimensional LOGICAL*1 array can be used to contain a single string whose length can vary from zero characters to one fewer than the dimensioned length of the array. For example:

```
LOGICAL*1 A(45)      !ALLOCATE SPACE FOR STRING VARIABLE A
```

The preceding example allows array A to be used as a string variable that can contain a string of 44 or fewer characters. Similarly, a two-dimensional LOGICAL*1 array can be used to contain a one-dimensional array of strings. Each string in the array can have a length up to one less than the first dimension of the LOGICAL*1 array. There can be as many strings as the number specified for the second dimension of the LOGICAL*1 array. For example:

```
LOGICAL*1 W(21,10)      !ALLOCATE AN ARRAY OF STRINGS
```

The preceding example creates a string array W that has ten string elements, each of which can contain up to 20 characters. String I in array W is referenced in subroutine or function calls as W(1,I).

A two-dimensional string array can be allocated. For example:

```
LOGICAL*1 T(14,5,7)      !ALLOCATE A 5 BY 7 STRING ARRAY
```

In the preceding example, each string in array T can vary in length to a maximum of 13 characters. String I,J of the array can be referenced as T(1,I,J). Note that T is the same as T(1,1,1). This dimensioning process can be continued to create string arrays of up to six dimensions (represented by LOGICAL*1 arrays of up to seven dimensions).

4.2.4.2 **Passing Strings to Subprograms** - The LOGICAL*1 arrays that contain strings can be placed in a COMMON block and referenced by any or all routines with a similar COMMON declaration. However, care should be taken when a LOGICAL*1 array is placed in a COMMON block, for if such an array has an odd length, it causes all succeeding variables in the COMMON block to be assigned odd addresses.

A LOGICAL*1 array has an odd length only if the product of its dimensions is odd. For example:

```
LOGICAL*1 B(10,7)      !(10*7)=70; EVEN LENGTH
LOGICAL*1 H(21)        !21 IS ODD; ODD LENGTH
```

If odd length arrays are to be placed in a COMMON block, they should either be placed at the end of the block or they should be paired to result in an effective even length. For example:

```
COMMON A1,A2,A3(10),H(21)          !PLACE ODD-SIZED ARRAY AT END

     or

COMMON A1,A2,H(21),H1(7),A3(10)   !PAIR ODD-SIZED ARRAYS H AND H1
```

Note that these cautions apply only to LOGICAL*1 variables and arrays.

The second method of passing strings to subprograms is through arguments and formal parameters. A single string can be passed by using its array name as an argument. For example:

```
LOGICAL*1 A(21)      !STRING VARIABLE "A", 20 CHARACTERS MAXIMUM
CALL SUBR(A)         !PASS STRING A TO SUBROUTINE SUBR
```

If the maximum length of a string argument is unknown in a subroutine or function, or if the routine is used to handle many different length strings, the dummy argument in the routine should be declared as a LOGICAL*1 array with a dimension of one, such as LOGICAL*1 ARG(1). In this case, the string routines correctly determine the length of ARG whenever it is used, but it is not possible to determine the maximum size string that can be stored in ARG. If a multi-dimensional array of strings is passed to a routine, it must be declared in the called program with the same dimensions as were specified in the calling program.

NOTE

> The length argument specified in many of the character string functions refers to the maximum length of the string excluding the necessary null byte terminator. The length of the LOGICAL*1 array to receive the string must be at least one greater than the length argument.

4.2.4.3 **Using Quoted-String Literals** - Quoted-strings can be used as arguments to any of the string routines that are invoked by functions or the CALL statement. They can be used for routines invoked as functions. The following example compares the string in the array NAME to the constant string DOE, JOHN and sets the value of the integer variable M accordingly.

```
CALL SCOMP(NAME,'DOE, JOHN',M)
```

4.3 LIBRARY FUNCTIONS AND SUBROUTINES

This section presents all SYSF4 functions and subroutines in alphabetical order. To reference these subprograms by usage, see Table 4-1.

AJFLT

4.3.1 AJFLT

The AJFLT function converts an INTEGER*4 value to a REAL*4 value and returns that result as the function value.

Form: a = AJFLT (jsrc)

where: jsrc is the INTEGER*4 variable to be converted.

Function Results:

The function result is the REAL*4 value that is the result of the operation.

Example:

The following example converts the INTEGER*4 value contained in JVAL to single precision (REAL*4), multiplies it by 3.5, and stores the result in VALUE.

```
REAL*4 VALUE,AJFLT
INTEGER*4 JVAL
.
.
.
VALUE=AJFLT(JVAL)*3.5
```

CHAIN

4.3.2 CHAIN

The CHAIN subroutine allows a background program (or any program in the single-job system) to transfer control directly to another background program, passing it specified information. CHAIN cannot be called from a completion or interrupt routine. CHAIN does not close any of the FORTRAN logical units. When CHAINing to any other program, the user should explicitly close the opened logical units with calls to the CLOSE routine. Any routines specified in a FORTRAN USEREX library call are not executed if a CHAIN is accomplished.

Form: CALL CHAIN (dblk,var,wcnt)

where: dblk is the address of a four-word Radix-50 descriptor of the file specification for the program to be run. (See the PDP-11 FORTRAN Language Reference Manual for the format of the file specification.)

var — is the first variable (must start on a word boundary) in a sequence of variables with increasing memory addresses to be passed between programs in the chain parameter area (absolute locations 510 up to 700). A single array or a COMMON block (or portion of a COMMON block) is a suitable sequence of variables.

wcnt — is a word count (up to 60 words) specifying the number of words (beginning at var) to be passed to the called program. If no words are passed, then a word count of 0 is supplied.

If the size of the chain parameter area is insufficient, it can be increased by specifying the /B (or /BOTTOM) option to LINK for both the program executing the CHAIN call and the program receiving control.

The data passed can be accessed through a call to the RCHAIN routine. For more information on chaining to other programs, see Section 2.4.2.

Errors:

None.

Example:

The following example transfers control from the main program to PROG.SAV, on DT0, passing it variables.

```
REAL* PROGNM(2)                        !RAD50 FOR PROGRAM NAME
COMMON /BLK1/ A,B,C,D                  !DATA TO BE PASSED
DATA PROGNM/2RDTOPROG....SAV/
.
.
.
CALL CHAIN(PROGNM,A,8)                 !RUN DTO:PROG.SAV
CHAIN(PROGN,,0)                        !IF NO DATA PASSED
```

CLOSEC

4.3.3 CLOSEC

The CLOSEC subroutine terminates activity on the specified channel and frees it for use in another operation. The handler for the associated device must be in memory. CLOSEC cannot be called from a completion or interrupt routine.

Form: CALL CLOSEC (chan)

where: chan — is the channel number to be closed. This argument must be located so that the USR cannot swap over it.

A CLOSEC or PURGE must eventually be issued for any channel opened for either input or output. A CLOSEC call specifying a channel that is not open is ignored.

A CLOSEC performed on a file that was opened via an IENTER causes the device directory to be updated to make that file permanent. If the device associated with the specified channel already contains a file with the same name and type, the old copy is deleted when the new file is made permanent. A CLOSEC on a file opened via LOOKUP does not require any directory operations.

When an entered file is CLOSECed, its permanent length reflects the highest block of the file written since the file was entered; for example, if the highest block written is block number 0, the file is given a length of 1; if the file was never written, it is given a length of 0. If this length is less than the size of the area allocated at IENTER time, the unused blocks are reclaimed as an empty area on the device.

Errors:

CLOSEC does not generate any errors. If the device handler for the operation is not in memory, a fatal monitor error is generated.

Example:

The following example creates and processes a 56-block file.

```
          REAL*4 DBLK(2)
          DATA DBLK/6RSYONEW,6RFILDAT/
          DATA ISIZE/56/
          .
          .
          .
          ICHAN=IGETC()
          IF(ICHAN.LT.0) GOTO 100
          IF(IENTER(ICHAN,DBLK,ISIZE)-1) 10,110,120
    10    .
          .
          .
          CALL CLOSEC(ICHAN)
          CALL IFREEC(ICHAN)
          CALL EXIT
    100   STOP 'NO AVAILABLE CHANNELS'
    110   STOP 'CHANNEL ALREADY IN USE'
    120   STOP 'NOT ENOUGH ROOM ON DEVICE'
          END
```

CONCAT

4.3.4 CONCAT

The CONCAT subroutine is used to concatenate character strings.

Form: CALL CONCAT (a,b,out[,len[,err]])

where:

a	is the array containing the left string.
b	is the array containing the right string.
out	is the array into which the concatenated result is placed. This array must be at least one element longer than the maximum length of the resultant string (that is, one greater than the value of len, if specified).

len — is the integer number of characters representing the maximum length of the output string. The effect of len is to truncate the output string to a given length, if necessary.

err — is the logical error flag set if the output string is truncated to the length specified by len.

The string in array a immediately followed on the right by the string in array b and a terminating null character replaces the string in array out. Any combination of string arguments is allowed so long as b and out do not specify the same array. Concatenation stops either when a null character is detected in b or when the number of characters specified by len has been moved.

If either the left or right string is a null string, the other string is copied to out. If both are null strings, then out is set to a null string. The old contents of out are lost when this routine is called.

Errors:

Error conditions are indicated by err, if specified. If err is given and the output string would have been longer than len characters, then err is set to .TRUE.; otherwise, err is unchanged.

Example:

The following example concatenates the string in array STR and the string in array IN and stores the resultant string in array OUT. OUT cannot be larger than 29 characters.

```
LOGICAL*1 IN(30),OUT(30),STR(7)
.
.
.
CALL CONCAT(STR,IN,OUT,29)
```

CVTTIM

4.3.5 CVTTIM

The CVTTIM subroutine converts a two-word internal format time to hours, minutes, seconds, and ticks.

Form: CALL CVTTIM (time,hrs,min,sec,tick)

where:

time — is the two-word internal format time to be converted. If time is considered as a two-element INTEGER*2 array, then:

- time (1) is the high-order time.
- time (2) is the low-order time.

hrs — is the integer number of hours.

min — is the integer number of minutes.

sec — is the integer number of seconds.

tick	is the integer number of ticks (1/60 of a second for 60-cycle clocks; 1/50 of a second for 50-cycle clocks).

Errors:

None.

Example:

```
INTEGER*4 ITIME
.
.
.
CALL GTIM(ITIME)                        !GET CURRENT TIME-OF-DAY
CALL CVTTIM(ITIME,IHRS,IMIN,ISEC,ITCK)
IF(IHRS.GE.12) GOTO 100                 !TIME FOR LUNCH
```

DEVICE

4.3.6 DEVICE (FB and XM Only)

The DEVICE subroutine allows the user to set up a list of addresses to be loaded with specified values when the program is terminated. If a job terminates or is aborted with a CTRL/C from the terminal, this list is picked up by the system and the appropriate addresses are set up with the corresponding values.

This function is primarily designed to allow user programs to load device registers with necessary values. In particular, it is used to turn off a device's interrupt enable bit when the program servicing the device terminates.

Only one address list can be active at any given time; hence, if multiple DEVICE calls are issued, only the last one has any effect. The list must not be modified by the FORTRAN program after the DEVICE call has been issued, and the list must not be located in an overlay or an area over which the USR swaps.

The second argument of the call (link) provides support for a linked list of tables. The link argument is optional and causes the first word of the list to be processed as the link word.

Form: CALL DEVICE (ilist[,link])

where:	ilist	is an integer array containing address/value pairs, terminated by a zero word. On program termination, each value is moved to the corresponding address.
	link	an optional argument that is any value to indicate a linked list table is to be used. If the linked list form is used the first word of the array is the link list pointer.

For more information on loading values into device registers, see the assembly language .DEVICE request, Section 2.4.11.

Errors:

None.

Example:

```
INTEGER*2 IDR11(3)             !DEVICE ARRAY SPEC
DATA IDR11(1)/"167770/         !DR11 CSR ADDRESS (OCTAL)
DATA IDR11(2)/0/               !VALUE TO CLEAR INTERRUPT ENABLE
DATA IDR11(3)/0/               !AND END-OF-LIST FLAG
CALL DEVICE(IDR11)             !SET UP FOR ABORT
```

DJFLT

4.3.7 DJFLT

The DJFLT function converts an INTEGER*4 value into a REAL*8 (DOUBLE PRECISION) value and returns that result as the function value.

Form: d = DJFLT (jsrc)

where: jsrc specifies the INTEGER*4 variable which is to be converted.

Notes:

If DJFLT is used, it must be explicitly defined (REAL*8 DJFLT) or implicitly defined (IMPLICIT REAL*8 (D)) in the FORTRAN program. If this is not done, its type is assumed to be REAL*4 (single precision).

Function Results:

The function result is the REAL*8 value that is the result of the operation.

Example:

```
INTEGER*4 JVAL
REAL*8 DJFLT,D
.
.
.
D=DJFLT(JVAL)
```

GETSTR

4.3.8 GETSTR

The GETSTR subroutine reads a formatted ASCII record from a specified FORTRAN logical unit into a specified array. The data is truncated (trailing blanks removed) and a null byte is inserted at the end to form a character string.

GETSTR can be used in main program routines or in completion routines but cannot be used in both at the same time. If GETSTR is used in a completion routine, it cannot be the first I/O operation on the specified logical unit.

Form: CALL GETSTR (lun,out,len,err)

where:

lun	is the integer FORTRAN logical unit number of a formatted sequential file from which the string is to be read.
out	is the array to receive the string; this array must be one element longer than len.
len	is the integer number representing the maximum length of the string to be input.
err	is the LOGICAL*1 error flag that is set to .TRUE if an error occurred. If an error did not occur, it is .FALSE.

Errors:

Error conditions are indicated by err. If err is .TRUE, the values returned are as follows:

ERR = -1 End of file for a read operation
ERR = -2 Hard error for a read operation
ERR = -3 More than len bytes were contained in a record.

Example:

The following example reads a string of up to 80 characters from logical unit 5 into the array STRING.

```
LOGICAL*1 STRING(81),ERR
.
.
.
CALL GETSTR(5,STRING,80,ERR)
```

GTIM

4.3.9 GTIM

The GTIM subroutine allows user programs to access the current time of day. The time is returned in two words and is given in terms of clock ticks past midnight. If the system does not have a line clock, a value of 0 is returned. If an RT-11 monitor TIME command has not been entered, the value returned is the time elapsed since the system was bootstrapped, rather than the time of day.

Form: CALL GTIM (itime)

where: itime is the two-word area to receive the time of day.

The high-order time is returned in the first word, the low-order time in the second word. The SYSF4 routine CVTTIM (Section 4.3.5) can be used to convert the time into hours, minutes, seconds and ticks. CVTTIM performs the conversion based on the monitor configuration word for 50- or 60-cycle clocks (see Section 2.2.6). Under an FB or XM monitor, the time-of-day is automatically reset after 24:00 when a GTIM is executed; under the single-job monitor, it is not.

Errors:

None.

Example:

```
INTEGER*4 JTIME
.
.
.
CALL GTIM(JTIME)
```

GTJB

4.3.10 GTJB

The GTJB subroutine passes certain job parameters back to the user program.

Form: CALL GTJB (addr)

where: addr is an eight-word area to receive the job parameters. This area, considered as an eight-element INTEGER*2 array, has the following format:

addr(1)	job number. (0=background, 2=foreground)
addr(2)	high memory limit
addr(3)	low memory limit
addr(4)	beginning of I/O channel space
addr(5)-addr(8)	reserved for future use

For more information on passing job parameters, see the assembly language .GTJB request, Section 2.4.

Errors:

None.

Example:

```
      INTEGER*2 PARAMS(8)
      CALL GTJB(PARAMS)
      IF(PARAMS(1).EQ.0) TYPE 99
99    FORMAT (' THIS IS THE BACKGROUND JOB')
```

GTLIN

4.3.11 GTLIN

The GTLIN subroutine requires the USR. It transfers a line of input from the console terminal or an active indirect command file to the user program. This request is used to get information from the user, and it allows the program to operate through indirect files. The maximum size of the input line is 80 characters.

Form: CALL GTLIN (result[,prompt])

where: result — is the array receiving the string. This LOGICAL*1 array contains a maximum of 80 characters plus 0 as the end indicator.

prompt — is an optional prompt string to be printed before getting the input line. The string format is the same as that used by the PRINT subroutine.

Errors:

None

IADDR

4.3.12 IADDR

The IADDR function returns the 16-bit absolute memory address of its argument as the integer function value.

Form: i = IADDR (arg)

where: arg — is the variable, constant, or expression whose memory address is to be obtained.

Errors:

None.

Example:

IADDR can be used to find the address of an assembly language global area. For example:

```
EXTERNAL CAREA
J=IADDR(CAREA)
```

IAJFLT

4.3.13 IAJFLT

The IAJFLT function converts an INTEGER*4 value to a REAL*4 value and stores the result.

Form: i = IAJFLT (jsrc,ares)

where: jsrc — is the INTEGER*4 variable to be converted.

ares — is the REAL*4 variable or array element to receive the converted value.

Function Results:

The function result indicates the following:

i = -2	Significant digits were lost during the conversion.
= -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

```
        INTEGER*4 JVAL
        REAL*4 RESULT
        .
        .
        .
        IF(IAJFLT(JVAL,RESULT).EQ.-2) TYPE 99
99      FORMAT (' OVERFLOW IN INTEGER*4 TO REAL CONVERSION')
```

IASIGN

4.3.14 IASIGN

The IASIGN function sets information in the FORTRAN logical unit table (overriding the defaults) so that the specified information is used when the FORTRAN Object Time System (OTS) opens the logical unit. This function can be used with ICSI (see Section 4.3.18) to allow a FORTRAN program to accept a standard CSI input specification. IASIGN must be called before the unit is opened; that is, before any READ, WRITE, PRINT, TYPE, or ACCEPT statements are executed that reference the logical unit.

Form: i = IASIGN (lun,idev[,ifiltyp[,isize[,itype]]])

where:

lun	is an INTEGER*2 variable, constant, or expression specifying the FORTRAN logical unit for which information is being specified.
idev	is a one-word Radix-50 device name; this can be the first word of an ICSI input or output file specification.
ifiltyp	is a three-word Radix-50 file name and file type; this can be words 2 through 4 of an ICSI input or output file specification.
isize	is the length (in blocks) to allocate for an output file; this can be the fifth word of an ICSI output specification. If 0, the larger of either one-half the largest empty segment or the entire second largest empty segement is allocated (see Section 2.4). If the value specified for length is -1, the entire largest empty segment is allocated.

itype — is an integer value determining the optional attributes to be assigned to the file. This value is obtained by adding the values that correspond to the desired operations:

1	use double buffering for output
2	open the file as a temporary file
4	Force a LOOKUP on an existing file during the first I/O operation (otherwise, the first FORTRAN I/O operation determines how the file is opened). For example, if the next I/O operation is a write, an IENTER is performed on the specified logical unit. A read causes a LOOKUP.
8	expand carriage control information (see Notes below)
16	do not expand carriage control information
32	file is read-only

Notes:

Expanded carriage control information applies only to formatted output files and means that the first character of each record is used as a carriage control character when processing a write operation to the given logical unit. The first character is removed from the record and converted to the appropriate ASCII characters to simulate the requested carriage control.

If carriage control information is not expanded, the first character of each record is unmodified and the FORTRAN OTS outputs a line feed, followed by the record, followed by a carriage return.

If carriage control is unspecified, the FORTRAN OTS sends expanded carriage control information to the terminal and line printer and sends unexpanded carriage control information to all other devices and files. See the PDP-11 FORTRAN Language Reference Manual for further carriage control information.

Function Results:

i = 0	Normal return.
<> 0	The specified logical unit is already in use or there is no space for another logical unit association.

Example:

The following example creates an output file on logical unit 3 using the first output file given to the RT-11 Command String Interpreter (CSI), sets it up for double buffering, creates an input file on logical unit 4 based on the first input file specification given to the RT-11 CSI, and makes it available for read-only access.

```
        INTEGER*2 SPEC(39)
        REAL*4 EXT(2)
        DATA EXT/6RDATDAT,6RDATDAT/         !DEFAULT FILE TYPE IS DAT
        .
        .
        .
10      IF(ICSI(SPEC,TYP,,,0).NE.0) GOTO 10
C
C       DO NOT ACCEPT ANY SWITCHES
C
        CALL IASIGN(3,SPEC(1),SPEC(2),SPEC(5),1)
        CALL IASIGN(4,SPEC(16),SPEC(17),0,32)
```

ICDFN

4.3.15 ICDFN

The ICDFN function increases the number of input/output channels. Note that ICDFN defines new channels; the previously-defined channels are not used. Thus, an ICDFN for 20 channels (while the 16 original channels are defined) causes only 20 I/O channels to exist; the space for the original 16 is unused. The space for the new channel area is allocated out of the free space managed by the FORTRAN system.

Form: i = ICDFN (num)

where: num — is the integer number of channels to be allocated. The number of channels must be greater than 16 and can be a maximum of 256. SYSF4 can use all new channels greater than 16 without a call to IGETC; the FORTRAN system input/output uses only the first 16 channels. This argument must be positioned so that the USR cannot swap over it.

Notes:

1. ICDFN cannot be issued from a completion or interrupt routine.

2. It is recommended that the ICDFN function be used at the beginning of the main program before any I/O operations are initiated.

3. If ICDFN is executed more than once, a completely new set of channels is created each time ICDFN is called.

4. ICDFN requires that extra memory space be allocated to foreground programs (see Section 4.1.4).

Function Results:

i = 0	Normal return.
= 1	An attempt was made to allocate fewer channels than already exist.
= 2	Not enough free space is available for the channel area.

Example:

```
      IF(ICDFN(24).NE.0) STOP 'NOT ENOUGH MEMORY'
```

ICHCPY

4.3.16 ICHCPY (FB and XM Only)

The ICHCPY function opens a channel for input, logically connecting it to a file that is currently open by another job for either input or output. This function can be used by either the foreground or the background. An ICHCPY must be done before the first read or write for the given channel.

Form: i = ICHCPY (chan,ochan)

where:

	chan	is the channel the job will use to read the data.
	ochan	is the channel number of the other job that is to be copied.

Notes:

1. If the other job's channel was opened via an IENTER function or a .ENTER programmed request to create a file, the copier's channel indicates a file that extends to the highest block that the creator of the file had written at the time the ICHCPY was executed.

2. A channel that is open on a sequential-access device should not be copied, because buffer requests can become intermixed.

3. A program can write to a file (that is being created by the other job) on a copied channel just as it could if it were the creator. When the copier's channel is closed, however, no directory update takes place.

Errors:

i = 0	Normal return.
= 1	Other job does not exist or does not have the specified channel (ochan) open.
= 2	Channel (chan) is already open.

ICMKT

4.3.17 ICMKT

The ICMKT function causes one or more scheduling requests (made by an ISCHED, ITIMER or MRKT routine) to be cancelled. Support for ICMKT in SJ also requires timer support.

Form: i = ICMKT (id,time)

where:

	id	is the identification integer of the request to be cancelled. If id is equal to 0, all scheduling requests are cancelled.
	time	is the name of a 2-word area in which the monitor returns the amount of time remaining in the cancelled request.

For further information on cancelling scheduling requests, see the assembly language .CMKT request, Section 2.4.

Errors:

i = 0	Normal return.
= 1	id was not equal to 0 and no schedule request with that identification could be found.

Example:

```
INTEGER*4 J
.
.
.
CALL ICMKT(0,J)        !ABORT ALL TIMER REQUESTS NOW
.
.
.
END
```

ICSI

4.3.18 ICSI

The ICSI function calls the RT-11 Command String Interpreter in special mode to parse a command string and return file descriptors and options to the program. In this mode, the CSI does not perform any handler IFETCHes, CLOSECs, IENTERs, or LOOKUPs. An optional argument (cstring) provides ICSI with the capability of returning the original command string. This argument is allowed only when the input is from the console terminal. ICSI cannot be called from a completion or interrupt routine.

Form: i = ICSI (filspc,deftyp,[cstring],[option],x)

where: filspc is the 39-word area to receive the file specifications. The format of this area (considered as a 39-element INTEGER*2 array) is:

filspc(1)- filspc(4)	output file number 1 specification
filspc(5)	output file number 1 length
filspc(6)- filspc(9)	output file number 2 specification
filspc(10)	output file number 2 length
filspc(11)- filspc(14)	output file number 3 specification
filspc(15)	output file number 3 length
filspc(16)- filspc(19)	input file number 1 specification
filspc(20)- filspc(23)	input file number 2 specification
filspc(24)- filspc(27)	input file number 3 specification

	filspc(28)- filspc(31)	input file number 4 specification
	filspc(32)- filspc(35)	input file number 5 specification
	filspc(36)- filspc(39)	input file number 6 specification

deftyp — is the table of Radix-50 default file types to be assumed when a file is specified without a file type.

deftyp(1) is the default for all input file types.
deftyp(2) is the default file type for output file number 1.
deftyp(3) is the default file type for output file number 2.
deftyp(4) is the default file type for output file number 3.

cstring — is the area that contains the ASCIZ command string to be interpreted; the string must end in a zero byte. If the argument is omitted, the system prints the prompt character (*) at the terminal and accepts a command string.

option — is the name of an INTEGER*2 array dimensioned (4,n) where n represents the number of options defined to the program. This argument must be present if the value specified for "x" is non-zero. This array has the following format for the nth option described by the array.

option(1,n) is the one-character ASCII name of the option.
option(2,n) is set by the routine to 0, if the option did not occur; to 1, if the option occurred without a value; to 2, if the option occurred with a value.
option(3,n) is set to the file number on which the option is specified.
option(4,n) is set to the specified value if option(2,n) is equal to 2.

x — is the number of options defined in the array "option".

Notes:

The array "option" must be set up to contain the names of the valid options. For example, use the following to set up names for five options:

```
INTEGER*2 SW(4,5)
DATA SW(1,1)/'S'/,SW(1,2)/'M'/,SW(1,3)/'I'/
DATA SW(1,4)/'L'/,SW(1,5)/'E'/
```

Multiple occurrences of the same option are supported by allocating an entry in the option array for each occurrence of the option. Each time the option occurs in the option array, the next unused entry for the named option is used.

The arguments of ICSI must be positioned so that the USR cannot swap over them.

For more information on calling the Command String Interpreter, see the assembly language .CSISPC request, Section 2.4.

Errors:

i	= 0	Normal return.
	= 1	Illegal command line; no data was returned.
	= 2	An illegal device specification occurred in the string.
	= 3	An illegal option was specified, or a given option was specified more times than were allowed for in the option array.

Example:

The following example causes the program to loop until a valid command is typed at the console terminal.

```
        INTEGER*2 SPEC(39)
        REAL*4 EXT(2)
        DATA EXT/6RDATDAT,6RDATDAT/
        .
        .
        .
10      TYPE 99
99      FORMAT (' ENTER VALID CSI STRING WITH NO OPTIONS')
        IF(ICSI(SPEC,EXT,,,0).NE.0) GOTO 10
```

ICSTAT

4.3.19 ICSTAT (FB and XM Only)

The ICSTAT function furnishes the user with information about a channel. It is supported only in the FB or XM environment; no information is returned when operating under the single-job monitor.

Form: i = ICSTAT (chan,addr)

where:

chan — is the channel whose status is desired.

addr — is a six-word area to receive the status information. The area, as a six-element INTEGER*2 array, has the following format.

addr(1)	channel status word (see Section 2.4)
addr(2)	starting absolute block number of file on this channel
addr(3)	length of file
addr(4)	highest block number written since file was opened (see Section 2.4)

addr(5)	unit number of device with which this channel is associated
addr(6)	Radix-50 of device name with which the channel is associated

Errors:

i = 0	Normal return.
= 1	Channel specified is not open.

Example:

The following example obtains channel status information about channel I.

```
      INTEGER*2 AREA(6)
      I=7
      IF(ICSTAT(I,AREA).NE.0) TYPE 99,I
99    FORMAT(1X,'CHANNEL',I4,'IS NOT OPEN')
```

IDELET

4.3.20 IDELET

The IDELET function deletes a named file from an indicated device. Since this routine passes information to the USR, provisions must be made to prevent information required by the USR from being swapped. This is accomplished by moving all parameters to the stack and pointing to these parameters in the program request. IDELET cannot be issued from a completion or interrupt routine.

Form: i = IDELET (chan,dblk[,seqnum])

where:

chan — is the channel to be used for the delete operation.

dblk — is the four-word Radix-50 specification (dev:filnam.typ) for the file to be deleted.

seqnum — file number for cassette operations: if this argument is blank, a value of 0 is assumed.

For magtape operation, it describes a file sequence number that can have the following values:

Value	Meaning
-1	For LOOKUP or IDELET, this value suppresses rewinding and searching for a file name from the current tape position. Note that if the position is unknown, the handler executes a positioning algorithm that involves backspacing until an end-of-file label is found. The user should not use any other value since all other negative values are reserved for future use.

0 For LOOKUP or IDELET, this value rewinds the magtape and spaces forward until the file name is found. For .ENTER it rewinds the magtape and spaces forward until the file name is found or until the logical end of tape is detected. If the file name is found, it is deleted and tape search continues.

n Where n is any positive number. This value positions the magtape at file sequence number n. If the file represented by the file sequence number is greater than two files away from the beginning of tape, a rewind is performed. If not, the tape is backspaced to the file.

NOTE

The arguments of IDELET must be located so that the USR cannot swap over them.

The specified channel is left inactive when the IDELET is complete. IDELET requires that the handler to be used be resident (via an IFETCH call) at the time the IDELET is issued. If it is not, a monitor error occurs.

For further information on deleting files, see the assembly language .DELETE request, Section 2.4.

Errors:

i = 0	Normal return.
= 1	Channel specified is already open.
= 2	File specified was not found.
= 3	Device is use

Example:

The following example deletes a file named FTN5.DAT from SY0.

```
REAL*4 FILNAM(2)
DATA FILNAM/6RSYOFTN,6R5  DAT/
.
.
.
I=IGETC()
IF(I.LT.0) STOP 'NO CHANNEL'
CALL IDELET(I,FILNAM)
CALL IFREEC(I)
```

IDJFLT

4.3.21 IDJFLT

The IDJFLT function converts an INTEGER*4 value into a REAL*8 (DOUBLE PRECISION) value and stores the result.

Form: i = IDJFLT (jsrc,dres)

where: jsrc specifies the INTEGER*4 variable that is to be converted.

dres specifies the REAL*8 (or DOUBLE PRECISION) variable to receive the converted value.

Function Results:

The function result indicates the following:

i = -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

```
        INTEGER*4 JJ
        REAL*8 DJ
        .
        .
        .
        IF(IDJFLT(JJ,DJ).LE.0) TYPE 99
99      FORMAT (' VALUE IS NOT POSITIVE')
```

IDSTAT

4.3.22 IDSTAT

The IDSTAT function is used to obtain information about a particular device. IDSTAT cannot be issued from a completion or interrupt routine.

Form: i = IDSTAT (devnam,cblk)

where: devnam is the Radix-50 device name.

cblk is the four-word area used to store the status information. The area, as a four-element INTEGER*2 array, has the following format:

cblk(1)	device status word (see Section 2.4.13)
cblk(2)	size of handler in bytes
cblk(3)	entry point of handler (non-zero implies that the handler is in memory)
cblk(4)	size of the device (in 256-word blocks) for block-replaceable devices; zero for sequential-access devices

NOTE

The arguments of IDSTAT must be positioned so that the USR cannot swap over them.

IDSTAT looks for the device specified by devnam and, if found, returns four words of status in cblk. Errors:

i = 0	Normal return.
= 1	Device not found in monitor tables.

Example:

The following example determines whether the line printer handler is in memory. If it is not, the program stops and prints a message to indicate that the handler must be loaded.

```
REAL*4 IDNAM
INTEGER*2 CBLK(4)
DATA IDNAM/3RLP /
DATA CBLK/4*0/
CALL IDSTAT(IDNAM,CBLK)
IF(CBLK(3).EQ.0) STOP 'LOAD THE LP HANDLER AND RERUN'
```

IENTER

4.3.23 IENTER

The IENTER function allocates space on the specified device and creates a tentative directory entry for the named file. If a file of the same name already exists on the specified device, it is not deleted until the tentative entry is made permanent by CLOSEC. The file is attached to the channel number specified.

Form: i = IENTER (chan,dblk,length[,seqnum])

where:

chan — is the integer specification for the RT-11 channel to be associated with the file.

dblk — is the four-word Radix-50 descriptor of the file to be operated upon.

length — is the integer number of blocks to be allocated for the file. If 0, the larger of either one-half the largest empty segment or the entire second largest empty segment is allocated (see Section 2.4.14). If the value specified for length is -1, the entire largest empty segment is allocated.

seqnum — file number for cassette. If this argument is blank, a value of 0 is assumed.

For magtape, it describes a file sequence number that can have the following values:

-1 - means space to the logical-end-of-tape and enter file

0 - means rewind the magtape and space forward until the file name is found or the logical-end-of-tape is detected. If file name is found, delete it and continue tape search.

n - means position magtape at file sequence number n. If the file represented by the file sequence number is greater than two files away from beginning of tape, then a rewind is performed. If not, the tape is backspaced to the file.

Notes:

1. IENTER cannot be issued from a completion or interrupt routine.

2. IENTER requires that the appropriate device handler be in memory.

3. The arguments of IENTER must be positioned so that the USR does not swap over them.

For further information on creating tentative directory entries, see the assembly language .ENTER request, Section 2.4.

Errors:

i = n	Normal return; number of blocks actually allocated (n = 0 for nonfile-structured IENTER).
= -1	Channel (chan) is already in use.
= -2	In a fixed-length request, no space greater than or equal to length was found.
= -3	Device in use
= -4	File sequence number not found

Example:

The following example allocates a channel for file TEMP.TMP on Y0. If no channel is available, the program prints a message and halts.

```
      REAL*4 DBLK(2)
      DATA DBLK/6RSYOTEM,6RP  TMP/
      ICHAN=IGETC()
      IF(ICHAN.LT.0) STOP 'NO AVAILABLE CHANNEL'
C
C     CREATE TEMPORARY WORK FILE
C
      IF(IENTER(ICHAN,DBLK,20).LT.0) STOP 'ENTER FAILURE'
      .
      .
      .
      CALL PURGE(ICHAN)
      CALL IFREEC(ICHAN)
```

IFETCH

4.3.24 IFETCH

The IFETCH function loads a device handler into memory from the system device, making the device available for input/output operations. The handler is loaded into the free area managed by the FORTRAN system. Once the handler is loaded, it cannot be released and the memory in which it resides cannot be reclaimed. IFETCH cannot be issued from a completion or interrupt routine.

Form: i = IFETCH (devnam)

where: devnam is the one-word Radix-50 name of the device for which the handler is desired. This argument can be the first word of an ICSI input or output file specification. This argument must be positioned so that the USR cannot swap over it.

For further information on loading device handlers into memory, see the assembly language .FETCH request, Section 2.4.16.

Errors:

i = 0	Normal return.
= 1	Device name specified does not exist.
= 2	Not enough room exists to load the handler.
= 3	No handler for the specified device exists on the system device.

Example:

The following example requests the DX1 handler to be loaded into memory; execution stops if the handler cannot be loaded.

```
REAL*4 IDNAM
DATA IDNAM/3RDX1/
.
.
.
IF (IFETCH(IDNAM).NE.0) STOP 'FATAL ERROR FETCHING HANDLER'
```

IFREEC

4.3.25 IFREEC

The IFREEC function returns a specified RT-11 channel to the available pool of channels. Before IFREEC is called, the specified channel must be closed or deactivated with a CLOSEC (see Section 4.3.3) or a PURGE (see Section 4.3.66) call. IFREEC cannot be called from a completion or interrupt routine. IFREEC calls must be issued only for channels that have been successfully allocated by IGETC calls; otherwise, the results are unpredictable.

Form: i = IFREEC (chan)

where: chan is the integer number of the channel to be freed.

Errors:

i = 0	Normal return.
= 1	Specified channel is not currently allocated.

Example:

See the example under IGETC, (Section 4.3.26).

4.3.26 IGETC

The IGETC function allocates an RT-11 channel (in the range 0-17 octal) and marks it in use for the FORTRAN I/O system. IGETC cannot be issued from a completion or interrupt routine.

Form: i = IGETC()

Function Results:

i = -1	No channels are available.
= n	Channel n has been allocated.

Example:

```
ICHAN=IGETC()                !ALLOCATE CHANNEL
IF(ICHAN.LT.0) STOP 'CANNOT ALLOCATE CHANNEL'
.
.
.
CALL IFREEC(ICHAN)           !FREE IT WHEN THROUGH
.
.
.
END
```


4.3.27 IGETSP

The IGETSP subroutine returns the address and size (number of words) of free space obtained from the FORTRAN system. When this space is obtained, it is allocated for the duration of the program.

Form: i = IGETSP (min,max,iaddr)

where:

min	is the minimum space to be obtained without an error indicating that the desired amount of space is not available.
max	is the maximum space to be obtained without an error indicating that the desired amount of space is not available.
iaddr	is the integer specifying the address of the start of the free space (buffer).

Function Results:

i = -1	Error: not enough free space is available to meet the minimum requirements; no allocation was taken from the FORTRAN system free space.
i = n	is the actual size allocated whose value is min .GE. n .LE. max.

The size (min, max, n) is specified in words. Extreme caution should be exercised to avoid using all of the free space allocated by the FORTRAN system. If the FORTRAN system runs out of dynamic free space, fatal errors (Error 29, 30, 42, etc.) occur.

IJCVT

4.3.28 IJCVT

The IJCVT function converts an INTEGER*4 value to INTEGER*2 format. If ires is not specified, the result returned is the INTEGER*2 value of jsrc. If ires is specified, the result is stored there.

Form: i = IJCVT (jsrc[,ires])

where:

where:	jsrc	specifies the INTEGER*4 variable or array element whose value is to be converted.
	ires	specifies the INTEGER*2 entity to receive the conversion result.

Function results if ires is specified:

i = -2	An overflow occurred during conversion.
= -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

```
      INTEGER*4 JVAL
      INTEGER*2 IVAL
      .
      .
      .
      IF(IJCVT(JVAL,IVAL).EQ.-2) TYPE 99
99    FORMAT(' NUMBER TOO LARGE IN IJCVT CONVERSION')
```

ILUN

4.3.29 ILUN

The ILUN function returns the RT-11 channel number with which a FORTRAN logical unit is associated.

Form: i = ILUN (lun)

where:	lun	is an integer expression whose value is a FORTRAN logical unit number in the range 1-99.

Function Results:

i = -1	Logical unit is not open.
= -2	Logical unit is opened to console terminal.
= +n	RT-11 channel number n is associated with lun.

Example:

```
      PRINT 99
99    FORMAT(' PRINT DEFAULTS TO LOGICAL UNIT 6, WHICH FURTHER DEFAULTS TO LP:')
      LUNRT=ILUN(6)                  !WHICH RT-11 CHANNEL IS RECEIVING I/O?
```

INDEX

4.3.30 INDEX

The INDEX subroutine searches a string for the occurrence of another string and returns the character position of the first occurrence of that string.

Form: CALL INDEX (a,pattrn,[i],m)

or

m = INDEX (a,pattrn[,i])

where:

	a	is the array containing the string to be searched.
	pattrn	is the string being sought.
	i	is the integer starting character position of the search in a. If i is omitted, a is searched beginning at the first character position.
	m	is the integer result of the search; m equals the starting character position of pattrn in a, if found; otherwise it is 0.

Errors:

None.

Example:

The following example searches the array STRING for the first occurrence of strings EFG and XYZ and searches the string ABCABCABC for the occurrence of string ABC after position 5.

```
      CALL SCOPY('ABCDEFGHI',STRING)      !INITIALIZE STRING
      CALL INDEX(STRING,'EFG',,M)         !M=5
      CALL INDEX(STRING,'XYZ',,N)         !N=0
      CALL INDEX('ABCABCABC','ABC',5,L)   !L=7
```

INSERT

4.3.31 INSERT

The INSERT subroutine replaces a portion of one string with another string.

Form: CALL INSERT (in,out,i[,m])

Example:

```
     EXTERNAL CLKSUB                      !SUBR TO HANDLE KW11-P CLOCK
     .
     .
     .
     I=INTSET("104,6,0,CLKSUB)           !ATTACH ROUTINE
     IF (I.NE.0) GOTO 100                 !BRANCH IF ERROR
     .
     .
     .
     END
     SUBROUTINE CLKSUB(ID)
     .
     .
     .
     END
```

IPEEK

4.3.33 IPEEK

The IPEEK function returns the contents of the word located at a specified absolute 16-bit memory address. This function can be used to examine device registers or any location in memory.

Form: i = IPEEK (iaddr)

where: iaddr is the integer specification of the absolute address to be examined. If this argument is not an even value, a trap results.

Function Result:

The function result (i) is set to the value of the word examined.

Example:

```
     ISWIT = IPEEK("177570)              !GET VALUE OF CONSOLE SWITCHES
```

IPEEKB

4.3.34 IPEEKB

The IPEEKB subroutine returns the contents of a byte located at a specified absolute byte address. Since this routine operates in a byte mode, the address supplied can be even or odd. This subroutine can be used to examine device registers or any byte in memory.

Form: i = IPEEKB (iaddr)

where: iaddr is the integer specification of the absolute byte address to be examined. Unlike the IPEEK subroutine, the IPEEKB subroutine allows odd addresses.

Function Result:

The function result (i) is set to the value of the byte examined.

Example:

```
IERR = IPEEKB('53)  !Get error byte
```

IPOKE

4.3.35 IPOKE

The IPOKE subroutine stores a specified 16-bit integer value into a specified absolute memory location. This subroutine can be used to store values in device registers.

Form: CALL IPOKE (iaddr,ivalue)

where:

	iaddr	is the integer specification of the absolute address to be modified. If this argument is not an even value, a trap results.
	ivalue	is the integer value to be stored in the given address (iaddr).

Errors:

None.

Example:

The following example displays the value of IVAL in the console display register.

```
CALL IPOKE('177570,IVAL)
```

To set bit 12 in the JSW without zeroing any other bits in the JSW, use the following procedure.

```
CALL IPOKE('44,'10000.OR.IPEEK('44))
```

IPOKEB

4.3.36 IPOKEB

The IPOKEB subroutine stores a specified eight-bit integer value into a specified byte location. Since this routine operates in a byte mode, the address supplied can be even or odd. This subroutine can be used to store values in device registers.

Form: CALL IPOKEB (iaddr,ivalue)

where:

	iaddr	is the integer specification of the absolute address to be modified. Unlike the IPOKE subroutine, the IPOKEB subroutine allows odd addresses.
	ivalue	is the integer value to be stored in the given address specified by the iaddr argument.

Errors:

None

Example: (see section 4.3.35)

IQSET

4.3.37 IQSET

The IQSET function is used to make the RT-11 queue larger (that is, to add available elements to the queue). These elements are allocated out of the free space managed by the FORTRAN system. IQSET cannot be called from a completion or interrupt routine.

Form: i = IQSET (qleng)

where: qleng is the integer number of elements to be added to the queue. This argument must be positioned so that the USR does not swap over it.

All RT-11 I/O transfers are done through a centralized queue management system. If I/O traffic is very heavy and not enough queue elements are available, the program issuing the I/O requests can be suspended until a queue element becomes available. In an FB or XM system, the other job runs while the first program waits for the element. When IQSET is used in a program to be run in the foreground, the FRUN command must be modified to allocate space for the queue elements (see Section 4.1.4).

A general rule to follow is that each program should contain one more queue element than the total number of I/O and timer requests that will be active simultaneously. Timing functions such as ITWAIT and MRKT also cause elements to be used and must be considered when allocating queue elements for a program. Note that if synchronous I/O is done (IREADW/IWRITW, etc.) and no timing functions are done, no additional queue elements need be allocated. Note also that FORTRAN IV allocates four queue elements. See Section 4.2 for a list of SYSF4 calls that use queue elements.

For further information on adding elements to the queue, see the assembly language .QSET request, Section 2.4.

Function Results:

i = 0	Normal return.
= 1	Not enough free space is available for the number of queue elements to be added; no allocation was made.

Example:

```
IF(IQSET(5).NE.0) STOP 'NOT ENOUGH FREE SPACE FOR QUEUE ELEMENTS'
```

IRAD50

4.3.38 IRAD50

The IRAD50 function converts a specified number of ASCII characters to Radix-50 and returns the number of characters converted. Conversion stops on the first non-Radix-50 character encountered in the input or when the specified number of ASCII characters have been converted.

Form: n = IRAD50 (icnt,input,output)

where:

icnt	is the number of ASCII characters to be converted.
input	is the area from which input characters are taken.
output	is the area into which Radix-50 words are stored.

Three characters of text are packed into each word of output. The number of output words modified is computed by the expression (in integer words):

(icnt+2)/3

Thus, if a count of 4 is specified, two words of output are written even if only a one-character input string is given as an argument.

Function Results:

The integer number of input characters actually converted (n) is returned as the function result.

Example:

```
REAL*8 FSPEC
CALL IRAD50(12,'SYOTEMP  DAT',FSPEC)
```

IRCVD/IRCVDC/IRCVDF/IRCVDW

4.3.39 IRCVD/IRCVDC/IRCVDF/IRCVDW(FB and XM Only)

There are four forms of the receive data function; these are used in conjunction with the ISDAT (send data) functions to allow a general data/message transfer system. The receive data functions issue RT-11 receive data programmed requests (see Section 2.4). These functions require a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

IRCVD

The IRCVD function is used to receive data and continue execution. The operation is queued and the issuing job continues execution. At some point when the job must receive the transmitted message, an MWAIT should be executed. This causes the job to be suspended until the message has been received.

Form: i = IRCVD (buff,wcnt)

where: buff — is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVD is complete.

wcnt — is the maximum integer number of words that can be received.

Errors:

i = 0 Normal return.
= 1 No other job exists in the system.

Example:

```
INTEGER*2 MSG(41)
.
.
.
CALL IRCVD(MSG,40)
.
.
.
CALL MWAIT
```

IRCVDC

The IRCVDC function receives data and enters an assembly language completion routine when the message is received. The IRCVDC is queued and program execution stays with the issuing job. When the other job sends a message, the completion routine specified is entered.

Form: i = IRCVDC (buff,wcnt,crtn)

where: buff — is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDC is complete.

wcnt — is the maximum integer number of words to be received.

crtn — is the assembly language completion routine to be entered. This name must be specified in a FORTRAN EXTERNAL statement in the routine that issues the IRCVDC call.

Errors:

i = 0 Normal return.
= 1 No other job exists in the system.

IRCVDF

The IRCVDF function receives data and enters a FORTRAN completion subroutine (see Section 4.2.1) when the message is received. The

IRCVDF is queued and program execution continues with the issuing job. When the other job sends a message, the FORTRAN completion routine specified is entered.

Form: i = IRCVDF (buff,wcnt,area,crtn)

where:

buff
: is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDF is complete.

wcnt
: is the maximum integer number of words to be received.

area
: is a four-word area to be set aside for linkage information. This area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion routines when crtn has been entered.

crtn
: is the FORTRAN completion routine to be entered. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IRCVDF call.

Errors:

i =	0	Normal return.
=	1	No other job exists in the system.

Example:

```
INTEGER*2 MSG(41),AREA(4)
EXTERNAL RMSGRT
.
.
.
CALL IRCVDF(MSG,40,AREA,RMSGRT)
```

IRCVDW

The IRCVDW function is used to receive data and wait. This function queues a message request and suspends the job issuing the request until the other job sends a message. When execution of the issuing job resumes, the message has been received, and the first word of the buffer indicates the number of words transmitted.

Form: i = IRCVDW (buff,wcnt)

where:

buff
: is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDW is complete.

wcnt
: is the maximum integer number of words to be received.

Errors:

i = 0	Normal return.
= 1	No other job exists in the system.

Example:

```
INTEGER*2 MSG(41)
IF(IRCVDW(MSG,40).NE.0) STOP 'UNEXPECTED ERROR'
```

IREAD/IREADC/IREADF/IREADW

4.3.40 IREAD/IREADC/IREADF/IREADW

SYSF4 provides four modes of I/O: IREAD/IWRITE, IREADC/IWRITC, IREADF/IWRITF, and IREADW/IWRITW. These functions require a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

IREAD

The IREAD function transfers a specified number of words from the file (first block of file = 0) associated with the indicated channel into memory. Control returns to the user program immediately after the IREAD function is initiated. No special action is taken when the transfer is completed.

Form: i = IREAD (wcnt,buff,blk,chan)

where:

wcnt	is the relative integer number of words to be transferred.
buff	is the array to be used as the buffer. This array must contain at least wcnt words.
blk	is the relative integer block number of the file to be read. The user program normally updates blk before it is used again.
chan	is the relative integer specification for the RT-11 channel to be used.

NOTE

The blk argument must be updated, if necessary, by the user program. For example, if the program is reading two blocks at a time, blk should be updated by 2.

When the user program needs to access the data read on the specified channel, an IWAIT function should be issued. This ensures that the IREAD operation has been completed. If an error occurred during the transfer, the IWAIT function indicates the error.

Errors:

i = n	Normal return; n equals the number of words read (0 for non-file-structured read, multiple of 256 for file-structured read). For example: If wcnt is a multiple of 256 and less than that number of words remain in the file, n is shortened to the number of words that remain in the file; for example if wcnt is 512 and only 256 words remain, i = 256. If wcnt is not a multiple of 256 and more than wcnt words remain in the file, n is rounded up to the next block; for example, if wcnt is 312 and more than 312 words remain, i = 512, but only 312 are read. If wcnt is not a multiple of 256 and less than wcnt words remain in the file, n equals a multiple of 256 that is the actual number of words being read.
= -1	Attempt to read past end-of-file; no words remain in the file.
= -2	Hardware error occurred on channel.
= -3	Specified channel is not open.

Example:

```
      INTEGER*2 BUFFER(256),RCODE,BLK
      .
      .
      .
      RCODE = IREAD(256,BUFFER,BLK,ICHAN)
      IF(RCODE+1) 1010,1000,10
C     IF NO ERROR, START HERE
10    .
      .
      .
      IF(IWAIT(ICHAN).NE.0) GOTO 1010
      .
      .
      .
1000  CONTINUE
C     END OF FILE PROCESSING
      .
      .
      .
      CALL EXIT             !NORMAL END OF PROGRAM
1010  STOP 'FATAL READ'
      END
```

IREADC

The IREADC function transfers a specified number of words from the indicated channel into memory. Control returns to the user program immediately after the IREADC function is initiated. When the operation is complete, the specified assembly language routine (crtn) is entered as an asynchronous completion routine.

Form: i = IREADC (wcnt,buff,blk,chan,crtn) where: wcnt is the integer number of words to be transferred.

buff is the array to be used as the buffer. This array must contain at least wcnt words.

blk is the integer block number of the file to be read. The user program normally updates blk before it is used again.

chan is the integer specification for the RT-11 channel to be used.

crtn is the assembly language routine to be activated when the transfer is complete. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IREADC call.

Errors:

See Errors under IREAD.

Example:

```
INTEGER*2 IBUF(256),RCODE,IBLK
EXTERNAL RDCMP
.
.
.
RCODE=IREADC(256,IBUF,IBLK,ICHAN,RDCMP)
```

IREADF

The IREADF function transfers a specified number of words from the indicated channel into memory. Control returns to the user program immediately after the IREADF function is initiated. When the operation is complete, the specified FORTRAN subprogram (crtn) is entered as an asynchronous completion routine (see Section 4.2.1).

Form: i = IREADF (wcnt,buff,blk,chan,area,crtn)

where: wcnt is the integer number of words to be transferred.

buff is the array to be used as the buffer. This array must contain at least wcnt words.

blk is the integer block number of the file to be used. The user program normally updates blk before it is used again.

chan is the integer specification for the RT-11 channel to be used.

area is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program or swapped over by the USR. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.

crtn — is the FORTRAN routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the routine that issues the IREADF call. The subroutine has two arguments:

SUBROUTINE crtn (iargl,iarg2)

iargl — is the channel status word (see Section 2.4.34) for the operation just completed. If bit 0 is set, a hardware error occurred during the transfer.

iarg2 — is the octal channel number used for the operation just completed.

Errors:

See Errors under IREAD.

Example:

```
      INTEGER*2 DBLK(4),BUFFER(256),BLKNO
      DATA DBLK/3RDX0,3RINP,3RUT ,3RDAT/,BLKNO/0/
      EXTERNAL RCMPLT
      .
      .
      .
      ICHAN=IGETC()
      IF(ICHAN.LT.0) STOP 'NO CHANNEL AVAILABLE'
      IF(IFETCH(DBLK).NE.0) STOP 'BAD FETCH'
      IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
      .
      .
      .
20    IF(IREADF(256,BUFFER,BLKNO,ICHAN,DBLK,RCMPLT).LT.0) GOTO 100
C     PERFORM OVERLAP PROCESSING
      .
      .
      .
C     SYNCHRONIZER
      CALL IWAIT(ICHAN)   !WAIT FOR COMPLETION ROUTINE TO RUN
      BLKNO=BLKNO+1       !UPDATE BLOCK NUMBER
      GOTO 20
      .
      .
      .
C     END OF FILE PROCESSING
100   CALL CLOSEC(ICHAN)
      CALL IFREEC(ICHAN)
      .
      .
      .
      CALL EXIT
      END
      SUBROUTINE RCMPLT(I,J)
C     THIS IS THE COMPLETION ROUTINE
      .
      .
      .
      RETURN
      END
```

IREADW

The IREADW function transfers a specified number of words from the indicated channel into memory. Control returns to the user program when the transfer is complete or when an error is detected.

Form: i = IREADW (wcnt,buff,blk,chan)

where:

wcnt	is the integer number of words to be transferred.
buff	is the array to be used as the buffer. This array must contain at least wcnt words.
blk	is the integer block number of the file to be read. The user program normally updates blk before it is used again.
chan	is the integer specification for the RT-11 channel to be used.

Errors:

See Errors under IREAD.

Example:

```
      INTEGER*2 IBUF(1024)
      .
      .
      .
      ICODE=IREADW(1024,IBUF,IBLK,ICHAN)
      IF(ICODE.EQ.-1) GOTO 100       !END OF FILE PROCESSING AT 100
      IF(ICODE.LT.-1) GOTO 200       !ERROR PROCESSING AT 200
C
C     MODIFY BLOCKS
C
      .
      .
      .
C
C     WRITE THEM OUT
C
      ICODE=IWRITW(1024,IBUF,IBLK,ICHAN)
```

IRENAM

4.3.41 IRENAM

The IRENAM function causes an immediate change of the name of a specified file. An error occurs if the channel specified is already open.

Form: i = IRENAM (chan,dblk)

where:

chan	is the integer specification for the RT-11 channel to be used for the operation.

dblk — is the eight-word area specifying the name of the existing file and the new name to be assigned. If considered as an eight-element INTEGER*2 array, dblk has the form:

dblk(1)-dblk(4)	specify the Radix-50 file descriptor for the old file name.
dblk(5)-dblk(8)	specify the Radix-50 file descriptor for the new file name.

NOTE

The arguments of IRENAM must be positioned so that the USR does not swap over them.

If a file already exists with the same name as the new file on the indicated device, it is deleted. The specified channel is left closed when the **IRENAM** is complete. IRENAM requires that the handler to be used be resident at the time the **IRENAM** is issued. If it is not, a monitor error occurs. The device names specified in the file descriptors must be the same.

For more information on renaming files, see the assembly language **.RENAME** request, Section 2.4.

Errors:

i =	0	Normal return.
=	1	Specified channel is already open.
=	2	Specified file was not found. Example:

```
REAL*8 NAME(2)
DATA NAME/12RDKOFTN2  DAT,12RDKOFTN2  OLD/
.
.
.
ICHAN=IGETC()
IF(ICHAN.LT.0) STOP 'NO CHANNEL'
CALL IRENAM(ICHAN,NAME)     !PRESERVE OLD DATA FILE
CALL IFREEC(ICHAN)
```

IREOPN

4.3.42 IREOPN

The IREOPN function reassociates a specified channel with a file on which an ISAVES was performed. The ISAVES/IREOPN combination is useful when a large number of files must be operated on at one time. Necessary files can be opened with LOOKUP and their status preserved with ISAVES. When data is required from a file, an IREOPN enables the program to read from the file. The IREOPN need not be done on the same channel as the original LOOKUP and ISAVES.

Form: i = IREOPN (chan,cblk)

where: chan is the integer specification for the RT-11 channel to be associated with the reopened file. This channel must be initially inactive.

cblk is the five-word block where the channel status information was stored by a previous ISAVES. This block, considered as a five-element INTEGER*2 array, has the following format:

cblk(1)	Channel status word (see Section 2.4).
cblk(2)	Starting block number of the file; zero for non-file-structured devices.
cblk(3)	Length of file (in 256-word blocks).
cblk(4)	(Reserved for future use.)
cblk(5)	Two information bytes. Even byte: I/O count of the number of requests made on this channel. Odd byte: unit number of the device associated with the channel.

Errors:

i = 0	Normal return.
= 1	Specified channel is already in use.

Example:

```
INTEGER*2 SAVES(5,10)
DATA ISVPTR/1/
.
.
.
CALL ISAVES(ICHAN,SAVES(1,ISVPTR))
.
.
.
CALL IREOPN(ICHAN,SAVES(1,ISVPTR))
```

ISAVES

4.3.43 ISAVES

The ISAVES function stores five words of channel status information into a user-specified array. These words contain all the information that RT-11 requires to completely define a file. When an ISAVES is finished, the data words are placed in memory and the specified channel is closed and is again available for use. When the saved channel data is required, the IREOPN function (Section 4.3.42) is used.

ISAVES can be used only if a file was opened with a LOOKUP call (see Section 4.3.70). If IENTER was used, ISAVES is illegal and returns an error. Note that ISAVES is not legal on magtape or cassette files.

Form: i = ISAVES (chan,cblk)

where: chan is the integer specification for the RT-11 channel whose status is to be saved.

cblk is a five-word block into which the channel status information describing the open file is stored. See Section 4.3.42 for the format of this block.

The ISAVES/IREOPN combination is very useful, but care must be exercised when using it. In particular, the following cases should be avoided.

1. If an ISAVES is performed on a file and the same file is then deleted before it is reopened, the space occupied by the file becomes available as an empty space which could then be used by the IENTER function. If this sequence occurs, the contents of the file whose status was supposedly saved changes.

2. Although the handler for the required peripheral need not be in memory for execution of an IREOPN, a fatal error is generated if the handler is not in memory when an IREAD or IWRITE is executed.

Errors:

i = 0	Normal return.
= 1	The specified channel is not currently associated with any file.
= 2	The file was opened with an IENTER call; an ISAVES is illegal.

Example:

```
INTEGER*2 BLK(5)
.
.
.
IF(ISAVES(ICHAN,BLK).NE.0) STOP 'ISAVES ERROR'
```

ISCHED

4.3.44 ISCHED

The ISCHED function schedules a specified FORTRAN subroutine to be run as an asynchronous completion routine at a specified time of day. Support for ISCHED in SJ also requires timer support.

Form: i = ISCHED (hrs,min,sec,tick,area,id,crtn)

where: hrs is the integer number of hours.

min is the integer number of minutes.

sec is the integer number of seconds.

tick is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks).

area — is a four-word area that must be provided for link information; this area must never be modified by the FORTRAN program, and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.

id — is the identification integer to be passed to the routine being scheduled.

crtn — is the name of the FORTRAN subroutine to be entered at the time of day specified. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISCHED call. The subroutine has one argument. For example:

```
SUBROUTINE crtn(id)
INTEGER id
```

When the routine is entered, the value of the integer argument is the value specified for id in the appropriate ISCHED call.

Notes:

1. The scheduling request made by this function can be cancelled at a later time by an ICMKT function call.

2. If the system is busy, the actual time of day that the completion routine is run can be greater than the requested time of day.

3. A FORTRAN subroutine can periodically reschedule itself by issuing its own ISCHED or ITIMER calls from within the routine.

4. ISCHED requires a queue element; this should be considered when the IQSET function (Section 4.3.33) is executed.

Errors:

i = 0		Normal return.
= 1		No queue elements available; unable to schedule request.

Example:

```
      INTEGER*2 LINK(4)                   !LINKAGE AREA
      EXTERNAL NOON                       !NAME OF ROUTINE TO RUN
      .
      .
      .
      I=ISCHED(12,0,0,0,LINK,0,NOON)      !RUN SUBR NOON AT 12 PM
      .
      .   (rest of main program)
      .
      END
      SUBROUTINE NOON(ID)
C
C     THIS ROUTINE WILL TERMINATE EXECUTION AT LUNCHTIME,
C     IF THE JOB HAS NOT COMPLETED BY THAT TIME.
C
      STOP      'ABORT JOB -- LUNCHTIME'
      END
```

4.3.45 ISDAT/ISDATC/ISDATF/ISDATW (FB and XM Only)

These functions are used with the IRCVD/IRCVDC/IRCVDF, and IRCVDW calls to allow message transfers under the FB or monitor. Note that the buffer containing the message should not be modified or reused until the message has been received by the other job. These functions require a queue element; this should be considered when the IQSET function (see Section 4.3.37) is executed.

ISDAT

The ISDAT function transfers a specified number of words from one job to the other. Control returns to the user program immediately after the transfer is queued. This call is used with the MWAIT routine (see Section 4.3.80).

Form: i = ISDAT (buff,wcnt)

where:

buff	is the array containing the data to be transferred.
wcnt	is the integer number of data words to be transferred.

Errors:

i = 0	Normal return.
= 1	No other job currently exists in the system.

Example:

```
      INTEGER*2 MSG(40)
      .
      .
      .
      CALL ISDAT(MSG,40)
      .
      .
      .
      CALL MWAIT
C     PUT NEW MESSAGE IN BUFFER
```

ISDATC

The ISDATC function transfers a specified number of words from one job to another. Control returns to the user program immediately after the transfer is queued. When the other job accepts the message through a receive data request, the specified assembly language routine (crtn) is activated as an asynchronous completion routine.

Form: i = ISDATC (buff,wcnt,crtn)

where:

buff	is the array containing the data to be transferred.

wcnt — is the integer number of data words to be transferred.

crtn — is the name of an assembly language routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISDATC call.

Errors:

i = 0 Normal return.
= 1 No other job currently exists in the system.

Example:

```
INTEGER*2 MSG(40)
EXTERNAL RTN
.
.
.
CALL ISDATC(MSG,40,RTN)
```

ISDATF

The ISDATF function transfers a specified number of words from one job to the other. Control returns to the user program immediately after the transfer is queued and execution continues. When the other job accepts the message through a receive data request, the specified FORTRAN subprogram (crtn) is activated as an asynchronous completion routine (see Section 4.2.1).

Form: i = ISDATF (buff,wcnt,area,crtn)

where:

buff — is the array containing the data to be transferred.

wcnt — is the integer number of data words to be transferred.

area — is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.

crtn — is the name of a FORTRAN routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISDATF call.

Errors:

i = 0 Normal return.
= 1 No other job currently exists in the system.

Example:

```
INTEGER*2 MSG(40),SPOT(4)
EXTERNAL RTN
.
.
.
CALL ISDATF(MSG,40,SPOT,RTN)
```

ISDATW

The ISDATW function transfers a specified number of words from one job to the other. Control returns to the user program when the other job has accepted the data through a receive data request.

Form: i = ISDATW (buff,wcnt)

where:

buff	is the array containing the data to be transferred.
wcnt	is the integer number of data words to be transferred.

Errors:

i = 0	Normal return.
= 1	No other job currently exists in the system.

Example:

```
INTEGER*2 MSG(40)
.
.
.
IF (ISDATW(MSG,40).NE.0) STOP 'FOREGROUND JOB NOT RUNNING'
```

ISLEEP

4.3.46 ISLEEP

The ISLEEP function suspends the main program execution of a job for a specified amount of time. The specified time is the sum of hours, minutes, seconds, and ticks specified in the ISLEEP call. All completion routines continue to execute. Support for ISLEEP in SJ also requires timer support.

Form: i = ISLEEP (hrs,min,sec,tick)

where:

hrs	is the integer number of hours.
min	is the integer number of minutes.
sec	is the integer number of seconds.
tick	is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks).

Notes:

1. ISLEEP requires a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

2. If the system is busy, the time that execution is suspended may be greater than that specified.

Errors:

i = 0	Normal return.
= 1	No queue element available.

Example:

```
.
.
.
CALL IQSET(2)
.
.
.
CALL ISLEEP(0,0,0,4)        !GIVE BACKGROUND JOB SOME TIME
```

ISPFN/ISPFNC/ISPFNF/ISPFNW

4.3.47 ISPFN/ISPFNC/ISPFNF/ISPFNW

These functions are used in conjunction with special functions to various handlers. They provide a means of doing device-dependent functions, such as rewind and backspace, to those devices. If ISPFN function calls are made to any other devices, the function call is ignored. For more information on programming for specific devices, see Section 1.4.7.

To use these functions, the handler must be in memory and a channel associated with a file via a non-file-structured LOOKUP call. These functions require a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

ISPFN

The ISPFN function queues the specified operation and immediately returns control to the user program. The IWAIT function can be used to ensure completion of the operation.

Form: i = ISPFN (code,chan[,wcnt,buff,blk])

where:

code — is the integer numeric code of the function to be performed (see Table 4-2).

chan — is the integer specification for the RT-11 channel to be used for the operation.

wcnt — is the integer number of data words in the operation.* Default value is 0. In magtape operations, it specifies the number of records to space forward or backward. For a backspace operation (wcnt=0), the tape drive backspaces to a tape mark or to the beginning-of-tape. For a forward space operation (wcnt=0), the tape drive forward spaces to a tape mark or the end-of-tape.

buff — is the array to be used as the data buffer.* Default value is 0.

blk — is the integer block number of the file to be operated upon.* Default value is 0.

---

* These parameters are optional with some ISPFUN calls, depending on the particular function.

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to zero.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
      .
ERRADR = IADDR (ERRBLK) !GET THE ADDRESS OF  THE  4-WORD ERROR BLOCK
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

Table 4-2
Special Function Codes (Octal)

Function	MT,MM	CT	DX	DM	DY	DL
Read absolute			377	377	377	377
Write absolute			376	376	376	376
Write absolute with deleted data			375		375	
Forward to last file		377				
Forward to last block		376				
Forward to next file		375				
Forward to next block		374				
Rewind to load point	373	373				
Write file gap		372				
Write end-of-file	377					
Forward 1 record	376					
Backspace 1 record	375					
Initialize the bad block replacement table				374		374
Write with extended record gap	374					
Offline	372					
Return volume size				373	373	373

Errors:

i = 0	Normal return.
= 1	Attempt to read or write past end-of-file.
= 2	Hardware error occurred on channel.
= 3	Channel specified is not open.

Example:

```
CALL ISPFN("373,ICHAN)      !REWIND
```

ISPFNC

The ISPFNC function queues the specified operation and immediately returns control to the user program. When the operation is complete, the specified assembly language routine (crtn) is entered as an asynchronous completion routine.

Form: i = ISPFNC (code,chan,wcnt,buff,blk,crtn)

where:

code — is the integer numeric code of the function to be performed (see Table 4-2).

chan — is the integer specification for the RT-11 channel to be used for the operation.

wcnt — is the integer number of data words in the operation. This argument must be 0 if not required.

buff — is the array to be used as the data buffer. This argument must be 0 if not required.

blk — is the integer block number of the file to be operated upon. This argument must be 0 if not required.

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
      .
ERRADR = IADDR (ERRBLK) !GET THE ADDRESS OF  THE  4-WORD ERROR BLOCK
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

crtn — is the name of an assembly language routine to be activated on completion of the operation. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISPFNC call.

Errors:

i = 0	Normal return.
= 1	Attempt to read or write past end-of-file.
= 2	Hardware error occurred on channel.
= 3	Channel specified is not open.

ISPFNF

The ISPFNF function queues the specified operation and immediately returns control to the user program. When the operation is complete, the specified FORTRAN subprogram (crtn) is entered as an asynchronous completion routine.

Form: i = ISPFNF (code,chan,wcnt,buff,blk,area,crtn)

where:

code — is the integer numeric code of the function to be performed (see Table 4-2).

chan — is the integer specification for the RT-11 channel to be used for the operation.

wcnt — is the integer number of data words in the operation. This argument must be 0 if not required.

buff — is the array to be used as the data buffer. This argument must be 0 if not required.

blk — is the integer block number of the file to be operated upon. This argument must be 0 if not required.

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2      ERRADR, ERRBLK(4)
DATA ERRBLK    /0,0,0,0,/
      .
      .
      .
      .
ERRADR = IADDR (ERRBLK) !GET THE ADDRESS OF  THE  4-WORD ERROR BLOCK
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

area — is a four-word area to be set aside for linkage information; this area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.

crtn — is the name of a FORTRAN routine to be activated on completion of the operation. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISPFNF call. The subroutine has two arguments:

SUBROUTINE crtn (iargl,iarg2)

iarg1	is the channel status word (see Section 2.4) for the operation just completed. If bit 0 is set, a hardware error occurred during the transfer.
iarg2	is the channel number used for the operation just completed.

Errors:

i = 0	Normal return.
= 1	Attempt to read or write past end-of-file.
= 2	Hardware error occurred on channel.
= 3	Channel specified is not open.

Example:

```
      REAL*4 MTNAME(2),AREA(2)
      DATA MTNAME/3RMTO,0./
      EXTERNAL DONSUB
      .
      .
      .
      I=IGETC()                  !ALLOCATE CHANNEL
      CALL IFETCH(MTNAME)        !FETCH MT HANDLER
      CALL LOOKUP(I,MTNAME)      !NON-FILE-STRUCTURED LOOKUP ON MTO
      IERR=ISPFNF("373,I,0,0,0,AREA,DONSUB)    !REWIND MAGTAPE
      .
      .
      .
      END
      SUBROUTINE DONSUB
C
C     RUNS WHEN MTO HAS BEEN REWOUND
C
      .
      .
      .
      END
```

ISPFNW

The ISPFNW function queues the specified operation and returns control to the user program when the operation is complete.

Form: i = ISPFNW (code,chan[,wcnt,buff,blk])

where:		
	code	is the integer numeric code of the function to be performed (see Table 4-2).
	chan	is the integer specification for the RT-11 channel to be used for the operation.
	wcnt	is the integer number of data words in the operation.*
	buff	is the array to be used as the data buffer.*
	blk	is the integer block number of the file to be operated upon.*

* These parameters are optional with some ISPFUN calls, depending on the particular function.

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2      ERRADR, ERRBLK(4)
DATA ERRBLK    /0,0,0,0,/
      .
      .
      .
      .
ERRADR = IADDR (ERRBLK) !GET THE ADDRESS OF  THE  4-WORD ERROR BLOCK
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

Errors:

```
i = 0   Normal return.
  = 1   Attempt to read or write past end-of-file.
  = 2   Hardware error occurred on channel.
  = 3   Channel specified is not open.
```

Example:

```
        INTEGER*2 BUF(65),TRACK,SECTOR,DBLK(4)
        DATA DBLK/3RDX0,0,0,0/
        .
        .
        .
        ICHAN=IGETC()
        IF(ICHAN.LT.0) STOP 'NO CHANNEL AVAILABLE'
        IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
        .
        .
        .
C       READ AN ABSOLUTE TRACK AND SECTOR FROM THE FLOPPY
C
        ICODE=ISPFNW('377,ICHAN,TRACK,BUF,SECTOR)
C
C       BUF(1) IS THE DELETED DATA FLAG
C       BUF(2-65) IS THE DATA
```

ISPY

4.3.48 ISPY

The ISPY function returns the integer value of the word at a specified offset from the RT-11 resident monitor. This subroutine uses the .GVAL programmed request to return fixed monitor offsets. (See Section 2.2.6 for information on fixed offset references.)

Form: i = ISPY (ioff)

where: ioff is the offset (from the base of RMON) to be examined.

Function Result:

The function result (i) is set to the value of the word examined.

Example:

```
C
C     BRANCH TO 200 IF RUNNING UNDER FB MONITOR
C
      IF(ISPY("300).AND.1) GOTO 200
C
C     WORD AT OCTAL 300 FROM RMON IS
C     THE CONFIGURATION WORD.
```

ITIMER

4.3.49 ITIMER

The ITIMER function schedules a specified FORTRAN subroutine to be run as an asynchronous completion routine after a specified time interval has elapsed. This request is supported by SJ when the timer support option is included during system generation.

Form: i = ITIMER (hrs,min,sec,tick,area,id,crtn)

where:

- hrs — is the integer number of hours.
- min — is the integer number of minutes
- sec — is the integer number of seconds.
- tick — is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks).
- area — is a four-word area that must be provided for link information; this area must never be modified by the FORTRAN program, and the USR must never swap over it. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.
- id — is the identification integer to be passed to the routine being scheduled.
- crtn — is the name of the FORTRAN subroutine to be entered when the specified time interval elapses. This name must be specified in an EXTERNAL statement in the FORTRAN routine that references ITIMER. The subroutine has one argument. For example:

  ```
  SUBROUTINE crtn(id)
  INTEGER id
  ```

  When the routine is entered, the value of the integer argument is the value specified for id in the appropriate ITIMER call.

Notes:

1. This function can be cancelled at a later time by an ICMKT function call.

2. If the system is busy, the actual time interval at which the completion routine is run can be greater than the time interval requested.

3. FORTRAN subroutines can periodically reschedule themselves by issuing ISCHED or ITIMER calls.

4. ITIMER requires a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

For more information on scheduling completion routines, see Section 4.2.1 and the assembly language .MRKT request, Section 2.4.

Errors:

i	= 0	Normal return
	= 1	No queue elements available; unable to schedule request.

Example:

```
      INTEGER*2 AREA(4)
      EXTERNAL WATCHD
      .
      .
      .
C     IF THE CODE FOLLOWING ITIMER DOES NOT REACH THE ICMKT CALL
C     IN 12 MINUTES, WATCH DOG COMPLETION ROUTINE WILL BE
C     ENTERED WITH ID OF 3
C
      CALL ITIMER(0,12,0,0,AREA,3,WATCHD)
      .
      .
      .
      CALL ICMKT(3,AREA)
      .
      .
      .
      END
      SUBROUTINE WATCHD(ID)
C
C     THIS IS CALLED AFTER 12 MINUTES
      .
      .
      .
      RETURN
      END
```

ITLOCK

4.3.50 ITLOCK (FB and XM Only)

The ITLOCK function is used in an FB or XM system to attempt to gain ownership of the USR. It is similiar to LOCK (Section 4.3.69) in that if successful, the user job returns with the USR in memory. However, if a job attempts to LOCK the USR while the other job is using it, the requesting job is suspended until the USR is free. With ITLOCK, if the USR is not available, control returns immediately and the lock failure is indicated. ITLOCK cannot be called from a completion or interrupt routine.

Form: i = ITLOCK()

For further information on gaining ownership of the USR, see the assembly language .TLOCK request, Section 2.4.

Errors:

i	= 0	Normal return.
	= 1	USR is already in use by another job.

Example:

```
IF(ITLOCK().NE.0) GOTO 100      !GOTO 100 IF USR BUSY
```

ITTINR

4.3.51 ITTINR

The ITTINR function transfers a character from the console terminal to the user program. If no characters are available, return is made with an error flag set.

Form: i = ITTINR()

If the function result (i) is less than 0 when execution of the ITTINR function is complete, it indicates that no character was available; the user has not yet typed a valid line. Under the FB or XM monitor, ITTINR does not return a result of less than zero unless bit 6 of the job status word was on when the request was issued.

There are two modes of doing console terminal input. The mode is governed by bit 12 of the job status word (JSW). The JSW is at octal location 44. If bit 12 equals 0, normal I/O is performed. In this mode, the following conditions apply:

1. The monitor echoes all characters typed.

2. CTRL/U and RUBOUT perform line deletion and character deletion, respectively.

3. A carriage return, line feed, CTRL/Z, or CTRL/C must be struck before characters on the current line are available to the program. When one of these is typed, characters on the line typed are passed one by one to the user program. Both carriage return and line feed are passed to the program.

If bit 12 equals 1, the console is in special mode. The effects are:

1. The monitor does not echo characters typed except for CTRL/C and CTRL/O.

2. CTRL/U and RUBOUT do not perform special functions.

3. Characters are immediately available to the program.

4. No ALTMODE conversion is done.

In special mode, the user program must echo the characters desired. However, CTRL/C and CTRL/O are acted on by the monitor in the usual way. Bits 12 and 14 in the JSW must be set by the user program if special console mode or lower case characters are desired (see the example under Section 4.3.35). These bits are cleared when control returns to RT-11.

NOTE

To set and/or clear bits in the JSW, do an IPEEK and then an IPOKE. In special terminal mode (JSW bit 12 set), normal FORTRAN formatted I/O from the console is undefined.

In the FB or XM monitor, CTRL/F and CTRL/B are not affected by the setting of bit 12. The monitor always acts on these characters if the SET TT FB command is in effect.

Under the FB or XM monitor, if a terminal input request is made and no character is available, job execution is suspended until a character is ready. If a program really requires execution to continue and ITTINR to return a result of less than zero, it must turn on bit 6 of the JSW before the ITTINR. Bit 6 is cleared when a program terminates.

NOTE

If a foreground job has characters in the TT output buffer, they are not output under the following conditions:

(1) If a background job is doing output to the console TT, the foreground job cannot output characters from its buffer until the background job outputs a line feed character. This can be troublesome if the console device is a graphics terminal, and the background job is doing graphic output without sending any line feeds.

(2) If no background job is running (that is, KMON is in control of background), the foreground job cannot output its characters until the user types a carriage return or a line feed. In the former case, KMON gets control again and locks out foreground output as soon as the foreground output buffer is empty.

Function Results:

i	>0	Normal return; character read.
	<0	Error return; no character available.

Example:

```
ICHAR=ITTINR()            !READ A CHARACTER FROM THE CONSOLE
IF(ICHAR.LT.0) GOTO 100   !CHARACTER NOT AVAILABLE
```

ITTOUR

4.3.52 ITTOUR

The ITTOUR function transfers a character from the user program to the console terminal if there is room for the character in the monitor buffer. If it is not currently possible to output a character, an error flag is returned.

Form: i = ITTOUR (char)

where: char is the character to be output, right-justified in the integer (can be LOGICAL*1 entity if desired).

If the function result (i) is 1 when execution of the ITTOUR function is complete, it indicates that there is no room in the buffer and that no character was output. Under the FB or XM monitor, ITTOUR normally does not return a result of 1. Instead, the job is blocked until room is available in the output buffer. If a job really requires execution to continue and a result of 1 to be returned, it must turn on bit 6 of the JSW (location 44) before issuing the request.

The ITTINR and ITTOUR have been supplied as a help to those users who do not wish to suspend program execution until a console operation is complete. With these modes of I/O, if a no-character or no-room condition occurs, the user program can continue processing and try the operation again at a later time.

Errors:

i = 0	Normal return;	character was output.
= 1	Error return;	ring buffer is full.

Example:

```
      DO 20 I=1,5
10    IF(ITTOUR("007).NE.0) GOTO 10       !RING BELL 5 TIMES
20    CONTINUE
```

ITWAIT

4.3.53 ITWAIT (FB and XM Only)

The ITWAIT function suspends the main program execution of the current job for a specified time interval. All completion routines continue to execute.

Form: i = ITWAIT (itime)

where: itime is the two-word internal format time interval.

itime (1) is the high-order time.
itime (2) is the low-order time.

Notes:

1. ITWAIT requires a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

2. If the system is busy, the actual time interval for which execution is suspended can be greater than the time interval specified.

Errors:

i = 0	Normal return.
= 1	No queue element available.

Example:

```
INTEGER*2 TIME(2)
.
.
.
CALL ITWAIT(TIME)      !WAIT FOR TIME TIME
```

IUNTIL

4.3.54 IUNTIL (FB and XM Only)

The IUNTIL function suspends main program execution of the job until the time of day specified. All completion routines continue to run.

Form: i = IUNTIL (hrs,min,sec,tick)

where:

hrs	is the integer number of hours.
min	is the integer number of minutes.
sec	is the integer number of seconds.
tick	is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks).

Notes:

1. IUNTIL requires a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

2. If the system is busy, the actual time of day that the program resumes execution can be later than that requested.

Errors:

i = 0	Normal return.
= 1	No queue element available.

Example:

```
C     TAKE A LUNCH BREAK
      CALL IUNTIL(13,0,0,0)      !START UP AGAIN AT 1 P.M.
```

IWAIT

4.3.55 IWAIT

The IWAIT function suspends execution of the main program until all input/output operations on the specified channel are complete. This function is used with IREAD, IWRITE, and ISPFN calls. Completion routines continue to execute.

Form: i = IWAIT (chan)

where: chan — is the integer specification for the RT-11 channel to be used.

For further information on suspending execution of the main program, see the assembly language .WAIT request, Section 2.4.

Errors:

i = 0	Normal return.
= 1	Channel specified is not open.
= 2	Hardware error occurred during the previous I/O operation on this channel.

Example:

```
IF(IWAIT(ICHAN).NE.0) CALL IOERR(4)
```

IWRITC/IWRITE/IWRITF/IWRITW

4.3.56 IWRITC/IWRITE/IWRITF/IWRITW

These functions transfer a specified number of words from memory to the specified channel. The IWRITE functions require queue elements; this should be considered when the IQSET function (Section 4.3.37) is executed.

IWRITC

The IWRITC function transfers a specified number of words from memory to the specified channel. The request is queued and control returns to the user program. When the transfer is complete, the specified assembly language routine (crtn) is entered as an asynchronous completion routine.

Form: i = IWRITC (wcnt,buff,blk,chan,crtn)

where: wcnt — is the relative integer number of words to be transferred.

buff — is the array to be used as the output buffer.

blk — is the relative integer block number of the file to be written. The user program normally updates blk before it is used again.

chan — is the relative integer specification for the RT-11 channel to be used.

crtn — is the name of the assembly language routine to be activated upon completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IWRITC call.

NOTE

The blk argument must be updated, if necessary, by the user program. For example, if the program is writing two blocks at a time, blk should be updated by 2.

Errors:

i = n — Normal return; n equals the number of words written, rounded to a multiple of 256 (0 for non-file-structured writes).

NOTE

If the word count returned is less than that requested, an implied end-of-file has occurred although the normal return is indicated.

= -1 — Attempt to write past end-of-file; no more space is available in the file.
= -2 — Hardware error occurred.
= -3 — Channel specified is not open.

Example:

```
INTEGER*2 IBUF(256)
EXTERNAL CRTN
.
.
.
ICODE=IWRITC(256,IBUF,IBLK,ICHAN,CRTN)
```

IWRITE

The IWRITE function transfers a specified number of words from memory to the specified channel. Control returns to the user program immediately after the request is queued. No special action is taken upon completion of the operation.

Form: i = IWRITE (wcnt,buff,blk,chan)

where: wcnt — is the integer number of words to be transferred.

buff — is the array to be used as the output buffer.

blk — is the integer block number of the file to be written. The user program normally updates blk before it is used again.

chan — is the integer specification for the RT-11 channel to be used.

Errors:

See the errors under IWRITC.

Example:

See the example under IREAD, Section 4.3.40.

IWRITF

The IWRITF function transfers a number of words from memory to the specified channel. The transfer request is queued an: control returns to the user program. When the operation is complete, the specified FORTRAN subprogram (crtn) is entered as an asynchronous completion routine (see Section 4.2.1).

Form: i = IWRITF (wcnt,buff,blk,chan,area,crtn)

where:

wcnt — is the integer number of words to be transferred.

buff — is the array to be used as the output buffer.

blk — is the integer block number of the file to be written. The user program normally updates blk before it is used again.

chan — is the integer specification for the RT-11 channel to be used.

area — is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when crtn has been activated.

crtn — is the name of the FORTRAN routine to be activated upon completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IWRITF call. The subroutine has two arguments:

SUBROUTINE crtn (iarg1,iarg2)

iarg1 — is the channel status word (see Section 2.4) for the operation just completed. If bit 0 is set, a hardware error occurred during the transfer.

iarg2 — is the channel number used for the operation just completed.

Errors:

See the errors under IWRITC.

Example:

See the example under IREADF, Section 4.3.40.

IWRITW

The IWRITW function transfers a specified number of words from memory to the specified channel. Control returns to the user program when the transfer is complete.

Form: i = IWRITW (wcnt,buff,blk,chan)

where:

wcnt	is the integer number of words to be transferred.
buff	is the array to be used as the output buffer.
blk	is the integer block number of the file to be written. The user program normally updates blk before it is used again.
chan	is the integer specification for the RT-11 channel to be used.

Errors:

See the errors under IWRITC.

Example:

See the example under IREADW, Section 4.3.40.

JADD

4.3.57 JADD

The JADD function computes the sum of two INTEGER*4 values.

Form: i = JADD (jopr1,jopr2,jres)

where:

jopr1	is an INTEGER*4 variable.
jopr2	is an INTEGER*4 variable.
jres	is an INTEGER*4 variable that receives the sum of jopr1 and jopr2.

Function Results:

i = -2	An overflow occurred while computing the result.
= -1	Normal return; the result is negative.
= 0	Normal return; the result is zero.
= 1	Normal return; the result is positive.

Example:

```
        INTEGER*4 JOP1,JOP2,JRES
        .
        .
        .
        IF(JADD(JOP1,JOP2,JRES).EQ.-2) GOTO 100
```

JAFIX

4.3.58 **JAFIX**

The JAFIX function converts a REAL*4 value to INTEGER*4.

Form: i = JAFIX (asrc,jres)

where:	asrc	is a REAL*4 variable, constant, or expression to be converted to INTEGER*4.
	jres	is an INTEGER*4 variable that is to contain the result of the conversion.

Function Results:

i = -2	An overflow occurred while computing the result.
= -1	Normal return; the result is negative.
= 0	Normal return; the result is zero.
= 1	Normal return; the result is positive.

Example:

```
      INTEGER*4 JOP1
C     READ A LARGE INTEGER FROM THE TERMINAL
      ACCEPT 99,A
99    FORMAT (F15.0)
      IF(JAFIX(A,JOP1).EQ.-2) GOTO 100
      .
      .
      .
```

JCMP

4.3.59 **JCMP**

The JCMP function compares two INTEGER*4 values and returns an INTEGER*2 value that reflects the signed comparison result.

Form: i = JCMP (jopr1,jopr2)

where:	jopr1	is the INTEGER*4 variable or array element that is the first operand in the comparison.
	jopr2	is the INTEGER*4 variable or array element that is the second operand in the comparison.

Function Result:

```
i = -1    If jopr1 is less than jopr2
  = 0     If jopr1 is equal to jopr2
  = 1     If jopr1 is greater than jopr2
```

Example:

```
        INTEGER*4 JOPX,JOPY
        .
        .
        .
        IF(JCMP(JOPX,JOPY)) 10,20,30
```

JDFIX

4.3.60 JDFIX

The JDFIX function converts a REAL*8 (DOUBLE PRECISION) value to INTEGER*4.

Form: i = JDFIX (dsrc,jres)

where:

dsrc	is a REAL*8 variable, constant, or expression to be converted to INTEGER*4.
jres	is an INTEGER*4 variable to contain the conversion result.

Function Results:

```
i = -2        An overflow occurred while computing the result.
  = -1        Normal return;  the result is negative.
  = 0         Normal return;  the result is zero.
  = 1         Normal return;  the result is positive.
```

Example:

```
        INTEGER*4 JNUM
        REAL*8 DPNUM
        .
        .
        .
  20    TYPE 98
  98    FORMAT(' ENTER POSITIVE INTEGER')
        ACCEPT 99,DPNUM
  99    FORMAT(F20.0)
        IF(JDFIX(DPNUM,JNUM).LT.0) GOTO 20
        .
        .
        .
```

JDIV

4.3.61 JDIV

The JDIV function computes the quotient of two INTEGER*4 values.

Form: i = JDIV (jopr1,jopr2,jres[,jrem]) where: jopr1 is an INTEGER*4 variable that is the dividend of the operation.

jopr2 is an INTEGER*4 variable that is divisor of jopr1.

jres is an INTEGER*4 variable that receives the quotient of the operation (that is, jres=jopr1/jopr2).

jrem is an INTEGER*4 variable that receives the remainder of the operation. The sign is the same as that for jopr1.

Function Results:

i = -3	An attempt was made to divide by 0.
= -2	(not used)
= -1	Normal return; the quotient is negative.
= 0	Normal return; the quotient is 0.
= 1	Normal return; the quotient is positive.

Example:

```
INTEGER*4 JN1,JN2,JQUO
.
.
.
CALL JDIV(JN1,JN2,JQUO)
.
.
.
```

JICVT

4.3.62 JICVT

The JICVT function converts a specified INTEGER*2 value to INTEGER*4.

Form: i = JICVT (isrc,jres)

Where: isrc is the INTEGER*2 quantity to be converted.

jres is the INTEGER*4 variable or array element to receive the result.

Function Results:

i = -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

```
INTEGER*4 JVAL
CALL JICVT(478,JVAL)      !FORM A 32-BIT CONSTANT
```

JJCVT

4.3.63 JJCVT

The JJCVT function interchanges words of an INTEGER*4 value to form an internal format time or vice versa. This procedure is necessary when the INTEGER*4 variable is to be used as an argument in a timer-support function such as ITWAIT. When a two-word internal format time is specified to a function such as ITWAIT, it must have the high-order time as the first word and the low-order time as the second word.

Form: CALL JJCVT (jsrc)

where: jsrc — is the INTEGER*4 variable whose contents are to be interchanged.

Errors:

None.

Example:

```
INTEGER*4 TIME
.
.
.
CALL GTIM(TIME)      !GET TIME OF DAY
CALL JJCVT(TIME)     !TURN IT INTO INTEGER*4 FORMAT
```

JMOV

4.3.64 JMOV

The JMOV function assigns the value of an INTEGER*4 variable to another INTEGER*4 variable and returns the sign of the value moved.

Form: i = JMOV (jsrc,jdest)

where: jsrc — is the INTEGER*4 variable whose contents are to be moved.

jdest — is the INTEGER*4 variable that is the target of the assignment.

Function Result:

The value of the function is an INTEGER*2 value that represents the sign of the result as follows:

i = -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

The JMOV function allows an INTEGER*4 quantity to be compared with 0 by using it in a logical IF statement. For example:

```
INTEGER*4 INT1
.
.
.
IF(JMOV(INT1,INT1)) 300,100,300 !GO TO STMT 300 IF INT1 IS NOT 0
```

JMUL

4.3.65 JMUL

The JMUL function computes the product of two INTEGER*4 values.

Form: i = JMUL (jopr1,jopr2,jres)

where	jopr1	is an INTEGER*4 variable that is the multiplicand.
	jopr2	is an INTEGER*4 variable that is the multiplier.
	jres	is an INTEGER*4 variable that receives the product of the operation.

Function Results:

i = -2	An overflow occurred while computing the result.
= -1	Normal return; the product is negative.
= 0	Normal return; the product is 0.
= 1	Normal return; the product is positive.

Example:

```
      INTEGER*4 J1,J2,JRES
      .
      .
      .
      IF(JMUL(J1,J2,JRES)+1) 100,10,20
C     GOTO 100 IF OVERFLOW
C     GOTO 10 IF RESULT IS NEGATIVE
C     GOTO 20 IF RESULT IS POSITIVE OR ZERO
```

JSUB

4.3.66 JSUB

The JSUB function computes the difference between two INTEGER*4 values.

Form: i = JSUB (jopr1,jopr2,jres)

where:	jopr1	is an INTEGER*4 variable that is the minuend of the operation.

jopr2	is an INTEGER*4 variable that is the subtrahend of the operation.
jres	is an INTEGER*4 variable that is to receive the difference between iopr1 and iopr2 (that is, jres=jopr1-jopr2).

Function Results:

i = -2	An overflow occurred while computing the result.
= -1	Normal return; the result is negative.
= 0	Normal return; the result is 0.
= 1	Normal return; the result is positive.

Example:

```
INTEGER*4 JOP1,JOP2,J3
.
.
.
CALL JSUB(JOP1,JOP2,J3)
```

JTIME

4.3.67 JTIME

The JTIME subroutine converts the time specified to the internal two-word format time.

Form: CALL JTIME (hrs,min,sec,tick,time)

where:	hrs	is the integer number of hours.
	min	is the integer number of minutes.
	sec	is the integer number of seconds.
	tick	is the integer number of ticks (1/60 of a second for 60-cycle clocks; 1/50 of a second for 50-cycle clocks).
	time	is the two-word area to receive the internal format time: time(1) is the high-order time. time(2) is the low-order time.

Errors:

None.

Example:

```
      INTEGER*4 J1
      .
      .
      .
C     CONVERT 3 HRS, 7 MIN, 23 SECONDS TO INTEGER *4 VALUE
      CALL JTIME(3,7,23,0,J1)
      CALL JJCVT(J1)
```

LEN

4.3.68 LEN

The LEN function returns the number of characters currently in the string contained in a specified array. This number is computed as the number of characters preceeding the first null byte encountered. If the specified array contains a null string, a value of 0 is returned.

Form: i = LEN (a)

where: a specifies the array containing the string.

Errors:

None.

Example:

```
        LOGICAL*1 STRNG(73)
        .
        .
        .
        TYPE 99,(STRNG(I),I=1,LEN(STRNG))
99      FORMAT('0',132A1)
```

LOCK

4.3.69 LOCK

The LOCK subroutine is issued to keep the USR in memory for a series of operations. The USR (User Service Routine) is the section of the RT-11 system that performs various file management functions.

If all the conditions that cause swapping are satisfied, a portion of the user program is written out to the disk file SWAP.SYS and the USR is loaded. Otherwise, the USR in memory is used, and no swapping occurs. The USR is not released until an UNLOCK (see Section 4.3.102) is given. (Note that in an FB system, calling the CSI can also perform an implicit UNLOCK.) A program that has many USR requests to make can LOCK the USR in memory, make all the requests, and then UNLOCK the USR; no time is spent doing unnecessary swapping.

In an FB or XM environment, a LOCK inhibits the other job from using the USR. Thus, the USR should be locked only for as long as necessary.

NOTE

> Foreground jobs perform a LOCK when they require the USR. This can cause the USR to be unavailable for other jobs for a considerable period of time. The USR is not reentrant and it cannot be shared by other jobs. Only one job has use of the USR at a time and other jobs requiring it must queue up for it. This fact should be considered for systems requiring concurrent foreground and background jobs. This is particularly true when magtape and/or cassette are active.
>
> The USR does file operations, and these operations require a sequential search of the tape for magtape and cassette. This could lock out the foreground job for a long time while the background job does a tape operation. The programmer should keep this in mind when designing such systems. The FB and XM monitors supply the ITLOCK routine, which permits the foreground job to check for the availability of the USR.

Form: CALL LOCK

Note that the LOCK routine reduces time spent in file handling by eliminating the swapping of the USR. If the USR is currently resident, LOCK involves no I/O. (The USR is always resident in XM.) After a LOCK has been executed, the UNLOCK routine must be executed to release the USR from memory. The LOCK/UNLOCK routines are complementary and must be matched. That is, if three LOCKs are issued, at least three UNLOCKs must be done, otherwise the USR is not released. More UNLOCKs than LOCKs can occur without error; the extra UNLOCKs are ignored.

Notes:

1. It is vital that the LOCK call not come from within the area into which the USR will be swapped. If this should occur, the return from the USR request would not be to the user program, but to the USR itself, since the LOCK function causes part of the user program to be saved on disk and replaced in memory by the USR. Furthermore, subroutines, variables, and arrays in the area where the USR is swapping should not be referenced while the USR is LOCKed in memory.

2. Once a LOCK has been performed, it is not advisable for the program to destroy the area the USR is in, even though no further use of the USR is required. This causes unpredictable results when an UNLOCK is done.

3. LOCK cannot be called from a completion or interrupt routine.

4. If a SET USR NOSWAP command has been issued, LOCK and UNLOCK do not cause the USR to swap. However, in FB, LOCK still inhibits the other job from using the USR and UNLOCK allows the other job access to the USR.

5. The USR cannot accept argument lists, such as device file name specifications, located in the area into which it has been locked.

Errors:

None.

LOOKUP

4.3.70 LOOKUP

The LOOKUP function associates a specified channel with a device and/or file for the purpose of performing I/O operations. The channel used is then busy until one of the following functions is executed.

CLOSEC
ISAVES
PURGE

Form: i = LOOKUP (chan,dblk[,count])

where:

	chan	is the integer specification for the RT-11 channel to be associated with the file.
	dblk	is the four-word area specifying the Radix-50 file descriptor. Note that unpredictable results occur if the USR swaps over this four-word area.
	count	is an optional argument used for the cassette handler. This argument defaults to 0.

NOTE

The arguments of LOOKUP must be positioned so that the USR does not swap over them.

The handler for the selected device must be in memory for a LOOKUP. If the first word of the file name in dblk is 0 and the device is a file-structured device, absolute block 0 of the device is designated as the beginning of the file. This technique, called a non-file-structured LOOKUP, allows I/O to any physical block on the device. If a file name is specified for a device that is not file-structured (such as LP:FILE.TYP), the name is ignored.

NOTE

Since non-file-structured LOOKUPs allow I/O to any physical block on the device, the user must be aware that, in this mode, it is possible to overwrite the RT-11 device directory, thus destroying all file information on the device.

Errors:

i = n	Normal return; n equals the number of blocks in the file (0 for non-file-structured lookups on a cassette and magtape).
= -1	Channel specified is already open.
= -2	File specified was not found on the device.
-3	Device in use
-4	Tape drive is not available

Example:

```
INTEGER*2 DBLK(4)
DATA DBLK/3RDK0,3RFTN,3R44 ,3RDAT/
.
.
.
ICHAN=IGETC()
IF(ICHAN.LT.0) STOP 'NO CHANNEL'
IF(IFETCH(DBLK).NE.0) STOP 'BAD FETCH'
IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
.
.
.
CALL CLOSEC(ICHAN)
CALL IFREEC(ICHAN)
.
.
.
```

MRKT

4.3.71 MRKT

The MRKT function schedules an assembly language completion routine to be entered after a specified time interval has elapsed. Support for MRKT in SJ requires timer support.

Form: i = MRKT (id,crtn,time)

where:

id — is an integer identification number to be passed to the routine being scheduled.

crtn — is the name of the assembly language routine to be entered when the time interval elapses. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the MRKT call.

time — is the two-word internal format time interval; when this interval elapses, the routine is entered. If considered as a two-element INTEGER*2 array:

time (1) is the high-order time.

time (2) is the low-order time.

Notes:

1. MRKT requires a queue element; this should be considered when the IQSET function (Section 4.3.33) is executed.

2. If the system is busy, the time interval that elapses before the completion routine is run can be greater than that requested.

For further information on scheduling completion routines, see the assembly language .MRKT request, Section 2.4.22.

Errors:

i	= 0	Normal return
	= 1	No queue element was available; unable to schedule request.

Example:

```
INTEGER*2 TINT(2)
EXTERNAL ARTN
.
.
.
CALL MRKT(4,ARTN,TINT)
```

MTATCH

4.3.72 MTATCH (FB and XM Only)

The MTATCH subroutine attaches a terminal for exclusive use by the requesting job. This operation must be performed before any job can use a terminal with multi-terminal programmed requests.

Form: i = MTATCH (unit[,addr])

where:

addr	is the optional address of an asynchronous terminal status word. Omit this argument if the asynchronous terminal status word is not required. The asynchronous terminal status word is a SYSGEN option.
unit	is the logical unit number of the terminal (lun).

Errors:

i	= 0	Normal return
	= 3	Non-existent unit number
	= 5	Unit attached by another job
	= 6	In XM monitor, the optional status word address is not in a valid user virtual address space.

MTDTCH

4.3.73 MTDTCH (FB and XM Only)

The MTDTCH subroutine is the complement of the MTATCH subroutine. Its function is to detach a terminal from a particular job, and make it available for other jobs.

Form: i = MTDTCH(unit)

where: unit is the logical unit number of the terminal to be detached (lun).

Errors:

i = 0	Normal return
= 2	Illegal unit number. Terminal is not attached.
= 3	Non-existent unit number.

MTGET

4.3.74 MTGET (FB and XM Only)

The MTGET subroutine furnishes the user with information about a specific terminal in a multi-terminal system.

Form: i = MTGET (unit,addr)

where: addr is a four-word area to receive the status information. The area is a four-element INTEGER*2 array. See Section 2.4.36 for area format.

unit is the unit number of the line and terminal whose status is desired.

Errors:

i = 0	Normal return
= 2	Unit not attached
= 3	Non-existent unit number
= 6	In XM monitor, the terminal status buffer address is outside legal program limits.

MTIN

4.3.75 MTIN (FB and XM Only)

The MTIN subroutine transfers characters from a specified terminal to the user program. This subroutine is a multi-terminal form of ITTINR. If no characters are available, a flag is set to indicate an error upon return from the subroutine. If no character count argument is specified, one character is transferred.

Form: i = MTIN (unit,char[,chrcnt])

where:

unit	is the unit number of the terminal.
char	is the variable to contain the characters read in from the terminal indicated by the unit number.
chrcnt	is an optional argument that indicates the number of characters to be read.

Errors:

i = 0	Normal return
= 1	No input available
= 2	Unit not attached
= 3	Non-existent unit number

MTOUT

4.3.76 MTOUT (FB and XM Only)

The MTOUT subroutine transfers characters to a specified terminal. This subroutine is a multi-terminal form of ITTOU. If no room is available in the output ring buffer, a flag is set to indicate an error upon return from the subroutine. If no character count argument is specified, one character is transferred.

Form: i = MTOUT (unit,char[,chrcnt])

where:

unit	is the unit number of the terminal.
char	is the variable containing the characters to be output, right-justified in the integer (can be LOGICAL*1 if desired).
chrcnt	is an optional argument that indicates the number of characters to be output.

Errors:

i = 0	Normal return
i = 1	No room in output ring buffer.
i = 2	Unit not attached
i = 3	Non-existent unit number

MTPRNT

4.3.77 MTPRNT (FB and XM Only)

The MTPRNT subroutine operates the same as the PRINT subroutine (Section 4.3.81) in a multi-terminal environment. It allows output to be printed at the console terminal (see Section 2.4 for more details)

Form: i = MTPRNT (unit,string)

where: string — is the character string to be printed. Note that all quoted literals used in FORTRAN subroutine calls are in ASCIZ format. All character strings produced by the SYSF4 string handling package are also in the ASCIZ format.

unit — is the unit number associated with the terminal.

Errors:

i = 0	Normal return
i = 2	Unit not attached
i = 3	Non-existent unit number

MTRCTO

4.3.78 MTRCTO (FB and XM Only)

The MTRCTO subroutine operates the same as the .RCTRLO programmed request in a multi-terminal environment. This request resets the CTRL/O command originated at the console terminal.

Form: i = MTRCTO(unit)

where: unit — is the unit number associated with the terminal.

Errors:

i = 0	Normal return
= 2	Unit not attached
= 3	Non-existent unit number

MTSET

4.3.79 MTSET (FB and XM Only)

The MTSET subroutine allows the user program to set terminal and line characteristics. (See .MTSET program request in Chapter 2 for more details.)

Form: i = MTSET (unit,addr)

where: addr — is a four-word area to pass the status information. The area is a four-element INTEGER*2 array. See Section 2.4 for area format.

unit — is the unit number of the line and terminal whose characteristics are to be changed.

Errors:

i = 0	Normal return
= 2	Unit not attached
= 3	Non-existent unit number

= 6 In the XM monitor, the terminal status buffer address is outside legal program limits.

Example:

```
      PROGRAM MULTEM
C     MULTEM.FOR SYSF4 TEST FOR MULTI-TERMINAL ROUTINES
C
      DIMENSION IADDR(2,4)               !(IUNIT,STATUS WD)
      LOGICAL*1 PROMPT(8),ISTRNG(134)
      DATA PROMPT/'S','T','R','I','N','G','>','"200/
C
      CALL PRINT ('THE FOLLOWING NUMBERS ACTIVATE CERTAIN FUNCTIONS')
      CALL PRINT ('1 = MTSET')
      CALL PRINT ('2 = MTGET')
      CALL PRINT ('3 = MTIN')
      CALL PRINT ('4 = MTOUT')
      CALL PRINT ('5 = MTRCTO')
      CALL PRINT ('6 = MTATCH')
      CALL PRINT ('7 = MTDTCH')
      CALL PRINT ('8 = MTPRNT')
5     TYPE 19
      ACCEPT 8,IFUN                      !GET FUNCTION TO DO
      GOTO (100,200,300,400,500,600,700,800),IFUN
C
100   TYPE 109
109   FORMAT ('$SETUP TERMINAL STATUS BLOCK,STATUS WORD 1 ? ')
      ACCEPT 9,IADDR(IUNIT,1)
      IADDR(IUNIT,2) = 0
      TYPE 129
129   FORMAT ('$FILLER CHARACTER ? ')
      ACCEPT 9,J
      TYPE 139
139   FORMAT ('$NUMBER OF FILLERS ? ')
      ACCEPT 8,I
      IADDR(IUNIT,3)=I*256 + J
      TYPE 149
149   FORMAT ('$CARRIAGE WIDTH ? ')
      ACCEPT 8,J
      TYPE 159
159   FORMAT ('$STATE BYTE ? ')
      ACCEPT 8,I
      IADDR(IUNIT,4) = I*256 + J
      IERR = MTSET (IUNIT, IADDR(IUNIT,1))
      GOTO 999
C
200   IERR = MTGET (IUNIT, IADDR(IUNIT,1))
      TYPE 209,(IADDR(IUNIT,I),I=1,4)
209   FORMAT (' 4 WORD STATUS BLK IS:',4O6)
      GOTO 999
C
300   IERR = MTIN (IUNIT, ICHAR)
305   TYPE 309,ICHAR
309   FORMAT (' ICHAR=',A1)
      GOTO 999
C
400   IERR = MTOUT (IUNIT, ICHAR)
      GOTO 305
C
500   IERR = MTRCTO (IUNIT)
      GOTO 999
C
600   TYPE 609
```

```
609   FORMAT ('$IUNIT ? ')
      ACCEPT 8,IUNIT
      TYPE 619
619   FORMAT ('$ASYNCHRONOUS WORD ? ')
      ACCEPT 9,IASYN
      IERR = MTATCH (IUNIT, IASYN)
      GOTO 999
C
700   IERR = MTDTCH (IUNIT)
      GOTO 999
C
800   CALL GTLIN (ISTRNG,PROMPT)
      IERR = MTPRNT (IUNIT, ISTRNG)
999   TYPE 998,IERR
      GOTO 5
C
8     FORMAT (I4)
9     FORMAT (O6)
19    FORMAT ('$FUNCTION ? ')
998   FORMAT (' IERR =',I4)
      END
```

MWAIT

4.3.80 MWAIT (FB and XM Only)

The MWAIT subroutine suspends main program execution of the current job until all messages sent to or from the other job have been transmitted or received. It provides a means for ensuring that a required message has been processed. MWAIT is used primarily in conjunction with the IRCVD and ISDAT calls, where no action is taken when a message transmission is completed. This subroutine requires a queue element; this should be considered when the IQSET function (Section 4.3.37) is executed.

Form: CALL MWAIT

Errors:

None.

Example:

See the example under ISDAT, Section 4.3.45.

PRINT

4.3.81 PRINT

The PRINT subroutine causes output (from a specified string) to be printed at the console terminal. This routine can be used to print messages from completion routines without using the FORTRAN formatted I/O system. Control returns to the user program after all characers have been placed in the output buffer.

The string to be printed can be terminated with either a null (0) byte or a 200 (octal) byte. If the null (ASCIZ) format is used, the output is automatically followed by a carriage return/line feed pair (octal

15 and 12). If a 200 byte terminates the string, no carriage return/line feed pair is generated.

In the FB monitor, a change in the job that is controlling terminal output is indicated by a B> or F>. Any text following the message has been printed by the job indicated (foreground or background) until another B> or F> is printed. When PRINT is used by the foreground job, the message appears immediately, regardless of the state of the background job. Thus, for urgent messages, PRINT should be used rather than ITTOUR.

Form: CALL PRINT (string)

where: string is the string to be printed. Note that all quoted literals used in FORTRAN subroutine calls are in ASCIZ format as are all strings produced by the SYSF4 string handling package.

Errors:

None.

Example:

```
CALL PRINT ('THE COFFEE IS READY')
```

PURGE

4.3.82 PURGE

The PURGE subroutine is used to deactivate a channel without performing an ISAVES or a CLOSEC. Any tentative file currently associated with the channel is not made permanent. This subroutine is useful for keeping ENTERed (IENTER or .ENTER) files from becoming permanent directory entries.

Form: CALL PURGE (chan)

where: chan is the integer specification for the RT-11 channel to be deactivated.

Errors:

None.

Example:

See the example under IENTER, Section 4.3.22.

PUTSTR

4.3.83 PUTSTR

The PUTSTR subroutine writes a variable-length character string to a specified FORTRAN logical unit. PUTSTR can be used in main program routines or in completion routines but not in both in the same program at the same time. If PUTSTR is used in a completion routine, it must not be the first I/O operation on the specified logical unit.

Form: CALL PUTSTR (lun,in,char,err)

where:

- lun — is the integer specification of the FORTRAN logical unit number to which the string is to be written.
- in — is the array containing the string to be written.
- char — is an ASCII character that is appended to the beginning of the string before it is output. If 0, the first character of in is the first character of the record. This character is used primarily for carriage control purposes (see Section 4.3.13).
- err — is a LOGICAL*1 variable that is .TRUE for an error condition and .FALSE for a no error condition.

Errors:

ERR = -1	End-of-file for write operation
-2	Hardware error for write operation

Example:

```
LOGICAL*1 STRNG(81)
.
.
.
CALL PUTSTR(7,STRNG,'0')  !OUTPUT STRING WITH DOUBLE SPACING
```

R50ASC

4.3.84 R50ASC

The R50ASC subroutine converts a specified number of Radix-50 characters to ASCII.

Form: CALL R50ASC (icnt,input,output)

where:

- icnt — is the integer number of ASCII characters to be produced.
- input — is the area from which words of Radix-50 values to be converted are taken. Note that (icnt+2)/3 words are read for conversion.
- output — is the area into which the ASCII characters are stored.

Errors:

If an input word contains illegal Radix-50 codes (that is, if the input word is greater (unsigned) than 174777(octal)), the routine outputs question marks for the value.

Example:

```
REAL*8 NAME
LOGICAL*1 OUTP(12)
.
.
.
CALL R50ASC(12,NAME,OUTP)
```

RAD50

4.3.85 RAD50

The RAD50 function provides a method of encoding RT-11 file descriptors in Radix-50 notation. The RAD50 function converts six ASCII characters from the specified area, returning a REAL*4 result that is the two-word Radix-50 value.

Form: a = RAD50 (input)

where: input is the area from which the ASCII input characters are taken.

The RAD50 call:

```
A = RAD50 (LINE)
```

is exactly equivalent to the IRAD50 call:

```
CALL IRAD50 (6,LINE,A)
```

Function Results:

The two-word Radix-50 value is returned as the function result.

RCHAIN

4.3.86 RCHAIN

The RCHAIN subroutine allows a program to determine whether it has been chained to and to access variables passed across a chain. If RCHAIN is used, it must be used in the first executable FORTRAN statement in a program. RCHAIN cannot be called from a completion or interrupt routine.

Form: CALL RCHAIN (flag,var,wcnt)

where: flag is an integer variable that is set to -1 if the program has been chained to; otherwise, it is 0.

var is the first variable in a sequence of variables with increasing memory addresses to receive the information passed across the chain (see Section 4.3.2).

wcnt	is the number of words to be moved from the chain parameter area to the area specified by var. RCHAIN moves wcnt words into the area beginning at var.

Errors:

None.

Example:

```
INTEGER*2 PARMS(50)
CALL RCHAIN(IFLAG,PARMS,50)
IF(IFLAG) GOTO 10       !GOTO 10 IF CHAINED TO
.
.
.
```

RCTRLO

4.3.87 RCTRLO

The RCTRLO subroutine resets the effect of any console terminal CTRL/O command that was typed. After an RCTRLO call, any output directed to the console terminal prints until another CTRL/O is typed.

Form: CALL RCTRLO

Errors:

None.

Example:

```
CALL RCTRLO
CALL PRINT ('THE REACTOR IS ABOUT TO BLOW UP')
```

REPEAT

4.3.88 REPEAT

The REPEAT subroutine concatenates a specified string with itself to produce the indicated number of copies. REPEAT places the resulting string in a specified array.

Form: CALL REPEAT (in,out,i[,len[,err]])

where:	in	is the array containing the string to be repeated.
	out	is the array into which the resultant string is placed. This array must be at least one element longer than the value of len, if specified.
	i	is the integer number of times to repeat the string.

len	is the integer number representing the maximum length of the output string.
err	is the logical error flag set if the output string is truncated to the length specified by len.

Input and output strings can specify the same array only if the repeat count (i) is 1 or 0. When the repeat count is 1, this routine is the equivalent of SCOPY; when the repeat count is 0, out is replaced by a null string. The old contents of out are lost when this routine is called.

Errors:

Error conditions are indicated by err, if specified. If err is given and the output string would have been longer than len characters, then err is set to .TRUE.; otherwise, err is unchanged.

Example:

```
LOGICAL*1 SIN(21),SOUT(101)
.
.
.
CALL REPEAT(SIN,SOUT,5)
```

RESUME

4.3.89 RESUME (FB and XM Only)

The RESUME subroutine allows a job to resume execution of the main program. A RESUME call is normally issued from an asynchronous FORTRAN routine entered on I/O completion or because of a schedule request. (See Section 4.3.97 for more information.)

Form: CALL RESUME

Errors:

None.

Example:

See the example under SUSPND, Section 4.3.97.

SCCA

4.3.90 SCCA

The SCCA subroutine provides a CTRL/C intercept to perform the following functions:

1. Inhibit a CTRL/C abort
2. Indicate that a CTRL/C command is active
3. Distinguish between single and double CTRL/C commands

Form: CALL SCCA [(iflag)]

where: iflag is an integer terminal status word that must be tested and cleared to determine if two CTRL/Cs were typed at the console terminal. The iflag must be an INTEGER*2 variable (not LOGICAL*1).

When a CTRL/C is typed, the SCCA subroutine, having been previously called, makes the CTRL/C command inactive and places it in the input ring buffer. While residing in the buffer, the inactive command can be read as a valid character by the program. The program must test and clear the iflag to determine if two CTRL/C commands were typed consecutively. The iflag is set to non-zero when two CTRL/Cs are typed together. It is the responsibility of the program to abort itself, if appropriate, on an input of CTRL/C from the terminal. The SCCA subroutine with no argument disables the CTRL/C intercept.

Errors:

None

Example:

```
      PROGRAM SCCA
C     SCCA.FOR SYSF4 TEST FOR SCCA
C
      CALL PRINT ('PROGRAM HAS STARTED, TYPE')
      IFLAG=0
      CALL SCCA (IFLAG)
10    I = ITTINR()              !GET A CHARACTER
      IF (I .NE. 3) GOTO 10
C     A CTRL/C WAS TYPED
      CALL PRINT ('A CTRL/C WAS TYPED')
      IF (IFLAG .EQ. 0) GOTO 10
      CALL PRINT ('A DOUBLE CTRL/C WAS TYPED')
      TYPE 19,IFLAG
19    FORMAT (' IFLAG = ',06,/)
      CALL SCCA                 !DISABLE CTRL/C INTERCEPT
      CALL PRINT ('TYPE A CTRL/C TO EXIT')
20    GOTO 20                   !LOOP UNTIL CTRL/C TYPED
      END
```

SCOMP

4.3.91 SCOMP

The SCOMP routine compares two character strings and returns the integer result of the comparison.

Form: CALL SCOMP (a,b,i)

or

i = ISCOMP (a,b)

where: a is the array containing the first string.

b is the array containing the second string.

i — is the integer variable that receives the result of the comparison.

The strings are compared left to right, one character at a time, using the collating sequence specified by the ASCII codes for each character. If the two strings are not equal, the absolute value of variable i (or the result of the function ISCOMP) is the character position of the first inequality found in scanning left to right. Strings are terminated by a null (0) character.

If the strings are not the same length, the shorter one is treated as if it were padded on the right with blanks to the length of the other string. A null string argument is equivalent to a string containing only blanks.

Function Result:

i <0	if a is less than b
=0	if a is equal to b
>0	if a is greater than b

Example:

```
LOGICAL*1 INSTR(81)
.
.
.
CALL GETSTR(5,INSTR,80)
CALL SCOMP('YES',INSTR,IVAL)
IF(IVAL) GOTO 10      !IF INPUT STRING IS NOT YES GOTO 10
```

SCOPY

4.3.92 SCOPY

The SCOPY routine copies a character string from one array to another. Copying stops either when a null (0) character is encountered or when a specified number of characters have been moved.

Form: CALL SCOPY (in,out[,len[,err]])

where:

- in — is the array containing the string to be copied.
- out — is the array to receive the copied string. This array must be at least one element longer than the value of len, if specified.
- len — is the integer number representing the maximum length of the output string. The effect of len is to truncate the output string to a given length, if necessary.
- err — is a logical variable that receives the error indication if the output string was truncated to the length specified by len.

The input (in) and output (out) arguments can specify the same array. The string previously contained in the output array is lost when this subroutine is called.

Errors:

Error conditions are indicated by err, if specified. If err is given and the output string was truncated to the length specified by len, then err is set to .TRUE.; otherwise, err is unchanged.

Example:

SCOPY is useful for initializing strings to a constant value, for example:

```
LOGICAL*1 STRING(80)
CALL SCOPY('THIS IS THE INITIAL VALUE',STRING)
```

SECNDS

4.3.93 SECNDS

The SECNDS function returns the current system time, in seconds past midnight, minus the value of a specified argument. Thus, SECNDS can be used to calculate elapsed time. The value returned is single-precision floating point (REAL*4).

Form: a = SECNDS (atime)

where: atime is a REAL*4 variable, constant, or expression whose value is subtracted from the current time of day to form the result.

Notes:

This function does floating-point arithmetic. Elapsed time can also be calculated by using the GTIM call and the INTEGER*4 support functions.

Function Result:

The function result (a) is the REAL*4 value returned. Example:

```
C     START OF TIMED SEQUENCE
      T1=SECNDS(0.)
C
C     CODE TO BE TIMED GOES HERE
C
      DELTA=SECNDS(T1)    !DELTA IS ELAPSED TIME
```

SETCMD

4.3.94 SETCMD

The SETCMD routine allows a user program to pass a command line to the keyboard monitor to be executed after the program exits. The command lines are passed to the chain information area (500-777(octal)) and stored beginning at location 512(octal). No check is made to determine if the string extends into the stack space. For this reason, the command line should be short and the subroutine call should be made in the main program unit near the end of the program just before completion. If several commands are desired to be executed, an indirect command file that contains many command lines should be used.

The following monitor commands are disallowed if the SETCMD feature is used.

1. REENTER
2. START
3. CLOSE

Form: CALL SETCMD (string)

where: string is a keyboard monitor command line in ASCIZ format with no embedded carriage returns or line feeds.

Errors

None

Example:

```
LOGICAL*1 INPUT(134),PROMPT(8)
DATAPROMPT/'P','R','O','M','P','T','>','200/
CALL GTLIN (INPUT,PROMPT)
CALL SETCMD (INPUT)
END
```

NOTE

Set USR NOSWAP or specify /NOSWAP with the COMPILE, FORTRAN, or EXECUTE command.

STRPAD

4.3.95 **STRPAD**

The STRPAD routine pads a character string with rightmost blanks until that string is a specified length. This padding is done in place; the result string is contained in its original array. If the present length of the string is greater than or equal to the specified length, no padding occurs.

Form: CALL STRPAD (a,i[,err])

where:

a is the string to be padded.

i is the integer length of the desired result string.

err is the logical error flag that is set to .TRUE. if the string specified by a exceeds the value of i in length.

Errors:

Error conditions are indicated by err, if specified. If err is given and the string indicated is longer than i characters, err is set to .TRUE.; otherwise, the value of err is unchanged.

Example:

This routine is especially useful for preparing strings to be output in A-type FORMAT fields. For example:

```
         LOGICAL*1 STR(81)
         .
         .
         .
         CALL STRPAD(STR,80)        !ASSURE 80 VALID CHARACTERS
         PRINT 100,(STR(I),I=1,80) !PRINT STRING OF 80 CHARACTERS
     100 FORMAT(80A1)
```

SUBSTR

4.3.96 SUBSTR

The SUBSTR routine copies a substring from a specified position in a character string. If desired, the substring can then be placed in the same array as the string from which it was taken.

Form: CALL SUBSTR (in,out,i[,len])

where:

- in — is the array from which the substring is taken.
- out — is the array to contain the substring result. This array must be one element longer than len, if specified.
- i — is the integer character position in the input string of the first character of the desired substring.
- len — is the integer number of characters representing the maximum length of the substring.

If a maximum length (len) is not given, the substring contains all characters to the right of character position i in array in. If len is equal to zero, out is replaced by the null string. The old contents of array out are lost when this routine is called.

Errors:

None.

SUSPND

4.3.97 SUSPND (FB and XM Only)

The SUSPND subroutine suspends main program execution of the current job and allows only completion routines (for I/O and scheduling requests) to run.

Form: CALL SUSPND

Notes:

1. The monitor maintains a suspension counter for each job. This count is decremented by SUSPND and incremented by RESUME (see Section 4.3.81). A job will actually be suspended only if this counter is negative. Thus, if a RESUME is issued before a SUSPND, the latter function will return immediately.

2. A program must issue an equal number of SUSPNDs and RESUMEs.

3. A RESUME subroutine call from the main program or from a completion routine increments the suspension counter.

4. A SUSPND subroutine call from a completion routine decrements the suspension counter but does not suspend the main program. If a completion routine does a SUSPND, the main program continues until it also issues a SUSPND, at which time it is suspended and requires two RESUMEs to proceed.

5. Because SUSPND and RESUME are used to simulate an ITWAIT (see Section 4.3.45) in the monitor, a RESUME issued from a completion routine and not matched by a previously executed SUSPND can cause the main program execution to continue past a timed wait before the entire time interval has elapsed.

For further information on suspending main program execution of the current job, see the assembly language .SPND request, Section 2.4.

Errors:

None.

Example:

```
      INTEGER IAREA(4)
      COMMON /RDBLK/ IBUF(256)
      EXTERNAL RDFIN
      .
      .
      .
      IF(IREADF(256,IBUF,IBLK,ICHAN,IAREA,RDFIN).NE.0) GOTO 1000
C     GOTO 1000 FOR ANY TYPE OF ERROR
C
C     DO OVERLAPPED PROCESSING
      .
      .
      .
      CALL SUSPND      !SYNCHRONIZE WITH COMPLETION ROUTINE
      .
      .
      .
      END
```

```
SUBROUTINE RDFIN(IARG1,IARG2)
COMMON /RDBLK/ IBUF(256)
.
.
.
CALL RESUME      !CONTINUE MAIN PROGRAM
.
.
.
END
```

TIMASC

4.3.98 TIMASC

The TIMASC subroutine converts a two-word internal format time into an ASCII string of the form:

hh:mm:ss

where:	hh	is the two-digit hours indication
	mm	is the two-digit minutes indication
	ss	is the two-digit seconds indication

Form: CALL TIMASC (itime,strng)

where:	itime	is the two-word internal format time to be converted. itime (1) is the high-order time. itime (2) is the low-order time.
	strng	is the eight-element array to contain the ASCII time.

Errors:

None.

Example:

The following example determines the amount of time until 5 p.m. and prints it.

```
      INTEGER*4 J1,J2,J3
      LOGICAL*1 STRNG(8)
      .
      .
      .
      CALL JTIME(17,0,0,0,J1)
      CALL GTIM(J2)
      CALL JJCVT(J1)
      CALL JJCVT(J2)
      CALL JSUB(J1,J2,J3)
      CALL JJCVT(J3)
      CALL TIMASC(J3,STRNG)
      TYPE 99,(STRNG(I),I=1,8)
99    FORMAT(' IT IS ',8A1,' TILL 5 P.M.')
      .
      .
      .
```

TIME

4.3.99 TIME

The TIME subroutine returns the current system time of day as an eight-character ASCII string of the form:

hh:mm:ss

where: hh is the two-digit hours indication

mm is the two-digit minutes indication

ss is the two-digit seconds indication

Form: CALL TIME (strng)

where: strng is the eight-element array to receive the ASCII time.

Notes:

A 24-hour clock is used (for example, 1:00 p.m. is represented as 13:00:00). The DATE and IDATE subroutines are available as part of FORTRAN IV system routines.

Errors:

None.

Example:

```
        LOGICAL*1 STRNG(8)
        .
        .
        .
        CALL TIME(STRNG)
        TYPE 99,(STRNG(I),I=1,8)
 99     FORMAT (' IT IS NOW ',8A1)
```

TRANSL

4.3.100 TRANSL

The TRANSL routine performs character translation on a specified string. The TRANSL routine requires approximately 64 words on the R6 stack for its execution. This space should be considered when allocating stack space.

Form: CALL TRANSL (in,out,r[,p])

where: in is the array containing the input string.

out is the array to receive the translated string.

r — is the array containing the replacement string.

p — is the array containing the characters in in to be translated.

The string specified by array out is replaced by the string specified by array in, modified by the character translation process specified by arrays r and p. If any character position in in contains a character that appears in the string specified by p, it is replaced in out by the corresponding character from string r. If the array p is omitted, it is assumed to be the 127 seven-bit ASCII characters arranged in ascending order, beginning with the character whose ASCII code is 001. If strings r and p are given and differ in length, the longer string is truncated to the length of the shorter. If a character appears more than once in string p, only the last occurrence is significant. A character can appear any number of times in string r.

Errors:

None.

Examples:

The following example causes the string in array A to be copied to array B. All periods within A become minus signs, and all question marks become exclamation points.

```
CALL TRANSL(A,B,'-!','.?')
```

The following is an example of TRANSL being used to format character data.

```
      LOGICAL*1 STRING(27),RESULT(27),PATRN(27)
C     SET UP THE STRING TO BE REFORMATTED
C
      CALL SCOPY('THE HORN BLOWS AT MIDNIGHT',STRING)
C
C     00000000011111111112222222
C     12345678901234567890123456
C     THE HORN BLOWS AT MIDNIGHT
C     NOW SET UP PATRN TO CONTAIN THE FOLLOWING PATTERN:
C     16,17,18,19,20,21,22,23,24,25,26,15,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0
C
      DO 10 I=16,26
10    PATRN(I-15)=I
      PATRN(12)=15
      DO 20 I=1,14
20    PATRN(I+12)=I
      PATRN(27)=0
C
C     THE FOLLOWING CALL TO TRANSL REARRANGES THE CHARACTERS OF
C     THE INPUT STRING TO THE ORDER SPECIFIED BY PATRN:
C
      CALL TRANSL(PATRN,RESULT,STRING)
C
C     RESULT NOW CONTAINS THE STRING 'AT MIDNIGHT THE HORN BLOWS'
C     IN GENERAL, THIS METHOD CAN BE USED TO FORMAT INPUT STRINGS
C     OF UP TO 127 CHARACTERS.  THE RESULTANT STRING WILL BE
C     AS LONG AS THE PATTERN STRING (AS IN THE ABOVE EXAMPLE).
```

TRIM

4.3.101 TRIM

The TRIM routine shortens a specified character string by removing all trailing blanks. A trailing blank is a blank that has no non-blanks to its right. If the specified string contains all blank characters, it is replaced by the null string. If the specified string has no trailing blanks, it is unchanged.

Form: CALL TRIM (a)

where: a — is the array containing the string to be trimmed.

Errors:

None.

Example:

```
     LOGICAL*1 STRING(81)
     ACCEPT 100,(STRING(I),I=1,80)
100  FORMAT(80A1)
     CALL SCOPY(STRING,STRING,80)          !MAKE ASCIZ
     CALL TRIM(STRING)                     !TRIM TRAILING BLANKS
```

UNLOCK

4.3.102 UNLOCK

The UNLOCK subroutine releases the User Service Routine (USR) from memory if it was placed there by the LOCK routine. If the LOCK required a swap, the UNLOCK loads the user program back into memory. If the USR does not require swapping, the UNLOCK involves no I/O. The USR is always resident in XM.

Form: CALL UNLOCK

Notes:

1. It is important that at least as many UNLOCKS are given as LOCKs. If more LOCKs were done, the USR remains locked in memory. It is not harmful to give more UNLOCKS than are required; those that are extra are ignored.

2. When running two jobs in the FB system, use the LOCK/UNLOCK pairs only when absolutely necessary. If one job LOCKs the USR, the other job cannot use the USR until it is UNLOCKed. Thus, the USR should not be LOCKed unnecessarily, as this may degrade performance in some cases.

3. In an FB system, calling the CSI (ICSI) with input coming from the console terminal performs an implicit UNLOCK.

For further information on releasing the USR from memory, see the assembly language .LOCK/.UNLOCK requests, Section 2.4.

Errors:

None.

Example:

```
     .
     .
     .
C    GET READY TO DO MANY USR OPERATIONS
     CALL LOCK      !DISABLE USR SWAPPING
C    PERFORM THE USR CALLS
     .
     .
     .
C    FREE THE USR
     CALL UNLOCK
     .
     .
     .
```

VERIFY

4.3.103 VERIFY

The VERIFY routine determines whether each character of a specified string occurs anywhere in another string. If a character does not exist in the string being examined, VERIFY returns its character position in string b. If all characters exist, VERIFY returns a 0.

Form: CALL VERIFY (a,b,i)

or

i = IVERIF (a,b)

where:

	a	is the array containing the string to be scanned.
	b	is the array containing the string of characters to be accepted in a.
	i	is the integer result of the verification.

Function Result:

i = 0	if all characters of a exist in b; also if a is a null string.
= n	where n is the character position of the first character in a that does not appear in b; if b is a null string and a is not, i equals 1.

Example:

The following example accepts a one- to five-digit unsigned decimal number and returns its value.

```
        LOGICAL*1 INSTR(81)
        .
        .
        .
        CALL VERIFY(INSTR(IPOS),'0123456789',I)
        IF(I.EQ.1) STOP 'NUMBER MISSING'
        IF(I.EQ.0) I=LEN(INSTR)-IPOS+1
        IF(I.GT.5) STOP 'TOO MANY DIGITS'
        NUM=IVALUE(INSTR(IPOS),I)
        .
        .
        .
        END
        FUNCTION IVALUE(ARRAY,I)
        LOGICAL*1 ARRAY(1)
        DECODE(I,99,ARRAY) IVALUE
   99   FORMAT(I5)
        END
```

APPENDIX A

DISPLAY FILE HANDLER

This appendix describes the assembly language support provided under RT-11 for the VT11 graphic display hardware systems.

The following manuals are suggested for additional reference:

GT40/GT42 User's Guide
EK-GT40-OP-002

GT44 User's Guide
EK-GT44-OP-001

VT11 Graphic Display Processor
EK-VT11-TM-001

DECGRAPHIC-11 GT Series Reference Card
EH-02784-73

DECGraphic-11 FORTRAN Reference Manual
DEC-11-GFRMA-A-D

BASIC-11 Graphics Extensions User's Guide
DEC-11-LBGEA-A-D

A.1 DESCRIPTION

The graphics display terminals have hardware configurations that include a display processor and CRT (cathode ray tube) display. All systems are equipped with light pens and hardware character and vector generators, and are capable of high-quality graphics. The Display File Handler supports this graphics hardware at the assembly language level under the RT-11 monitor.

A.1.1 Assembly Language Display Support

The Display File Handler is not an RT-11 device handler, since it does not use the I/O structure of the RT-11 monitor. For example, it is not possible to use a utility program to transfer a text file to the display through the Display File Handler. Rather, the Display File Handler provides the graphics programmer the means for the display of graphics files and the easy management of the display processor. Included in its capabilities are such services as interrupt handling, light pen support, tracking object, and starting and stopping of the display processor.

The Display File Handler manages the display processor by means of a base segment (called VTBASE) which contains interrupt handlers, an internal display file and some pointers and flags. The display processor cycles through the internal display file; any user graphics files to be displayed are accessed by display subroutine calls from the Handler's display file. In this way, the Display File Handler exerts control over the display processor, relieving the assembly language user of the task.

Through the Display File Handler, the programmer can insert and remove calls to display files from the Handler's internal display file. Up to two user files may be inserted at one time, and that number may be increased by re-assembling the Handler. Any user file inserted for display may be blanked (the subroutine call to it bypassed) and unblanked by macro calls to the Display File Handler.

Since the Handler treats all user display files as graphics subroutines to its internal display file, a display processor subroutine call is required. This is implemented with software, using the display stop instruction, and is available for user programs. This instruction and several other extended instructions implemented with the display stop instruction are described in Section A.3.

The facilities of the Display File Handler are accessed through a file of macro definitions (VTMAC) which generate calls to a set of subroutines in VTLIB. VTMAC's call protocol is similar to that of the RT-11 macros. The expansion of the macros is shown in Section A.6. VTMAC also contains, for convenience in programming, the set of recommended display processor instruction mnemonics and their values. The mnemonics are listed in Section A.7 and are used in the examples throughout this appendix.

VTCAL1 through VTCAL4 are the set of subroutines which service the VTMAC calls. They include functions for display file and display processor management. These are described in detail in Section A.2. VTCAL1 through VTCAL4 are distributed, along with the base segment VTBASE, as a file of five object modules called VTHDLR.OBJ. VTHDLR is built into the graphics library VTLIB by using the monitor LIBRARY command. Section A.4.2 shows an example.

A.1.2 Monitor Display Support

The RT-11 monitor, under Version 03, directly supports the display as a console device. A keyboard monitor command, GT ON (GT OFF) permits the selection of the display as console device. Selection results in the allocation of approximately 1.25K words of memory for text buffer and code. The buffer holds approximately 2000 characters.

The text display includes a blinking cursor to indicate the position in the text where a character is added. The cursor initially appears at the top left corner of the text area. As lines are added to the text the cursor moves down the screen. When the maximum number of lines are on the screen, the top line is deleted from the text buffer when the line feed terminating a new line is received. This causes the appearance of "scrolling", as the text disappears off the top of the display.

When the maximum number of characters have been inserted in the text buffer, the scroller logic deletes a line from the top of the screen to make room for additional characters. Text may appear to move (scroll) off the top of the screen while the cursor is in the middle of a line.

The Display File Handler can operate simultaneously with the scroller program, permitting graphic displays and monitor dialogue to appear on the screen at the same time. It does this by inserting its internal display file into the display processor loop through the text buffer. However, the following should be noted. Under the SJ Monitor, if a program using the display for graphics is running with the scroller in use (that is, GT ON is in effect), and the program does a soft exit (.EXIT with R0 not equal to 0) with the display stopped, the display remains stopped until a CTRL/C is typed at the keyboard.

This can be recognized by failure of the monitor to echo on the screen when expected. If the scroller text display disappears after a program exit, always type CTRL/C to restore. If CTRL/C fails to restore the display, the running program probably has an error.

Four scroller control characters provide the user with the capability of halting the scroller, advancing the scrolling in page sections, and printing hard copy from the scroller.

NOTE

> The scroller logic does not limit the length of a line, but the length of text lines affects the number of lines which may be displayed, since the text buffer is finite. As text lines become longer, the scroller logic may delete extra lines to make room for new text, temporarily decreasing the number of lines displayed.

A.2 DESCRIPTION OF GRAPHICS MACROS

The facilities of the Display File Handler are accessed through a set of macros, contained in VTMAC, which generate assembly language calls to the Handler at assembly time. The calls take the form of subroutine calls to the subroutines in VTLIB. Arguments are passed to the subroutines through register 0 and, in the case of the .TRACK call, through both register 0 and the stack.

This call convention is similar to Version 1 RT-11 I/O macro calls, except that the subroutine call instruction is used instead of the EMT instruction. If a macro requires an argument but none is specified, it is assumed that the address of the argument has already been placed in register 0. The programmer should not assume that R0 is preserved through the call.

A.2.1 .BLANK

The .BLANK request temporarily blanks the user display file specified in the request. It does this by by-passing the call to the user display file, which prevents the display processor from cycling through the user file, effectively blanking it. This effect can later be cancelled by the .RESTR request, which restores the user file. When the call returns, the user is assured the display processor is not in the file that was blanked.

Macro Call: .BLANK faddr

where:	faddr	is the address of the user display file to be blanked.

Errors:

No error is returned. If the file specified was not found in the Handler file or has already been blanked, the request is ignored.

A.2.2 .CLEAR

The .CLEAR request initializes the Display File Handler, clearing out any calls to user display files and resetting all of the internal flags and pointers.

After initialization with .LNKRT (Section A.2.4), the .CLEAR request can be used any time in a program to clear the display and to reset pointers. All calls to user files are deleted and all pointers to status buffers are reset. They must be re-inserted if they are to be used again.

Macro Call: .CLEAR

Errors:

None.

Example:

This example uses a .CLEAR request to initialize the Handler then later uses the .CLEAR to re-initialize the display. The first .CLEAR is used for the case when a program may be restarted after a CTRL C or other exit.

```
                BR RSTRT

EX1:            BIS #20000,@#44     ;SET REENTER BIT IN JSW
RSTRT:          .UNLNK              ;CLEARS LINK FLAG FOR RESTART
                .LNKRT              ;SET UP VECTORS, START DISPLAY
                .CLEAR              ;INITIALIZE HANDLER
                .INSRT #FILE1       ;DISPLAY A PICTURE
1$:             .TTYIN              ;WAIT FOR A KEY STRIKE
                CMPB #12,R0         ;LINE FEED?
                BNE 1$              ;NO, LOOP
                .CLEAR              ;YES, CLEAR DISPLAY
                .INSRT #FILE2       ;DISPLAY NEW PICTURE
                    .
                    .
                    .
FILE1:          POINT           ;AT POINT (0,500)
                0
                500
                LONGV           ;DRAW A LINE
                500!INTX        ;TO (500,500)
                0
                DRET
                0

FILE2:          POINT           ;AT POINT (500,0)
                500
                0
                LONGV           ;DRAW A LINE
                0!INTX          ;TO (500,500)
                500
                DRET
                0

                .END EX1
```

A.2.3 .INSRT

The .INSRT request inserts a call to the user display file specified in the request into the Display File Handler's internal display file. .INSRT causes the display processor to cycle through the user file as a subroutine to the internal file. The handler permits two user files at one time. The call inserted in the handler looks like the following:

```
DJSR        ;DISPLAY SUBROUTINE
.+4         ;RETURN ADDRESS
.faddr      ;SUBROUTINE ADDRESS
```

The call to the user file is removed by replacing its address with the address of a null display file. The user file is blanked by replacing the DJSR with a DJMP instruction, bypassing the user file.

Macro Call: .INSRT faddr

where: faddr is the address of the user display file to be inserted.

Errors:

The .INSRT request returns with the C bit set if there was an error in processing the request. An error occurs only when the Handler's display file is full and cannot accept another file. If the user file specified exists, the request is not processed. Two display files with the same starting address cannot be inserted.

Example:

See the examples in Sections A.2.2 and A.2.4.

A.2.4 .LNKRT

The .LNKRT request sets up the display interrupt vectors and possibly links the Display File Handler to the scroll text buffer in the RT-11 monitor. It must be the first call to the Handler, and is used whether or not the RT-11 monitor is using the display for console output (i.e., the KMON command GT ON has been entered).

The .LNKRT request used with the Version 03 RT-11 monitor enables a display application program to determine the environment in which it is operating. Error codes are provided for the situations where there is no display hardware present on the system or the display hardware is already being used by another task (e.g., a foreground job in the foreground/background version).

The existence of the monitor scroller and the size of the Handler's subpicture stack are also returned to the caller. If a previous call to .LNKRT was made without a subsequent .UNLNK, the .LNKRT call is ignored and an error code is returned.

Macro Call: .LNKRT

Errors:

Error codes are returned in R0, with the N condition bit set.

Code	Meaning
-1	No VT11 display hardware is present on this system.
-2	VT11 hardware is presently in use.
-3	Handler has already been linked.

On completion of a successful .LNKRT request, R0 will contain the display subroutine stack size, indicating the depth to which display subroutines may be nested. The N bit will be zero.

If the RT-11 monitor scroll text buffer was not in memory at the time of the .LNKRT, the C bit will be returned set. The KMON commands GT ON and GT OFF cannot be issued while a task is using the display.

Example:

```
START:  .LNKRT                  ;LINK TO MONITOR
        BMI     ERROR           ;ERROR DOING LINK
        BCS     CONT            ;NO SCROLL IF C SET
        .SCROL  #SBUF           ;ADJUST SCROLL PARAMETERS
CONT:   .INSRT  #FILE1          ;DISPLAY A PICTURE
1$:     .TTYIN                  ;WAIT FOR KEY STRIKE
        CMPB    #12,R0          ;LINE FEED?
        BNE     1$              ;NO, LOOP
        .UNLNK                  ;YES, UNLINK AND EXIT
        .EXIT

SBUF:   .BYTE   5               ;LINE COUNT OF 5
        .BYTE   7               ;INTENSITY 7 (SCALE OF 1-8)
        .WORD   1000            ;POSITION OF TOP LINE

FILE1:  POINT                   ;AT POINT (500,500)
        500
        500
        CHAR                    ;DISPLAY SOME TEXT
        .ASCII  /FILE1 THIS IS FILE1.  TYPE CR TO EXIT/
        .EVEN
        DRET
        0

ERROR:  Error routine
```

A.2.5 .LPEN

The .LPEN request transfers the address of a light pen status data buffer to VTBASE. Once the buffer pointer has been passed to the Handler, the light pen interrupt handler in VTBASE will transfer display processor status data to the buffer, depending on the state of the buffer flag.

The buffer must have seven contiguous words of storage. The first word is the buffer flag, and it is initially cleared (set to zero) by the .LPEN request. When a light pen interrupt occurs, the interrupt handler transfers status data to the buffer and then sets the buffer flag non-zero. The program can loop on the buffer flag when waiting for a light pen hit (although doing this will tie up the processor; in a foreground/background environment, timed waits would be more desirable). No further data transfers take place, despite the occurrence of numerous light pen interrupts, until the buffer flag is again cleared to zero. This permits the program to process the data before it is destroyed by another interrupt.

The buffer structure looks like this:

```
Buffer Flag
Name
Subpicture Tag
Display Program Counter (DPC)
Display Status Register (DSR)
X Status Register (XSR)
Y Status Register (YSR)
```

The Name value is the contents of the software Name Register (described in A.3.5) at the time of interrupt. The Tag value is the tag of the subpicture being displayed at the time of interrupt. The last four data items are the contents of the display processor status registers at the time of interrupt. They are described in detail in Table A-1.

Macro Call: .LPEN baddr

where: baddr is the address of the 7-word light pen status data buffer.

Errors:

None.

If a .LPEN was already issued and a buffer specified, the new buffer address replaces the previous buffer address. Only one light pen buffer can be in use at a time.

Example:

```
        .INSRT   #LFILE    ;DISPLAY LFILE
        .LPEN    #LBUF     ;SET UP LPEN BUFFER
LOOP:   TST      LBUF      ;TEST LBUF FLAG, WHICH
        BEQ      LOOP      ;WILL BE SET NON-ZERO
                           ;ON LIGHT PEN HIT.
;PROCESS DATA IN LBUF HERE.
                           ;DATA IN LBUF
        CLR      LBUF      ;CLEAR THE BUFFER FLAG
                           ;PERMITTING ANOTHER "HIT"
        BR       LOOP      ;GO WAIT FOR IT
```

```
LBUF:      .BLKW     7          ;SEVEN WORD LPEN BUFFER
LFILE:
               .
               .
```

Table A-1
Description of Display Status Words

Bits	Significance
DISPLAY PROGRAM COUNTER (DPC=172000)	
0-15	Address of display processor program counter at time of interrupt.
DISPLAY STATUS REGISTER (DSR=172002)	
0-1	Line Type
2	Spare
3	Blink
4	Italics
5	Edge Indicator
6	Shift Out
7	Light Pen Flag
8-10	Intensity
11-14	Mode
15	Stop Flag
X STATUS REGISTER (XSR=172004)	
0-9	X Position
10-15	Graphplot Increment
Y STATUS REGISTER (YSR=172006)	
0-9	Y Position
10-15	Character Register

A.2.6 .NAME

The .NAME request has been added to the Version 03 Display File Handler. The contents of the name register are now stacked when a subpicture call is made. When a light pen interrupt occurs, the contents of the name register stack may be recovered if the user program has supplied the address of a buffer through the .NAME request.

The buffer must have a size equal to the stack depth (default is 10) plus one word for the flag. When the .NAME request is entered, the address of the buffer is passed to the Handler and the first word (the flag word) is cleared. When a light pen hit occurs, the stack's contents are transferred and the flag is set non-zero.

Macro Call: .NAME baddr

where: baddr is the address of the name register buffer.

Errors:

None.

If a .NAME request has been previously issued, the new buffer address replaces the previous buffer address.

A.2.7 .REMOV

The .REMOV request removes the call to a user display file previously inserted in the handler's display file by the .INSRT request. All reference to the user file is removed, unlike the .BLANK request, which merely bypasses the call while leaving it intact.

Macro Call: .REMOV faddr

where: faddr is the address of the display file to be removed.

Errors:

No errors are returned. If the file address given cannot be found, the request is ignored.

A.2.8 .RESTR

The .RESTR request restores a user display file that was previously blanked by a .BLANK request. It removes the by-pass of the call to the user file, so that the display processor once again cycles through the user file.

Macro Call: .RESTR faddr

where: faddr is the address of the user file that is to be restored to view.

Errors:

No errors are returned. If the file specified cannot be found, the request is ignored.

A.2.9 .SCROL

This request is used to modify the appearance of the Display Monitor's text display. The .SCROL request permits the programmer to change the maximum line count, intensity and the position of the top line of text of the scroller. The request passes the address of a two-word buffer which contains the parameter specifications. The first byte is the line count, the second byte is the intensity, and the second word is the Y position. Line count, intensity and Y position must all be octal numbers. The intensity may be any number from 0 to 7, ranging from dimmest to brightest. (If an intensity of 0 is specified, the scroller text will be almost unnoticeable at a BRIGHTNESS knob setting less than one-half). The scroller parameter change is temporary, since an .UNLNK or CTRL/C restores the previous values.

Macro Call: .SCROL baddr

where: baddr is the address of the two-word scroll parameters buffer.

Errors:

No errors are returned. No checking is done on the values of the parameters. A zero argument is interpreted to mean that the parameter value is not to be changed. A negative argument causes the default parameter value to be restored.

Example:

```
        .SCROL  #SCBUF  ;ADJUST SCROLL PARAMETERS
        .
        .
        .
SCBUF:  .BYTE 5         ;DECREASE #LINES TO 5.
        .BYTE 0         ;LEAVE INTENSITY UNCHANGED.
        .WORD 300       ;TOP LINE AT Y=300.
```

A.2.10 .START

The .START request starts the display processor if it was stopped by a .STOP directive. If the display processor is running, it is stopped first, then restarted. In either case, the subpicture stack is cleared and the display processor is started at the top of the handler's internal display file.

Macro Call: .START

Errors:

None.

A.2.11 .STAT

The .STAT request transfers the address of a seven-word status buffer to the display stop interrupt routine in VTBASE. Once the transfer has been made, display processor status data is transferred to the buffer by the display stop interrupt routine in VTBASE whenever a .DSTAT or .DHALT instruction is encountered (see Sections A.3.3 and A.3.4). The transfer is made only when the buffer flag is clear (zero). After the transfer is made, the buffer flag is set non-zero and the .DSTAT or .DHALT instruction is replaced by a .DNOP (Display NOP) instruction.

The status buffer must be a seven-word, contiguous block of memory. Its contents are the same as the light pen status buffer. For a detailed description of the buffer and an explanation of the status words, see Section A.2.5 and Table A-1.

Macro Call: .STAT baddr

where:	baddr	is the address of the status buffer receiving the data.

Errors:

No errors are indicated. If a buffer was previously set up, the new buffer address is replaced as the old buffer address.

A.2.12 .STOP

The .STOP request "stops" the display processor. It actually effects a stop by preventing the DPU from cycling through any user display files. It is useful for stopping the display during modification of a display file, a risky task when the display processor is running. However, a .BLANK could be equally useful for this purpose, since the .BLANK request does not return until the display processor has been removed from the user display file being blanked.

Macro Call: .STOP

Errors:

None.

NOTE

Since the display processor must cycle through the text buffer in the Display Monitor in order for console output to be processed, the text buffer remains visible after a .STOP request is processed, but all user files disappear.

A.2.13 .SYNC/.NOSYN

The .SYNC and .NOSYN requests provide program access to the power line synchronization feature of the display processor. The .SYNC request enables synchronization and the .NOSYN request disables it (the default case).

Synchronization is achieved by stopping the display and restarting it when the power line frequency reaches a trigger point, e.g., a peak or zero-crossing. Synchronization has the effect of fixing the display refresh time. This may be useful in some cases where small amounts of material are displayed but the amount frequently changes, causing changes in intensity. In most cases, however, using synchronization increases flicker.

Macro Calls: .SYNC
.NOSYN

Errors:

None.

A.2.14 .TRACK

The .TRACK request causes the tracking object to appear on the display CRT at the position specified in the request. The tracking object is a diamond-shaped display figure which is light-pen sensitive. If the light pen is placed over the tracking object and then moved, the tracking object follows the light pen, trying to center itself on the pen.

The tracking object first appears at a position specified in a two-word buffer whose address was supplied with the .TRACK request. As the tracking object moves to keep centered on the light pen, the new center position is returned to the buffer. A new set of X and Y values is returned for each light pen interrupt.

The tracking object cannot be lost by moving it off the visible portion of the display CRT. When the edge flag is set, indicating a side of the tracking object is crossing the edge of the display area, the tracking object stops until moved toward the center. To remove the tracking object from the screen, repeat the .TRACK request without arguments.

The .TRACK request may also include the address of a completion routine as the second argument. If a .TRACK completion routine is specified, the light pen interrupt handler passes control to the completion routine at interrupt level. The completion routine is called as a subroutine and the exit statement must be an RTS PC. The completion routine must also preserve any registers it may use.

Macro Call: .TRACK baddr, croutine

where:	baddr	is the address of the two-word buffer containing the X and Y position for the track object.
	croutine	is the address of the completion routine.

Errors:

None.

Example:

See Section A.10.

A.2.15 .UNLNK

The .UNLNK request is used before exiting from a program. In the case where the scroller is present, .UNLNK breaks the link, established by .LNKRT, between the Display File Handler's internal display file and the scroll file in the Display Monitor. The display processor is started cycling in the scroll text buffer, and no further graphics may be done until the link is established again. In the case where no scroller exists, the display processor is simply left stopped.

Macro Call: .UNLNK

Errors:

No errors are returned. An internal link flag is checked to determine if the link exists. If it does not exist, the request is ignored.

A.3 EXTENDED DISPLAY INSTRUCTIONS

The Display File Handler offers the assembly language graphics programmer an extended display processor instruction set, implemented in software through the use of the Load Status Register A (LSRA) instruction. The extended instruction set includes: subroutine call, subroutine return, display status return, display halt, and load name register.

A.3.1 DJSR Subroutine Call Instruction

The DJSR instruction (octal code is 173400) simulates a display subroutine call instruction by using the display stop instruction (LSRA instruction with interrupt bits set). The display stop interrupt handler interprets the non-zero word following the DJSR as the subroutine return address, and the second word following the DJSR as the address of the subroutine to be called. The instruction sequence is:

```
DJSR
Return address
Subroutine address
```

Example:

To call a subroutine SQUARE:

```
POINT           ;POSITION BEAM
100             ;AT (100,100)
100
DJSR            ;THEN CALL SUBROUTINE
.+4
SQUARE          ;TO DRAW A SQUARE
DRET
0
```

The use of the return address preceding the subroutine address offers several advantages. For example, the BASIC-11 graphics software uses the return address to branch around subpicture tag data stored following the subpicture address. This structure is described in Section A.5.3. In addition, a subroutine may be temporarily bypassed by replacing the DJSR code with a DJMP instruction, without the need to stop the display processor to make the by-pass.

The address of the return address is stacked by the display stop interrupt handler on an internal subpicture stack. The stack depth is conditionalized and has a default depth of 10. If the stack bottom is reached, the display stop interrupt handler attempts to protect the system by rejecting additional subroutine calls. In that case, the portions of the display exceeding the legal stack depth will not be displayed.

A.3.2 DRET Subroutine Return Instruction

The DRET instruction provides the means for returning from a display file subroutine. It uses the same octal code as DJSR, but with a single argument of zero. The DRET instruction causes the display stop interrupt handler to pop its subpicture stack and fetch the subroutine return address.

Example:

```
SQUARE: LONGV            ;DRAW A SQUARE
        100!INTX
        0
        0!INTX
        100
        100!INTX!MINUSX
        0
        0!INTX
        100!MINUSX
        DRET             ;RETURN FROM SUBPICTURE
        0
```

A.3.3 DSTAT Display Status Instruction

The DSTAT instruction (octal code is 173420) uses the LSRA instruction to produce a display stop interrupt, causing the display stop interrupt handler to return display status data to a seven-word user status buffer. The status buffer must first have been set up with a .STAT macro call (if not, the DSTAT is ignored and the display is resumed). The first word of the buffer is set non-zero to indicate the transfer has taken place, and the DSTAT is replaced with a DNOP (display NOP). The first word is the buffer flag and the next six words contain name register contents, current subpicture tag, display program counter, display status register, display X register, and display Y register. After transfer of status data, the display is resumed.

A.3.4 DHALT Display Halt Instruction

The DHALT instruction (octal code is 173500) operates similarly to the DSTAT instruction. The difference between the two instructions is that the DHALT instruction leaves the display processor stopped when exiting from the interrupt. A status data transfer takes place provided the buffer was initialized with a .STAT call. If not, the DHALT is ignored.

Example:

```
        .STAT   #SBUF           ;INIT BUFFER
        MOV     #DHALT,STPLOC   ;INSERT DHALT
        .INSRT  #DFILE          ;DISPLAY THE PICTURE
```

```
1$:         TST        SBUF           ;DHALT PROCESSED?
            BEQ        1$             ;NO, WAIT
             .
             .
SBUF:       .BLKW 7                   ;STATUS BUFFER
DFILE:      POINT                     ;POSITION NEAR TOP OF 12" TUBE
            .WORD      500,1350
            LONGV                     ;DRAW A LINE, MAYBE OVER EDGE
            .WORD      0,400          ;IF IT IS A 12" SCOPE.
STPLOC:     DNOP                      ;STATUS WILL BE RETURNED AT TH
            DRET
            0
```

A.3.5 DNAME Load Name Register Instruction

The Display File Handler provides a name register capability through the use of the display stop interrupt. When a DNAME instruction (octal code is 173520) is encountered, a display stop interrupt is generated. The display stop handler stores the argument following the DNAME instruction in an internal software register called the "name register". The current name register contents are returned whenever a DSTAT or DHALT is encountered, and more importantly, whenever a light pen interrupt occurs. The use of a "name" (with a valid range from 1 to 77777) enables the programmer to label each element of the display file with a unique name, permitting the easy identification of the particular display element selected by the light pen.

The name register contents are stacked on a subpicture call and restored on return from the subpicture.

Example:

The SQUARE subroutine with "named" sides.

```
SQUARE:     DNAME                ;NAME IS
            10                   ;10
            LONGV                ;DRAW A SIDE
            100!INTX
            0
            DNAME                ;THIS SIDE IS NAMED
            11                   ;11
            0!INTX               ;STILL IN LONG VECTOR MODE
            100
            DNAME
            12
            100!INTX!MINUSX
            0
            DNAME
            13
            0!INTX
            100!MINUSX
            DRET                 ;RETURN FROM SUBPICTURE
            0
```

A.4 USING THE DISPLAY FILE HANDLER

Graphics programs which intend to use the Display File Handler for display processor management can be written in MACRO assembly language. The display code portions of the program may use the mnemonics described in Section A.7. Calls to the Handler should have the format described in Section A.6.

The Display File Handler is supplied in two pieces, a file of MACRO definitions and a library containing the Display File Handler modules.

MACRO Definition File:	VTMAC.MAC	
Display File Handler:	VTLIB.OBJ	(consisting of:)
		VTBASE.OBJ VTCAL1.OBJ VTCAL2.OBJ VTCAL3.OBJ VTCAL4.OBJ

A.4.1 Assembling Graphics Programs

To assemble a graphics program using the display processor mnemonics or the Display Handler macro calls, the file VTMAC.MAC must be assembled with the program, and must precede the program in the assembler command string.

Example:

Assume PICTUR.MAC is a user graphics program to be assembled. An assembler command string would look like this:

```
MACRO VTMAC+PICTUR/OBJECT
```

A.4.2 Linking Graphics Programs

Once assembled with VTMAC, the graphics program must be linked with the Display File Handler, which is supplied as a single concatenated object module,VTHDLR.OBJ. The Handler may optionally be built as a library, following the directions in A.8.5. The advantage of using the library when linking is that the Linker will select from the library only those modules actually used. Linking with VTHDLR.OBJ results in all modules being included in the link.

To link a user program called PICTUR.OBJ using the concatenated object module supplied with RT-11:

```
LINK PICTUR,VTHDLR
```

To link a program called PICTUR.OBJ using the VTLIB library built by following the directions in A.8.5, be sure to use the Version 03 Linker:

```
LINK PICTUR,VTLIB
```

VTLIB (Handler Modules):

Module	CSECT	Contains	Globals
VTCAL1	$GT1	.CLEAR	$VINIT
		.START	$VSTRT
		.STOP	$VSTOP
		.INSRT	$VNSRT
		.REMOV	$VRMOV
VTCAL2	$GT2	.BLANK	$VBLNK
		.RESTR	$VRSTR
VTCAL3	$GT3	.LPEN	$VLPEN
		.NAME	$NAME
		.STAT	$VSTPM
		.SYNC	$SYNC
		.NOSYN	$NOSYN
		.TRACK	$VTRAK
VTCAL4	$GT4	.LNKRT	$VRTLK
		.UNLNK	$VUNLK
		.SCROL	$VSCRL
VTBASE	$GTB	Interrupt handlers and internal display file.	$DFILE

The five modules in VTHDLR can be used in three different ways. When space is not critical, the most straightforward way is to link VTHDLR directly with a display program. The following command is an example.

```
LINK PICTUR,VTHDLR
```

It is often necessary to conserve space, however, and selective loading of modules is possible by first creating an indexed object module library from VTHDLR and then by making global calls within the display program. The following command creates an indexed object module library.

```
LIBRARY/CREATE VTLIB VTHDLR
```

To further conserve space with overlays, it is also possible to extract individual object modules from a library and create separate object module files. For example, to link a display program using overlays, the following statements are a typical sequence of creating, extracting and linking commands. (NOTE: the modules VTCAL1 and VTCAL2 must be in the same overlay if any global in either one is used.)

```
              .
              .
              .
.LIBRARY/CREATE VTLIB VTHDLR
              .
              .
              .
.LIBRARY/EXTRACT VTLIB VTCAL1
GLOBAL? $VSTRT !moves entire module with $VSTRT to VTCAL1
GLOBAL? !Terminates prompting sequence
.LIBRARY/EXTRACT VTLIB VTCAL2
GLOBAL? $VBLNK !Moves the entire module to VTCAL2
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTCAL3
GLOBAL? $VLPEN !Moves the entire module
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTCAL4
GLOBAL? $VRTLK !Moves the entire module
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTBASE
GLOBAL? $DFILE !Moves the entire module
GLOBAL?
              .
              .
              .
.LINK/PROMPT PICTUR,VTBASE
*VTCAL1,VTCAL2,VTCAL3/O:1
*VTCAL4/O:1
*//
              .
              .
              .
```

A.5 DISPLAY FILE STRUCTURE

The Display File Handler supports a variety of display file structures, takes over the job of display processor management for the programmer, and may be used for both assembly language graphics programming and for systems program development. For example, the Handler supports the tagged subpicture file structure used by the BASIC-11 graphics software, as well as simple file structures. These are discussed in this section.

A.5.1 Subroutine Calls

A subroutine call instruction, with the mnemonic DJSR, is implemented using the display stop (DSTOP) instruction with an interrupt. The display stop interrupt routine in the Display File Handler simulates the DJSR instruction, and this allows great flexibility in choosing the characteristics of the DJSR instruction.

The DJSR instruction stops the display processor and requests an interrupt. The DJSR instruction may be followed by two or more words, and in this implementation the exact number may be varied by the programmer at any time. The basic subroutine call has this form:

```
DJSR
Return Address
Subroutine Address
```

In practice, simple calls to subroutines could look like:

```
DJSR
.WORD     .+4
.WORD     SUB
```

where SUB is the address of the subroutine. Control will return to the display instruction following the last word of the subroutine call. This structure permits a call to the subroutine to be easily by-passed without stopping the display processor, by replacing the DJSR with a display jump (DJMP) instruction:

```
DJMP
.WORD     .+4
.WORD     SUB
```

A more complex display file structure is possible if the return address is generalized:

```
.DJSR
.WORD     NEXT
.WORD     SUB
```

where NEXT is the generalized return address. This is equivalent to the sequence:

```
DJSR
.WORD     .+4
.WORD     SUB
DJMP
.WORD     NEXT
```

It is also possible to store non-graphic data such as tags and pointers in the subroutine call sequence, such as is done in the tagged subpicture display file structure of the BASIC-11 graphics software. This technique looks like:

```
        DJSR
        .WORD     NEXT
        .WORD     SUB
        DATA
NEXT:     .
          .
          .
```

For simple applications where the flexibility of the DJSR instruction described above is not needed and the resultant overhead is not desired, the Display File Handler (VTBASE.MAC and VTCALL.MAC) can be conditionally re-assembled to produce a simple DJSR call. If NOTAG is defined during the assembly, the Handler will be configured to support this simple DJSR call:

```
DJSR
.WORD     SUB
```

where SUB is the address of the subroutine. Defining NOTAG will eliminate the subpicture tag capability, and with it the tracking object, which uses the tag feature to identify itself to the light pen interrupt handler.

Whatever the DJSR format used, all subroutines and the user main file must be terminated with a subroutine return instruction. This is implemented as a display stop instruction (given the mnemonic DRET) with an argument of zero. A subroutine then has the form:

```
SUB: Display Code
     DRET
     .WORD 0
```

A.5.2 Main File/Subroutine Structure

A common method of structuring display files is to have a main file which calls a series of display subroutines. Each subroutine will produce a picture element and may be called many times to build up a picture, producing economy of code. If the following macros are defined:

```
.MACRO    CALL <ARG>
DJSR
.WORD     .+4
.WORD     ARG
.ENDM
.MACRO    RETURN
DRET
.WORD     0
.ENDM
```

then a main file/subroutine file structure would look like:

```
;MAIN DISPLAY FILE
;
MAIN:     Display Code
          CALL SUB1        ;CALL SUBROUTINE 1
          Display Code
          CALL SUB2        ;CALL SUBROUTINE 2
           .               ;ETC
           .
           .
          RETURN
```

```
;
;DISPLAY   SUBROUTINES
;
SUB1:      Display Code    ;SUBROUTINE 1
           RETURN
;
SUB2:      Display Code    ;SUBROUTINE 2
           RETURN
             .             ;ETC.
             .
             .
```

A.5.3 BASIC-11 Graphic Software Subroutine Structure

An example of another method of structuring display files is the tagged subpicture structure used by BASIC-11 graphic software. The display file is divided into distinguishable elements called subpictures, each of which has its own unique tag.

The subpicture is constructed as a subroutine call followed by the subroutine. It is essentially a merger of the main file/subroutine structure into an in-line sequence of calls and subroutines. As such, it facilitates the construction of display files in real time, one of the important advantages of BASIC-11 graphic software.

The following is an example of the subpicture structure. Each subpicture has a call to a subroutine with the return address set to be the address of the next subpicture. The subroutine called may either immediately follow the call, or may be a subroutine defined as part of a subpicture created earlier in the display file. This permits a subroutine to be used by several subpictures without duplication of code. Each subpicture has a tag to identify it, and it is this tag which is returned by the light pen interrupt routine. To facilitate finding subpictures in the display file, they are made into a linked list by inserting a forward pointer to the next tag.

```
SUB1:    DJSR                 ;START OF SUBPICTURE 1
         .WORD    SUB2        ;NEXT SUBPICTURE
         .WORD    SUB1+12     ;JUMP TO THIS SUBPICTURE
         .WORD    1           ;TAG = 1
         .WORD    SUB2+6      ;POINTER TO NEXT TAG

;BODY OF SUBPICTURE 1

         DRET                 ;RETURN FROM
         0                    ;SUBPICTURE 1

SUB2:    DJSR                 ;START SUBPICTURE 2
         .WORD    SUB3        ;NEXT SUBPICTURE
         .WORD    SUB2+12     ;JUMP TO THIS SUBPICTURE
         .WORD    2           ;TAG 2
         .WORD    SUB3+6      ;PTR TO NEXT TAG
```

```
;BODY OF SUBPICTURE 2

          DRET                  ;RETURN FROM
          .WORD     0           ;SUBPICTURE 2

SUB3:     DJSR                  ;START SUBPICTURE 3
          .WORD     SUB4        ;NEXT SUBPICTURE
          .WORD     SUB1+12     ;COPY SUBPICTURE 1
                                ;FOR THIS SUBPICTURE
          .WORD     3           ;BUT TAG IT 3.
          .WORD     SUB4+6      ;PTR TO NEXT TAG

SUB4:     DJSR                  ;START SUBPICTURE 4
          .                     ;ETC.
          .
          .
```

A.6 SUMMARY OF GRAPHICS MACRO CALLS

Mnemonic	Function	MACRO Call (see Note 1)	Assembly Language Expansion (see Note 2)
.BLANK	Temporarily blanks a user display file.	.BLANK faddr	.GLOBL $VBLNK .IF NB, faddr MOV faddr, ^100 .ENDC JSR ^07, $VBLNK
.CLEAR	Initializes handler.	.CLEAR	.GLOBL $VINIT JSR ^07, $VINIT
.INSRT	Inserts a call to user display file in handler's master display file.	.INSRT faddr	.GLOBL $VNSRT .IF NB, faddr MOV faddr, ^00 .ENDC JSR ^07, $VNSRT
.LNKRT	Sets up vectors and links display file handler to RT-11 scroller.	.LNKRT	.GLOBL $VRTLK JSR ^07, $VRTLK
.LPEN	Sets up light pen status buffer.	.LPEN baddr	.GLOBL $VLPEN .IF NB, baddr MOV baddr, ^00 .ENDC JSR ^07, $VLPEN
.NAME	Sets up buffer to receive name register stack contents.	.NAME \baddr	.GLOBL $NAME .IF NB, baddr MOV .BEDDR, ^00 .endc JSR ^07, $NAME
.NOSYN	Disables power line synchronization.	.NOSYN	.GLOBL $NOSYN JSR ^07, $NOSYN

Mnemonic	Function	MACRO Call (see Note 1)	Assembly Language Expansion (see Note 2)
.REMOV	Removes the call to a user display file.	.REMOV faddr	.GLOBL $VRMOV .IF NB, faddr MOV faddr, ^00 .ENDC JSR ^07, $VRMOV
.RESTR	Unblanks the user display file.	.RESTR faddr	.GLOBL $VRSTR IF NB, faddr MOV faddr, ^00 .ENDC JSR ^07, $VRSTR
.SCROL	Adjusts monitor scroller parameters.	.SCROL baddr	.GLOBL $VSCRL .IF NB, baddr MOV baddr, ^00 .ENDC JSR ^07, $VSCRL
.START	Starts the display.	.START	.GLOBL $VSTRT JSR ^07, $VSTRT
.STAT	Sets up status buffer.	.STAT baddr	.GLOBL $VSTPM .IF NB, baddr MOV baddr, ^00 .ENDC JSR ^07, $VSTPM
.STOP	Stops the display.	.STOP	.GLOBL $VSTOP JSR ^07, $VSTOP
.SYNC	Enables power line synchronization.	.SYNC	.GLOBL $SYNC JSR ^07, $SYNC

.TRACK	Enables the track object.	.TRACK baddr, croutine	.GLOBL $VTRAK
			.IF NB, baddr
			MOV baddr, ^00
			.ENDC
			.IF NB, croutine
			MOV croutine,-
			(^06)
			.IFF
			CLR-(^06)
			.ENDC
			.NARG T
			.IF EQ, T
			CLR ^00
			.ENDC
			JSR ^07, $VTRAK

Mnemonic	Function	MACRO Call (see Note 1)	Assembly Language Expansion (see Note 2)
.UNLNK	Unlinks display handler from RT-11 if linked (otherwise leaves display stopped).	.UNLNK	.GLOBL $VUNLK JSR ^07, $VUNLK

NOTE 1

baddr	Address of data buffer.
faddr	Address of start of user display file.
croutine	Address of .TRACK completion routine.

NOTE 2

The lines preceded by a dot will not be assembled. The code they enclose may or may not be assembled depending on the conditionals.

A.7 DISPLAY PROCESSOR MNEMONICS

Mnemonic		Value	Function
CHAR	=	100000	Character Mode
SHORTV	=	104000	Short Vector Mode
LONGV	=	110000	Long Vector Mode
POINT	=	114000	Point Mode
GRAPHX	=	120000	Graphplot X Mode
GRAPHY	=	124000	Graphplot Y Mode
RELATV	=	130000	Relative Point Mode
INT0	=	2000	Intensity 0 (Dim)
INT1	=	2200	Intensity 1
INT2	=	2400	Intensity 2
INT3	=	2600	Intensity 3
INT4	=	3000	Intensity 4
INT5	=	3200	Intensity 5
INT6	=	3400	Intensity 6
INT7	=	3600	Intensity 7 (Bright)
LPOFF	=	100	Light Pen Off
LPON	=	140	Light Pen On
BLKOFF	=	20	Blink Off
BLKON	=	30	Blink On
LINE0	=	4	Solid Line

LINE1	=	5	Long Dash
LINE2	=	6	Short Dash
LINE3	=	7	Dot Dash
DJMP	=	160000	Display Jump
DNOP	=	164000	Display No Operation
STATSA	=	170000	Load Status A Instruction
LPLITE	=	200	Light Pen Hit On
LPDARK	=	300	Light Pen Hit Off
ITAL0	=	40	Italics Off
ITAL1	=	60	Italics On
SYNC	=	4	Halt and Resume Synchronized
STATSB	=	174000	Load Status B Instruction
INCR	=	100	Graphplot Increment

(Vector/Point Mode)

INTX	=	40000	Intensity Vector or Point
MAXX	=	1777	Maximum X Component
MAXY	=	1377	Maximum Y Component
MINUSX	=	20000	Negative X Component
MINUSY	=	20000	Negative Y Component

(Short Vector Mode)

SHIFTX	=	200	
MAXSX	=	17600	Maximum X Component
MAXSY	=	77	Maximum Y Component
MISVX	=	20000	Negative X Component
MISVY	=	100	Negative Y Component

A.8 ASSEMBLY INSTRUCTIONS

A.8.1 General Instructions

All programs can be assembled in 16K, using RT-11 MACRO. All assemblies and all links should be error free. The following conventions are assumed:

1. Default file types are not explicitly typed. These are .MAC for source files, .OBJ for assembler output, and .SAV for Linker output.

2. The default device (DK) is used for all files in the example command strings.

3. Listings and link maps are not generated in the example command strings.

A.8.2 VTBASE

To assemble VTBASE with RT-11 link-up capability:

```
MACRO VTBASE
```

A.8.3 VTCAL1 - VTCAL4

To assemble the modules VTCAL1 through VTCAL4:

```
MACRO VTCAL1,VTCAL2,VTCAL3,VTCAL4
```

A.8.4 VTHDLR

To create the concatenated handler module:

```
COPY/BINARY  VTCAL1.OBJ,VTCAL2.OBJ,VTCAL3.OBJ,-
VTCAL4.OBJ,VTBASE.OBJ  VTHDLR.OBJ
```

A.8.5 Building VTLIB.OBJ

To build the VTLIB library:

```
LIBRARY/CREATE VTLIB VTHDLR
```

A.9 VTMAC

```
        .TITLE  VTMAC
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
; OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (C) 1978, DIGITAL EQUIPMENT CORPORATION.
;
; VTMAC IS A LIBRARY OF MACRO CALLS AND MNEMONIC DEFINITIONS WHICH
; PROVIDE SUPPORT OF THE VT11 DISPLAY PROCESSOR. THE MACROS PRODUCE
; CALLS TO THE VT11 DEVICE SUPPORT PACKAGE, USING GLOBAL REFERENCES.

; MACRO TO GENERATE A MACRO WITH ZERO ARGUMENTS.
```

```
.MACRO MAC0 NAME,CALL
        .MACRO NAME
        .GLOBL  CALL
        JSR     PC,CALL
        .ENDM
.ENDM

; MACRO TO GENERATE A MACRO WITH ONE ARGUMENT

.MACRO MAC1 NAME,CALL
        .MACRO NAME ARG
        .IF NB,ARG
        MOV     ARG,%^O0
        .ENDC
        .GLOBL  CALL
        JSR     PC,CALL
        .ENDM
.ENDM

; MACRO TO GENERATE A MACRO WITH TWO OPTIONAL ARGUMENTS

.MACRO MAC2 NAME,CALL
        .MACRO NAME ARG1,ARG2
        .GLOBL  CALL
        .IF NB,ARG1
        MOV     ARG1,%^O0
        .ENDC
        .IF NB,ARG2
        MOV     ARG2,-(SP)
        .IFF
        CLR     -(SP)
        .NARG   T
        .IF EQ,T
        CLR     %^O0
        .ENDC
        .ENDC
        JSR     PC,CALL
        .ENDM
.ENDM

; MACRO LIBRARY FOR VT11:

MAC0    <.CLEAR>,<$VINIT>
MAC0    <.STOP>,<$VSTOP>
MAC0    <.START>,<$VSTRT>
MAC1    <.INSRT>,<$VNSRT>
MAC1    <.REMOV>,<$VRMOV>
MAC1    <.BLANK>,<$VBLNK>
MAC1    <.RESTR>,<$VRSTR>
MAC1    <.STAT>,<$VSTPM>
MAC1    <.LPEN>,<$VLPEN>
MAC1    <.SCROL>,<$VSCRL>
MAC2    <.TRACK>,<$VTRAK>
MAC0    <.LNKRT>,<$VRTLK>
MAC0    <.UNLNK>,<$VUNLK>
```

```
; MNEMONIC DEFINITIONS FOR THE VT11 DISPLAY PROCESSOR

DJMP=160000     ;DISPLAY JUMP
DNOP=164000     ;DISPLAY NOP
DJSR=173400     ;DISPLAY SUBROUTINE CALL
DRET=173400     ;DISPLAY SUBROUTINE RETURN
DNAME=173520    ;SET NAME REGISTER
DSTAT=173420    ;RETURN STATUS DATA
DHALT=173500    ;STOP DISPLAY AND RETURN STATUS DATA

CHAR=100000     ;CHARACTER MODE
SHORTV=104000   ;SHORT VECTOR MODE
LONGV=110000    ;LONG VECTOR MODE
POINT=114000    ;POINT MODE
GRAPHX=120000   ;GRAPH X MODE
GRAPHY=124000   ;GRAPH Y MODE
RELATV=130000   ;RELATIVE VECTOR MODE

INT0=2000       ;INTENSITY 0
INT1=2200
INT2=2400
INT3=2600
INT4=3000
INT5=3200
INT6=3400
INT7=3600

LPOFF=100       ;LIGHT PEN OFF
LPON=140        ;LIGHT PEN ON
BLKOFF=20       ;BLINK OFF
BLKON=30        ;BLINK ON
LINE0=4         ;SOLID LINE
LINE1=5         ;LONG DASH
LINE2=6         ;SHORT DASH
LINE3=7         ;DOT DASH

STATSA=170000   ;LOAD STATUS REG A
LPLITE=200      ;INTENSIFY ON LPEN HIT
LPDARK=300      ;DON'T INTENSIFY
ITAL0=40        ;ITALICS OFF
ITAL1=60        ;ITALICS ON
SYNC=4          ;POWER LINE SYNC

STATSB=174000   ;LOAD STATUS REG B
INCR=100        ;GRAPH PLOT INCREMENT
INTX=40000      ;INTENSIFY VECTOR OR POINT
MAXX=1777       ;MAXIMUM X INCR. - LONGV
MAXY=1377       ;MAXIMUM Y INCR. - LONGV
MINUSX=20000    ;NEGATIVE X INCREMENT
MINUSY=20000    ;NEGATIVE Y INCREMENT
MAXSX=17600     ;MAXIMUM X INCR. - SHORTV
MAXSY=77        ;MAXIMUM Y INCR. - SHORTV
MISVX=20000     ;NEGATIVE X INCR. - SHORTV
MISVY=100       ;NEGATIVE Y INCR. - SHORTV
```

A.10 EXAMPLES USING GTON

```
EXAMPLE #1        MACRO X03.04 18-MAY-77 14:49:44 PAGE 5

                                              .TITLE  EXAMPLE #1
                                      ;
                                      ; THIS EXAMPLE USES THE .LPEN STATUS BUFFER AND THE
                                      ; NAME REGISTER TO MODIFY A DISPLAY FILE WITH THE LIGHT PEN.
                                      ;
          000000                              R0=%0
          000001                              R1=%1
          000007                              PC=%7
          000044                              JSW=44                       ;JOB STATUS WORD

                                              .MCALL  .TTINR,.EXIT,.PRINT
   000000                             START:  .LNKRT                       ;LINK TO MONITOR
   000004 100004                              BPL     1$                   ;LINK UP ERROR?
   000006                                     .PRINT  #EMSG                ;YES, PRINT MESSAGE
   000014                                     .EXIT                        ;AND EXIT.
   000016                             1$:     .SCROL  #SCBUF               ;ADJUST SCROLL
   000026                                     .PRINT  #MSG
   000034                                     .INSRT  #DFILE               ;INSERT DISPLAY FILE
   000044                                     .LPEN   #LBUF                ;SET UP LPEN BUFFER
   000054 052737  000100  000044              BIS     #100,@#JSW           ;SET JSW FOR TTINR
   000062 005767  000070          LTST:       TST     LBUF                 ;LIGHT PEN HIT?
   000066 001003                              BNE     1$                   ;YES
   000070                                     .TTINR                       ;NO, ANY TT INPUT?
   000072 103023                              BCC     EXIT                 ;YES, EXIT
   000074 000772                              BR      LTST                 ;NO, LOOP AGAIN
   000076 016777  000074  000102  1$:         MOV     I2,@IPTR             ;RESTORE PREVIOUS CODE
   000104 016701  000050                      MOV     LBUF+2,R1            ;GET NAME VALUE
   000110 005301                              DEC     R1                   ;SUBTRACT ONE
   000112 006301                              ASL     R1                   ;MULTIPLY BY TWO
   000114 060701                              ADD     PC,R1                ;USE TO INDEX
   000116 062701  000062                      ADD     #DTABL-.,R1          ;OFF TABLE DTABL.
   000122 011167  000060                      MOV     (R1),IPTR            ;MOVE ADDR INTO IPTR
   000126 016777  000042  000052              MOV     I1,@IPTR             ;MODIFY THAT CODE
   000134 005067  000016                      CLR     LBUF                 ;CLEAR BUFFER FLAG TO
                                                                           ;ENABLE ANOTHER LP HIT.
   000140 000750                              BR      LTST                 ;LOOP AGAIN
   000142 022700  000012          EXIT:       CMP     #12,R0               ;LINE FEED?
   000146 001345                              BNE     LTST                 ;NO, GET ANOTHER
   000150                                     .UNLNK                       ;UNLINK FROM MONITOR
   000154                                     .EXIT
   000156                         LBUF:       .BLKW   7                    ;LPEN STATUS BUFFER
   000174 103370                  I1:         .WORD   CHAR!INT5!BLKON!LPON
   000176 103160                  I2:         .WORD   CHAR!INT4!BLKOFF!LPON
   000200 000252' 000272' 000312' DTABL:      .WORD   D1,D2,D3             ;TABLE OF DISPLAY FILE
                                                                           ;LOCATIONS TO BE MODIFIED
   000206 000252'                 IPTR:       .WORD   D1                   ;PREVIOUS LOCATION MODIFIED
   000210 000002                  SCBUF:      .WORD   2                    ;SCROLL LINE COUNT
   000212 001000                              .WORD   1000                 ;SCROLL TOP Y POS.
   000214    041     105     122  EMSG:       .ASCIZ  /!ERROR!/            ;ERROR MESSAGE
   000217    122     117     122
   000222    041     000
                                  .EVEN
   000224    105     130     101  MSG:        .ASCIZ  /EXAMPLE #1/         ;I.D. MESSAGE
   000227    115     120     114
   000232    105     040     043
   000235    061     000
                                              .EVEN
```

```
EXAMPLE #1        MACRO X03.04 18-MAY-77 14:49:44 PAGE 5-1

                                      ;
                                      ; DISPLAY FILE FOR EXAMPLE #1
                                      ;
   000240 114000                  DFILE:      POINT
   000242 000100                              100
   000244 000500                              500
   000246 173520                              DNAME
   000250 000001                              1
   000252 103160                  D1:         CHAR!BLKOFF!INT4!LPON
   000254    117     116     105              .ASCII  /ONE./
   000257    056
   000260 114000                              POINT
   000262 000100                              100
   000264 000300                              300
   000266 173520                              DNAME
   000270 000002                              2
   000272 103160                  D2:         CHAR!BLKOFF!INT4!LPON
   000274    124     127     117              .ASCII  /TWO./
   000277    056
   000300 114000                              POINT
   000302 000100                              100
   000304 000100                              100
   000306 173520                              DNAME
   000310 000003                              3
   000312 103160                  D3:         CHAR!BLKOFF!INT4!LPON
   000314    124     110     122              .ASCII  /THREE./
   000317    105     105     056
   000322 173400                              DRET
   000324 000000                              0
          000000'                             .END    START
```

```
EXAMPLE #1      MACRO X03.04 18-MAY-77 14:49:44 PAGE 5-2
SYMBOL TABLE

D2        000272R     INTX   = 040000    INT4   = 003000    ITAL1  = 000060    DFILE    000240R
EXIT      000142R     INT2   = 002400    LPON   = 000140    ITBT     000062R   INT7   = 003600
INT0    = 002000      LINE1  = 000005    MINUSX= 020000     INT5   = 003200    MAXY   = 001377
MAXSX   = 017600      DJMP   = 160000    MINUSY= 020000     T2       000170R   SHORTV= 104000
D3        000312R     LONGV  = 110000    POINT  = 114000    IPTR     000220R   D1       000252R
MAXSY   = 000077      LPDARK= 000300     EMSG     000214R   DSTAT  = 173424    STATSA= 170000
MISVY   = 000100      LINE2  = 000006    ITAL0  = 000040    SCBUF    000210R   STATSB= 174000
MSG       000224R     INT3   = 002600    T1       000174R   SYNC   = 000004    $VSCPL= ****** G
INT1    = 002200      RELATV= 130000     $VLPEN= ****** G   INT6   = 003400    GRAPHX= 120000
BLKON   = 000030      DRET   = 173400    BLKOFF= 000020     JSW    = 000044    GRAPHY= 124000
DHALT   = 173500      LINE3  = 000007    DJSR   = 173400    MAXX   = 001777    $VNSRT= ****** G
LINE0   = 000004      LBUF     000156R   $VRTLK= ****** G   START    000000R   LPOFF  = 000100
CHAR    = 100000      INCR   = 000100    DNAME  = 173520    $VINLK= ****** G   MISVX  = 020000
DTABL     000200R     LPLITE= 000200     DNOP   = 164000

. ABS.   000000    000
         000326    001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  3564 WORDS  ( 14 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  64 PAGES
,LP:=VTMAC,MANEX1
```

```
EXAMPLE #2      MACRO X03.04 18-MAY-77 14:49:57 PAGE 5

                                            .TITLE  EXAMPLE #2
                                    ;
                                    ; THIS EXAMPLE USES THE TRACKING OBJECT AND THE TRACK
                                    ; COMPLETION ROUTINE TO CAUSE A VECTOR TO FOLLOW
                                    ; THE LIGHT PEN FROM A SET POINT AT (500,500).
                                    ;
           000000                           R0=%0
           000001                           R1=%1
           000006                           SP=%6
           000007                           PC=%7
                                            .MCALL  .EXIT,.TTYIN,.PRINT
   000000                           START:  .LNKRT                  ;LINK TO MONITOR
   000004  100004                           BPL     1$              ;LINK UP ERROR?
   000006                                   .PRINT  #EMSG           ;YES, INFORM USER
   000014                                   .EXIT                   ;AND EXIT
   000016                           1$:     .INSRT  #DFILE          ;INSERT DISPLAY FILE
   000026                                   .TRACK  #TBUF,#TCOM     ;DISPLAY TRACK OBJECT
   000042  004767  000006                   JSR     PC,WAIT         ;WAIT FOR <CR>
   000046                                   .UNLNK                  ;UNLINK FROM MONITOR
   000052                                   .EXIT
   000054                           WAIT:   .TTYIN                  ;GET CHAR. FROM TTY
   000060  022700  000012                   CMP     #12,R0          ;LINE FEED?
   000064  001373                           BNE     WAIT            ;NO, GET ANOTHER
   000066  000207                           RTS     PC
   000070  000500  000500           TBUF:   .WORD   500,500         ;TRACK BUFFER INITED TO
                                                                    ;START TRACK AT (500,500)
                                    ;
                                    ; TRACK COMPLETION ROUTINE ENTERED AT INTERRUPT LEVEL
                                    ; FROM DISPLAY FILE HANDLER WITH DISPLAY STOPPED.
                                    ; USED TO UPDATE DISPLAY FILE WITH DATA FROM TBUF.
                                    ;
   000074  010146                   TCOM:   MOV     R1,-(SP)        ;SAVE R1
   000076  016701  177766                   MOV     TBUF,R1         ;NEW X
   000102  166701  000052                   SUB     DX,R1           ;NEW X - OLD X
   000106  100003                           BPL     1$              ;POSITIVE DIFFERENCE?
   000110  005401                           NEG     R1              ;NO, SO MAKE POSITIVE
   000112  052701  020000                   BIS     #MINUSX,R1      ;BUT SET MINUS BIT
   000116  052701  040000           1$:     BIS     #INTX,R1        ;ALSO SET INTENSIFY BIT
   000122  010167  000040                   MOV     R1,DX           ;THEN STORE IN DFILE.
   000126  016701  177740                   MOV     TBUF+2,R1       ;NEW Y
   000132  166701  000024                   SUB     DY,R1           ;NEW Y - OLD Y
   000136  100003                           BPL     2$              ;POSITIVE DIFFERENCE?
   000140  005401                           NEG     R1              ;NO,SO MAKE POSITIVE
   000142  052701  020000                   BIS     #MINUSY,R1      ;AND SET MINUS BIT
   000146  010167  000016           2$:     MOV     R1,DY           ;THEN STORE IN DFILE
   000152  012601                           MOV     (SP)+,R1        ;RESTORE R1
   000154  000207                           RTS     PC              ;EXIT FROM COMPLETION ROUTINE
                                    ;
                                    ; DISPLAY FILE FOR EXAMPLE #2
                                    ;
   000156  114000                   DFILE:  POINT                   ;SET POINT AT
   000160  000500                   DX:     500
   000162  000500                   DY:     500                     ;(500,500)
   000164  113000                           LONGV!INT4              ;DRAW A VECTOR
   000166  000000                   DX:     .WORD   0               ;INITIALLY NOWHERE
   000170  000000                   DY:     .WORD   0
   000172  173400                           DRET                    ;DISPLAY FILE END
```

```
EXAMPLE #2      MACRO X03.04 18-MAY-77 14:49:57 PAGE 5-1

        000174  000000                                  M
        000176     123      117      122  EMSG:   .ASCIZ  /SORRY, THERE SEEMS TO BE A PROBLEM/
        000201     122      131      054
        000204     040      124      110
        000207     105      122      105
        000212     040      123      105
        000215     105      115      123
        000220     040      124      117
        000223     040      102      105
        000226     040      101      040
        000231     120      122      117
        000234     102      114      105
        000237     115      000
                                          .EVEN
                000000'                           .END    START
```

```
EXAMPLE #2      MACRO X03.04 18-MAY-77 14:49:57 PAGE 5-2
SYMBOL TABLE

INT0  = 002000        LONGV = 110000        LPLITE= 000200        SVOILK= ****** G      SVUNLK= ****** G
MAXSX = 017600        LPDARK= 000300        WAIT    000054R       DNAME = 173520        DFILE   000150R
MAXSY = 000077        LINE2 = 000006        SYTRAK= ****** G      DNOP  = 164000        INT7  = 003600
MISVY = 000100        DX      000166R       INT4  = 003000        ITAL1 = 000060        MAXY  = 001377
INT1  = 002200        INT3  = 002600        LPON  = 000140        INT5  = 003200        SHORTV= 104000
BLKON = 000030        RELATV= 130000        MINUSX= 020000        DSTAT = 173420        STATSA= 170000

DHALT = 173500        TCOM    000074R       MINUSY= 020000        SYNC  = 000004        STATSB= 174000
LINE0 = 000004        DRET  = 173400        POINT = 114000        INT6  = 003400        GRAPHX= 120000
CHAR  = 100000        LINE3 = 000007        EMSG    000176R       MAYX  = 001777        GRAPHY= 124000
INTX  = 040000        DY      000170R       ITAL0 = 000040        OX      000160R       SVNSRT= ****** G
INT2  = 002400        TBUF    000070R       BLKOFF= 000020        START   000000R       LPOFF = 000100
LINE1 = 000005        INCR  = 000100        DJSR  = 173400        OY      000162R       MISVX = 020000
DJMP  = 160000

. ABS.  000000    000
        000242    001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  3717 WORDS  ( 15 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  64 PAGES
,LP:=VTMAC,MANEX2
```

APPENDIX B

SYSTEM MACRO LIBRARY

The following is a listing of the system macro library (SYSMAC.SML) for the RT-11 V03B release. This library is stored on the system device and is used by MACRO when it expands the programmed requests discussed in Chapter 2.

```
; SYSMAC.MAC--SYSTEM MACRO LIBRARY
;
; RT-11 VERSION 3B
;
; COPYRIGHT (C) 1977, 1978
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS. 01754
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A
; SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY 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 EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO
; AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP OF THE
; SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT
; NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
; EQUIPMENT CORPORATION.
;
; DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.

.MACRO ..V1..
.MCALL  ...CM0,...CM1,...CM2,...CM3,...CM4,...CM5,...CM6
...V1=1
.ENDM

.MACRO ..V2..
.MCALL  ...CM0,...CM1,...CM2,...CM3,...CM4,...CM5,...CM6
...V1=2.
.ENDM

.MACRO .MACS
.MCALL  ...CM0,...CM1,...CM2,...CM3,...CM4,...CM5,...CM6
...V1=3.
.ENDM
```

```
.MACRO ...CM0 STARG
.IF B <STARG>
        CLR     -(6.)
.IFF
.IF IDN <STARG>,#0
        CLR     -(6.)
.IFF
        MOV     STARG,-(6.)
.ENDC
.ENDC
.ENDM

.MACRO ...CM1   AREA,IC,CHAN,FLAG
...CM5  <AREA>
...V2=0
.IF B <FLAG>
.IIF B <AREA>, ...V2=1
.IFF
.IIF DIF <FLAG>,SET, ...V2=1
.ENDC
.IF NE ...V2
.IF IDN <CHAN>,<#0>
        CLRB    (0)
.IFF
.IF NB <CHAN>
        MOVB    CHAN,(0)
.ENDC
.ENDC
.IFF
.IF B <CHAN>
        MOVB    #IC,1(0)
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^027
        MOV     CHAN+<IC*^0400>,(0)
.IFF
        MOV     #IC*^0400,(0)
        MOVB    CHAN,(0)
.ENDC
.ENDC
.ENDC
.ENDM

.MACRO ...CM2   ARG,OFFSE,INS,CSET,BB
.IF B <ARG>
.IF NB <CSET>
.IF NE ...V1-3.
        CLR'BB  OFFSE(0)
.ENDC
.ENDC
.IFF
.IF IDN <ARG>,#0
        CLR'BB  OFFSE(0)
.IFF
        MOV'BB  ARG,OFFSE(0)
.ENDC
.ENDC
.IF NB <INS>
        EMT     ^0375
.ENDC
.ENDM

.MACRO ...CM3   CHAN,IC
.IF B <CHAN>
        MOV     #IC*^0400,%0
```

```
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^027
        MOV     CHAN+<IC*^0400>,%0
.IFF
        MOV     #IC*^0400,%0
        BISB    CHAN,%0
.ENDC
.ENDC
        EMT     ^0374
.ENDM

.MACRO ...CM4   AREA,CHAN,BUF,WCNT,BLK,CRTN,IC,CODE
...CM1 <AREA>,<IC>,<CHAN>,<CODE>
...CM2 <BLK>,2.
...CM2 <BUF>,4.
...CM2 <WCNT>,6.
...CM2 <CRTN>,8.,X
.ENDM

.MACRO ...CM5   SRC,BB
.IF NB <SRC>
.IF DIF <SRC>,R0
        MOV'BB  SRC,%0
.ENDC
.ENDC
.ENDM

.MACRO ...CM6   AREA,IC,CHAN,FLAG
...CM5  <AREA>
.IF B <FLAG>
.IF NB <AREA>
        MOV     #IC*^0400+CHAN,(0)
.ENDC
.IFF
.IF IDN <FLAG>,SET
        MOV     #IC*^0400+CHAN,(0)
.ENDC
.ENDC
.ENDM

.MACRO .CDFN    AREA,ADDR,NUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,13.,0,<CODE>
...CM2  <ADDR>,2.
...CM2  <NUM>,4.,X
.ENDM

.MACRO .CHAIN
        MOV     #8.*^0400,%0
        EMT     ^0374
.ENDM

.MACRO .CHCOP   AREA,CHAN,OCHAN,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM1  <AREA>,11.,<CHAN>,<CODE>
...CM2  <OCHAN>,2.,X
.ENDM
```

```
.MACRO .CLOSE   CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
        EMT     ^O<160+CHAN>
.IFF
...CM3 <CHAN>,6.
.ENDC
.ENDM

.MACRO .CNTXS   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,27.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .CMKT    AREA,ID,TIME,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,19.,0,<CODE>
...CM2  <ID>,2.
...CM2  <TIME>,4.,X,X
.ENDM

.MACRO .CRAW    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,2.,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .CRRG    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .CSIGE   DEVSPC,DEFEXT,CSTRNG,LINBUF
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF NB <LINBUF>
...CM0  <LINBUF>
.NTYPE  ...V2,DEVSPC
.IF EQ ...V2-^O27
...CM0  <DEVSPC'+1>
.IFF
...CM0  <DEVSPC>
        INC     (6.)
.ENDC
.IFF
```

```
...CM0  <DEVSPC>
.ENDC
...CM0  <DEFEXT>
...CM0  <CSTRNG>
        EMT     ^O344
.ENDM

.MACRO .CSISP   OUTSPC,DEFEXT,CSTRNG,LINBUF
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF NB <LINBUF>
...CM0  <LINBUF>
.NTYPE  ...V2,OUTSPC
.IF EQ ...V2-^O27
...CM0  <OUTSPC'+1>
.IFF
...CM0  <OUTSPC>
        INC     (6.)
.ENDC
.IFF
...CM0  <OUTSPC>
.ENDC
...CM0  <DEFEXT>
...CM0  <CSTRNG>
        EMT     ^O345
.ENDM

.MACRO .CSTAT   AREA,CHAN,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM1  <AREA>,23.,<CHAN>,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .CTIMI   TBK
        JSR     %5,@$TIMIT
        .WORD   TBK-.
        .WORD   1
.ENDM

.MACRO .DATE
        MOV     #10.*^O400,%0
        EMT     ^O374
.ENDM

.MACRO .DELET   AREA,CHAN,DBLK,SEQNUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
        EMT     ^O<AREA>
.IFF
...CM5  <AREA>
.IF IDN <CHAN>,#0
        CLR     (0)
.IFF
...V2=0
.IF B <CODE>
```

```
.IIF B <AREA>, ...V2=1
.IFF
.IIF DIF <CODE>,SET, ...V2=1
.ENDC
.IF NE ...V2
.IF NB <CHAN>
        MOVB    CHAN,(0)
.ENDC
.IFF
.IF B <CHAN>
        CLRB    1(0)
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^O27
        MOV     CHAN,(0)
.IFF
        CLR     (0)
        MOVB    CHAN,(0)
.ENDC
.ENDC
.ENDC
.ENDC
...CM2  <DBLK>,2.
...CM2  <SEQNUM>,4.,X,X
.ENDC
.ENDM

.MACRO .DEVIC   AREA,ADDR,LINK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF B LINK
...CM6  <AREA>,12.,0,<CODE>
.IFF
...CM6  <AREA>,12.,1,<CODE>
.ENDC
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .DRAST   NAME,PRI,ABT
.GLOBL  $INPTR
.IIF B <ABT>    RTS     %7
.IIF NB <ABT>   BR      ABT
NAME'INT:: JSR  %5,@$INPTR
        .WORD   ^C<PRI*^O40>&^O340
.ENDM

.MACRO .DRBEG   NAME,VEC,DSIZ,DSTS,VTBL
.IF NDF $SYSDV
.ASECT
. = 52
.GLOBL  NAME'END
        .WORD   <NAME'END - NAME'STRT>
        .WORD   DSIZ
        .WORD   DSTS
.PSECT
.IFF
$SYDSZ == DSIZ
.PSECT  SYSHND
.ENDC
NAME'STRT::
.IF B VTBL
```

```
.GLOBL  NAME'INT
        .WORD   VEC
        .WORD   NAME'INT - .
.IFF
.GLOBL  VTBL,NAME'INT
        .WORD   <VTBL-.>/2. -1 + ^O100000
        .WORD   NAME'INT - .
.ENDC
        .WORD   ^O340
NAME'SYS::
NAME'LQE::      .WORD   0
NAME'CQE::      .WORD   0
.ENDM

.MACRO .DREND   NAME
...V2=0
.IF NE MMG$T
...V2=...V2+2.
.IF DF $SYSDV
.GLOBL $RELOC,$MPPHY,$GETBYT,$PUTBYT,$PUTWRD
$RLPTR:: .WORD  $RELOC
$MPPTR:: .WORD  $MPPHY
$GTBYT:: .WORD  $GETBYT
$PTBYT:: .WORD  $PUTBYT
$PTWRD:: .WORD  $PUTWRD
.IFF
$RLPTR:: .WORD  0
$MPPTR:: .WORD  0
$GTBYT:: .WORD  0
$PTBYT:: .WORD  0
$PTWRD:: .WORD  0
.ENDC
.ENDC
.IF NE ERL$G
...V2=...V2+1
.IF DF $SYSDV
.GLOBL  $ERLOG
$ELPTR:: .WORD  $ERLOG
.IFF
$ELPTR:: .WORD  0
.ENDC
.ENDC
.IF NE TIM$IT
...V2=...V2+4.
.IF DF $SYSDV
.GLOBL  $TIMIO
$TIMIT:: .WORD  $TIMIO
.IFF
$TIMIT:: .WORD  0
.ENDC
.ENDC
.IF DF $SYSDV
.GLOBL $FORK,$INTEN
$INPTR:: .WORD  $INTEN
$FKPTR:: .WORD  $FORK
.IFF
$INPTR:: .WORD  0
$FKPTR:: .WORD  0
.IFTF
.GLOBL  NAME'STRT
NAME'END == .
.IFT
$SYHSZ == NAME'END - NAME'STRT
.IFF
```

```
.ASECT
.=60
         .WORD    ...V2
.PSECT
.ENDC
.ENDM

.MACRO .DRFIN   NAME
.GLOBL  NAME'CQE
        MOV      %7,%4
        ADD      #NAME'CQE-.,%4
        MOV      @#^054,%5
        JMP      @^0270(5)
.ENDM

.MACRO .DSTAT   RETSPC,DNAM
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <DNAM>
...CM0  <RETSPC>
        EMT      ^0342
.ENDM

.MACRO .ELAW    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,3.,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .ELRG    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,1,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .ENTER   AREA,CHAN,DBLK,LEN,SEQNUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
...CM0  <DBLK>
        EMT      ^O<40+AREA>
.IFF
...CM1  <AREA>,2.,<CHAN>,<CODE>
...CM2  <DBLK>,2.
...CM2  <LEN>,4.,,X
...CM2  <SEQNUM>,6.,X,X
.ENDC
.ENDM

.MACRO .EXIT
        EMT      ^0350
.ENDM
```

```
.MACRO .FETCH   ADDR,DNAM
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <DNAM>
...CM0  <ADDR>
        EMT     ^O343
.ENDM

.MACRO .FORK    FKBLK
        JSR     %5,@$FKPTR
        .WORD   FKBLK - .
.ENDM

.MACRO .GMCX    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,6.,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .GTIM    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,17.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .GTJB    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,16.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .GTLIN   LINBUF,PROMPT
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM0  <LINBUF>
...CM0  #1
...CM0  <PROMPT>
        CLR     -(6.)
        EMT     ^O345
.ENDM

.MACRO .GVAL    AREA,OFFSE,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,28.,0,<CODE>
...CM2  <OFFSE>,2.,X
.ENDM
```

```
.MACRO .HERR
        MOV     #5.*^O400,%0
        EMT     ^O374
.ENDM

.MACRO .HRESE
        EMT     ^O357
.ENDM

.MACRO .INTEN   PRIO,PIC
.IF B PIC
        JSR     5.,@^O54
.IFF
        MOV     @#^O54,-(6.)
        JSR     5.,@(6.)+
.ENDC
        .WORD   ^C<PRIO*32.>&224.
.ENDM

.MACRO .LOCK
        EMT     ^O346
.ENDM

.MACRO .LOOKU   AREA,CHAN,DBLK,SEQNUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
        EMT     ^O<20+AREA>
.IFF
...CM1  <AREA>,1,<CHAN>,<CODE>
...CM2  <DBLK>,2.
...CM2  <SEQNUM>,4.,X,X
.ENDC
.ENDM

.MACRO .MAP     AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,4.,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .MTATC   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31.,5.,<CODE>
...CM2  <ADDR>,2.
...CM2  <UNIT>,4.,X,,B
.ENDM

.MACRO .MTDTC   AREA,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31.,6.,<CODE>
...CM2  <UNIT>,4.,X
.ENDM
```

```
.MACRO .MTPRN   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  AREA,31.,7.,<CODE>
...CM2  ADDR,2.
...CM2  <UNIT>,4.,X,,B
.ENDM

.MACRO .MFPS    ADDR
        MOV     @#^054,-(6.)
        ADD     #^0362,(6.)
        JSR     7.,@(6.)+
.IIF NB <ADDR>  MOVB    (6.)+,ADDR
.ENDM

.MACRO .MTRCT   AREA,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31.,4.,<CODE>
...CM2  <UNIT>,4.,X
.ENDM

.MACRO .MRKT    AREA,TIME,CRTN,ID,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,18.,0,<CODE>
...CM2  <TIME>,2.
...CM2  <CRTN>,4.
...CM2  <ID>,6.,X
.ENDM

.MACRO .MTGET   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  AREA,31.,1,<CODE>
...CM2  ADDR,2.
...CM2  <UNIT>,4.,X,,B
.ENDM

.MACRO .MTPS    ADDR
.IIF NB <ADDR>  CLR     -(6.)
.IIF NB <ADDR>  MOVB    ADDR,(6.)
        MOV     @#^054,-(6.)
        ADD     #^0360,(6.)
        JSR     7.,@(6.)+
.ENDM

.MACRO .MTSET   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  AREA,31.,0,<CODE>
...CM2  ADDR,2.
...CM2  <UNIT>,4.,X,,B
.ENDM
```

```
.MACRO .MTIN    AREA,ADDR,UNIT,CHRCNT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  AREA,31.,2.,<CODE>
...CM2  ADDR,2.
...CM2  <UNIT>,4.,,,B
...CM2  <CHRCNT>,5.,X,,B
.ENDM

.MACRO .MTOUT   AREA,ADDR,UNIT,CHRCNT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  AREA,31.,3.,<CODE>
...CM2  ADDR,2.
...CM2  <UNIT>,4.,,,B
...CM2  <CHRCNT>,5.,X,,B
.ENDM

.MACRO .MWAIT
        MOV     #9.*^O400,%0
        EMT     ^O374
.ENDM

.MACRO .PRINT   ADDR
.IF NB <ADDR>
.IF DIF <ADDR>,R0
        MOV     ADDR,%0
.ENDC
.ENDC
        EMT     ^O351
.ENDM

.MACRO .PROTE   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,25.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .PURGE   CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM3  <CHAN>,3.
.ENDM

.MACRO .QELDF
Q.LINK=0
Q.CSW=2.
Q.BLKN=4.
Q.FUNC=6.
Q.JNUM=7.
Q.UNIT=7.
Q.BUFF=^O10
Q.WCNT=^O12
Q.COMP=^O14
.IF EQ MMG$T
```

```
Q.ELGH=^O16
.IFF
Q.PAR=^O16
Q.ELGH=^O24
.ENDC
.ENDM

.MACRO .QSET    ADDR,LEN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <LEN>,B
...CM0  <ADDR>
        EMT     ^O353
.ENDM

.MACRO .RCTRL
        EMT     ^O355
.ENDM

.MACRO .RCVD    AREA,BUF,WCNT,CRTN=#1,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.ENDM

.MACRO .RCVDC   AREA,BUF,WCNT,CRTN,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.ENDM

.MACRO .RCVDW   AREA,BUF,WCNT,CRTN=#0,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,22.,<CODE>
.ENDM

.MACRO .RDBBK   RGSIZ
.MCALL  .RDBDF
.RDBDF
        .WORD
        .WORD   RGSIZ
        .WORD
.ENDM

.MACRO .RDBDF
R.GID   =0
R.GSIZ  =2.
R.GSTS  =4.
R.GLGH  =6.
RS.CRR  =^O100000
RS.UNM  =^O40000
RS.NAL  =^O20000
.ENDM
```

```
.MACRO .READ    AREA,CHAN,BUF,WCNT,BLK,CRTN=#1,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <WCNT>
...CM0  #1
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<200+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8.,<CODE>
.ENDC
.ENDM

.MACRO .READC   AREA,CHAN,BUF,WCNT,CRTN,BLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CRTN>
...CM0  <WCNT>
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<200+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8.,<CODE>
.ENDC
.ENDM

.MACRO .READW   AREA,CHAN,BUF,WCNT,BLK,CRTN=#0,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <WCNT>
...CM0
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<200+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8.,<CODE>
.ENDC
.ENDM

.MACRO .REGDEF
.ENDM

.MACRO .RELEA   DNAM
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <DNAM>
...CM0
        EMT     ^O343
.ENDM

.MACRO .RENAM   AREA,CHAN,DBLK,CODE
.IF NDF ...V1
.MCALL .MACS
```

```
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
        EMT     ^O<100+AREA>
.IFF
...CM1  <AREA>,4.,<CHAN>,<CODE>
...CM2  <DBLK>,2.,X
.ENDC
.ENDM

.MACRO .REOPE   AREA,CHAN,CBLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
        EMT     ^O<140+AREA>
.IFF
...CM1  <AREA>,6.,<CHAN>,<CODE>
...CM2  <CBLK>,2.,X
.ENDC
.ENDM

.MACRO .SAVES   AREA,CHAN,CBLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>
        EMT     ^O<120+AREA>
.IFF
...CM1  <AREA>,5.,<CHAN>,<CODE>
...CM2  <CBLK>,2.,X
.ENDC
.ENDM

.MACRO .RSUM
        MOV     #2.*^O400,%0
        EMT     ^O374
.ENDM

.MACRO .SDAT    AREA,BUF,WCNT,CRTN=#1,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.ENDM

.MACRO .SDATC   AREA,BUF,WCNT,CRTN,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.ENDM

.MACRO .SDATW   AREA,BUF,WCNT,CRTN=#0,CODE
.IF NDF ...V1
.MCALL .MACS
```

```
.MACS
.ENDC
.IIF IDN <CODE>,NOSET, ...CM4 <AREA>,,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.IIF DIF <CODE>,NOSET, ...CM4 <AREA>,#0,<BUF>,<WCNT>,,<CRTN>,21.,<CODE>
.ENDM

.MACRO .SERR
        MOV     #4.*^O400,%0
        EMT     ^O374
.ENDM

.MACRO .SETTO   ADDR
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <ADDR>
        EMT     ^O354
.ENDM

.MACRO .SCCA    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,29.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .SFPA    AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,24.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .SPFUN   AREA,CHAN,FUNC,BUF,WCNT,BLK,CRTN,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM1  <AREA>,26.,<CHAN>,<CODE>
...CM2  <BLK>,2.
...CM2  <BUF>,4.
...CM2  <WCNT>,6.
.IF NB FUNC
.NTYPE ...V2,FUNC
.IF NE ...V2-^O27
.IIF DIF <CODE>,NOSET,...CM2    #^O377,8.,,,B
...CM2  <FUNC>,9.,,,B
.IFF
...CM2  <FUNC'*^O400+^O377>,8.
.ENDC
.ENDC
...CM2  <CRTN>,10.,X,X
.ENDM

.MACRO .SRESE
        EMT     ^O352
.ENDM
```

```
.MACRO .SPND
        MOV     #1*^0400,%0
        EMT     ^0374
.ENDM

.MACRO .SYNCH   AREA,PIC
.IF B PIC
.IIF NB <AREA>  MOV     AREA,%4
.IFF
.IF NB AREA
        MOV     %7,%4
        ADD     #AREA-.,%4
.ENDC
.ENDC
        MOV     @#^054,%5
        JSR     5.,@^0324(5.)
.ENDM

.MACRO .TIMIO   TBK,HI,LO
        JSR     %5,@$TIMIT
        .WORD   TBK-.
        .WORD   0
        .WORD   HI
        .WORD   LO
.ENDM

.MACRO .TLOCK
        MOV     #7.*^0400,%0
        EMT     ^0374
.ENDM

.MACRO .TRPSE   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,3.,0,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .TTINR
        EMT     ^0340
.ENDM

.MACRO .TTYIN   CHAR
        EMT     ^0340
        BCS     .-2.
.IF NB <CHAR>
.IF DIF <CHAR>,R0
        MOVB    %0,CHAR
.ENDC
.ENDC
.ENDM

.MACRO .TTOUT
        EMT     ^0341
.ENDM

.MACRO .TTYOU   CHAR
.IF NB <CHAR>
.IF DIF <CHAR>,R0
        MOVB    CHAR,%0
.ENDC
```

```
.ENDC
        EMT     ^O341
        BCS     .-2.
.ENDM

.MACRO .TWAIT   AREA,TIME,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,20.,0,<CODE>
...CM2  <TIME>,2.,X
.ENDM

.MACRO .UNLOC
        EMT     ^O347
.ENDM

.MACRO .UNMAP   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30.,5.,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .UNPRO   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,25.,1,<CODE>
...CM2  <ADDR>,2.,X
.ENDM

.MACRO .WAIT    CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
        EMT     ^O<240+CHAN>
.IFF
.IF B <CHAN>
        CLR     %0
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^O27
.IF IDN <CHAN>,#0
        CLR     %0
.IFF
        MOV     CHAN,%0
.ENDC
.IFF
        CLR     %0
        BISB    CHAN,%0
.ENDC
.ENDC
        EMT     ^O374
.ENDC
.ENDM
```

```
.MACRO .WDBBK   WNAPR,WNSIZ,WNRID,WNOFF,WNLEN,WNSTS
.MCALL  .WDBDF
.WDBDF
        .BYTE
        .BYTE   WNAPR
        .WORD
        .WORD   WNSIZ
        .WORD   WNRID
        .WORD   WNOFF
        .WORD   WNLEN
        .WORD   WNSTS
.ENDM

.MACRO .WDBDF
W.NID   =0
W.NAPR  =1
W.NBAS  =2.
W.NSIZ  =4.
W.NRID  =6.
W.NOFF  =^O10
W.NLEN  =^O12
W.NSTS  =^O14
W.NLGH  =^O16
WS.CRW  =^O100000
WS.UNM  =^O40000
WS.ELW  =^O20000
WS.MAP  =^O400
.ENDM

.MACRO .WRITC   AREA,CHAN,BUF,WCNT,CRTN,BLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CRTN>
...CM0  <WCNT>
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<220+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9.,<CODE>
.ENDC
.ENDM

.MACRO .WRITE   AREA,CHAN,BUF,WCNT,BLK,CRTN=#1,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <WCNT>
...CM0  #1
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<220+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9.,<CODE>
.ENDC
.ENDM

.MACRO .WRITW   AREA,CHAN,BUF,WCNT,BLK,CRTN=#0,CODE
.IF NDF ...V1
.MCALL .MACS
```

```
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <WCNT>
...CM0
...CM0  <BUF>
...CM0  <CHAN>
        EMT     ^O<220+AREA>
.IFF
...CM4 <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9.,<CODE>
.ENDC
.ENDM
```

APPENDIX C

ADDITIONAL I/O INFORMATION

This appendix provides some additional information on I/O processing that is useful especially to users who need to write their own device handlers. It contains the I/O data structure formats, a flowchart of the sequence of events involved in queued I/O processing, and source listings of two RT-11 device handlers with liberal comments. In addition, this appendix provides information on device directory formats and file structures.

Before writing a device handler, programmers should be familiar with the material in Chapter 1 of this manual. RT-11 provides macros to make handler writing easier; Chapter 1 describes these macros. Appendix B contains a listing of the RT-11 system macro library. It can be helpful to consult the library listing in order to understand how the macros expand and, therefore, how use them correctly.

Programmers should have a thorough knowledge of the hardware device for which they are writing the handler. The PDP-11 Peripherals Handbook contains information on DIGITAL peripherals. The hardware manuals and engineering prints are the most complete source of information for DIGITAL devices and those from other manufacturers.

C.1 I/O Data Structures

RT-11 I/O data structures are described in this section. These data structures provide conventions for communication among an application program, the monitor, and a device handler.

C.1.1 Monitor 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, which is created by system generation and which is assembled separately from the module RMON. SYSTBL is linked with RMON and other modules to form the resident monitor. The symbol $SLOT, which is defined at system generation time, defines the maximum number of devices the system can have.

C.1.1.1 **$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. Extra slots can be allocated at assembly time. Entries are made in $PNAME at monitor assembly time for each device that is built into the system. Free slots can be created 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 keyboard monitor command. The INSTALL keyboard monitor command can be used to install a different device handler into the table after the system has been booted. INSTALL does not make a device entry permanent. The DEV macro in SYSTBL must be used to permanently add a device to the system. The DEV macro is described in Section C.1.1.7.

Each table entry consists of a single word that contains the Radix-50 code for the 2-character physical device name. For example, the entry for DECtape is .RAD50 /DT/. The TT device must be first in the table. 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, and zeroes the $PNAME entry for that device if the parameters do not match. INSTALL cannot install a handler whose conditional parameters do not match those of the monitor.

C.1.1.2 **$STAT Table** - The device status table is called $STAT. Entries to this table are made at assembly time for those devices that are built into the RT-11 system. When the system is bootstrapped, the entries for those devices that are built into the system are updated with information in the handler files that are present on the system device. The system device handler does not have to be present on the system device as a separate .SYS file because it is already a part of the monitor. Entries are made for devices that are not built into the system at assembly time when they are installed with the INSTALL monitor command. Each device in the system must have 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. Figure C-1 shows the meaning of the bits in the status word. For a user-written handler, the programmer sets up the device status word according to the layout in Figure C-1 so it can be stored in block 0 of the handler file. Figures C-10 and C-12, below, show examples of the device status word as it is set up in device handlers. The device status word is part of the information returned to a running program by the .DSTATUS programmed request.

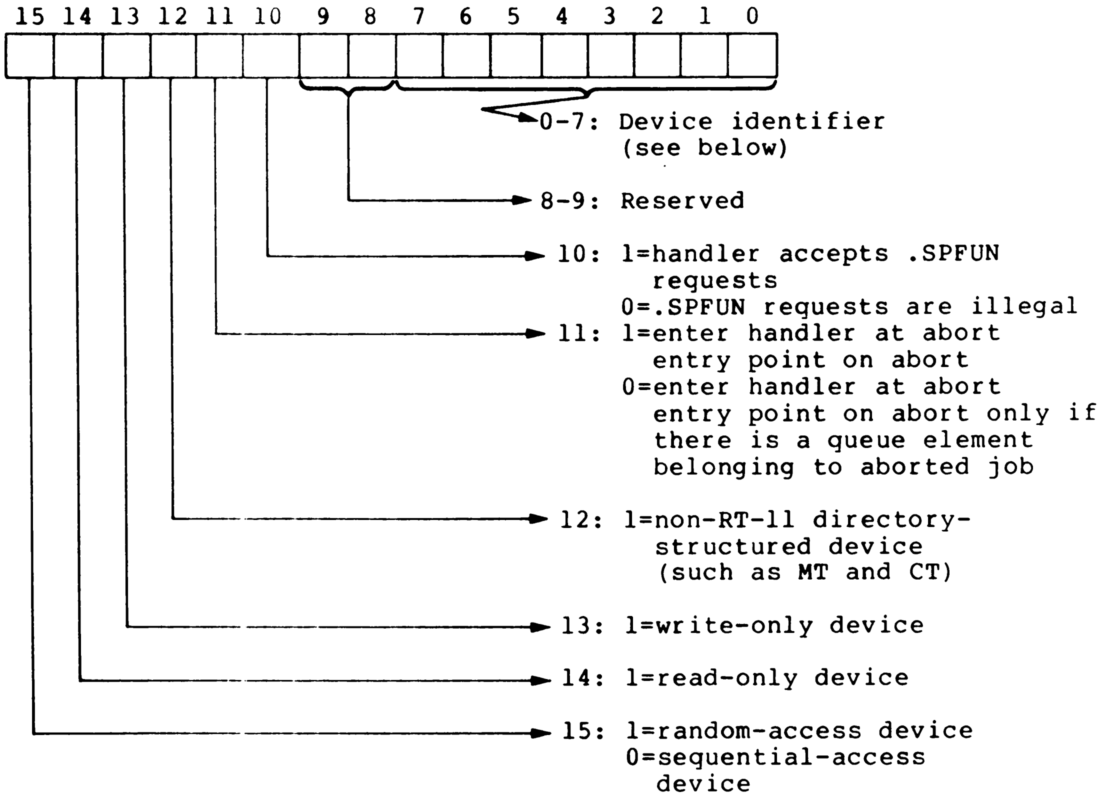


Figure C-1 Device Status Word

Note that bit 11 in the status word should be set only for device handlers that remove the queue element on entry and queue internally.

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

In RT-11, symbolic names are defined for certain bit patterns. This provides a meaningful way to refer to the bits in the device status word. The SYSTBL source file defines the following bit patterns:

```
FILST$  =  100000
RONLY$  =   40000
WONLY$  =   20000
SPECL$  =   10000
HNDLR$  =    4000
SPFUN$  =    2000
```

A programmer can first use direct assignment statements to set up the symbolic names for the bit patterns, as shown above. Then the device status word can easily be constructed by adding the device identifier (described below) to the appropriate bit patterns, according to the following outline:

```
.WORD     device identifier + symbol
```

An example of this is the way the RT-11 code in the file SYSTBL.MAC sets up the device status word for device DX:

```
.WORD     22 + FILST$ + SPFUN$
```

See Section C.1.1.7 for more information on the DEV macro in SYSTBL.

The device-identifier byte uniquely identifies each device in the system. The values are currently defined in octal as follows:

```
     0 = RK05 disk
     1 = TC11 DECtape
     2 = reserved
     3 = line printer
     4 = console terminal or batch handler
     5 = RL01 disk
     6 = RX02 diskette
     7 = PC11 high-speed paper tape reader and punch
    10 = reserved
    11 = magtape
    12 = RF11 disk
    13 = TA11 cassette
    14 = card reader (CR11,CM11)
    15 = reserved
    16 = RJS03/4 fixed-head disks
    17 = reserved
    20 = TJU16 magtape
    21 = RP02/RP03 disk
    22 = RX01 diskette
    23 = RK06/RK07 disk
    24 = error log handler
    25 = null handler
 26-30 = reserved (for Networks)
 31-33 = reserved (for DIBOL LQ, LR, LS)
    34 = TU58 data cartridge
```

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

C.1.1.3 **$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 found on the system device. (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 that 0 is a valid $DVREC entry for permanently resident devices.

C.1.1.4 **$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.

C.1.1.5 **$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. These assignments associate the logical names DK: and SY: with the device from which the system was booted. The value of $SLOT limits the total number of logical name assignments (excluding SY and DK).

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.

C.1.1.6 **$OWNER Table** - The device ownership table is called $OWNER. It is used in the FB and XM environments to arbitrate device ownership. The table is ($SLOT*2) words in length and is divided into 2-word entries for each device. Entries are made into this table when the LOAD keyboard monitor command is issued. Each 2-word entry is in turn divided into eight 4-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. Thus, each device is presumed to have up to eight units, each assigned independently of the others. However, if the device is nonfile-structured, units are not assigned independently: the monitor ASSIGN code ensures that ownership of all units is assigned to one job.

When either a background or a foreground 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 the other job owns the unit, a fatal error is generated.

The device is assumed to be public if the 4-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. Bit 0 of the field being set indicates that the unit ownership is assigned to a job (1 for the background job and 3 for the foreground job).

C.1.1.7 **Adding a Device to the Tables** - The DEV macro in SYSTBL.MAC is used to define devices in the system. The format of the DEV macro is as follows:

```
DEV name,s,type
```

The arguments in the macro shown above have the following meaning:

name
: represents the two-character physical device name, such as RK or DX.

s
: represents the device status word. This word consists of a device identification code plus a set of device characteristics bits from the following set:

```
FILST$ = 100000
RONLY$ =  40000
WONLY$ =  20000
SPECL$ =  10000
HNDLR$ =   4000
SPFUN$ =   2000
```

type
: must be SYS if the device can be a system device. A device can be a system device if it is random-access and file-structured.

Examples of the DEV macro as used in SYSTBL are as follows:

```
DEV RK,0+FILST$,SYS

DEV LP,3+WONLY$

DEV MT,11+SPECL$+SPFUN$
```

C.1.2 The 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, the bitmap is updated 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 a new background program is run.

The bitmap is a 20 (decimal) byte table that starts 326 (octal) bytes from the beginning of the Resident Monitor. Table C-1 lists the offset from RMON and the corresponding locations represented by that byte.

Table C-1
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 8 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 individual words are protected, not bytes. Thus, protecting word 0 means that both locations 0 and 1 are protected.

If locations 24 and 26 are protected, the second byte of the table contains:

    00110000

The leftmost bit represents location 20 and the rightmost bit represents location 36. To protect locations 300 through 306, 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 users need to protect vectors, they should use one of the two following methods:

1. Use PATCH to manually modify the bitmap
2. Dynamically modify the bitmap from within a running program

For example, to protect locations 300 through 306 dynamically, the following instructions can be used:

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

Protecting locations with PATCH means that the vector is permanently protected, even if the system is rebootstrapped. The dynamic method provides a temporary measure and does not remain effective across bootstraps. Users are cautioned that the dynamic method involves storing data directly into the monitor. For this reason, it is recommended that SJ users use PATCH to protect vectors.

C.1.3 Queue Elements

The RT-11 system uses queues to organize requests in a first-in/first-out order. Requests for I/O transfers, completion routines, and timer routines are queued for later service. Each request uses one queue element. The elements are arranged in linked lists so that they are processed in order. Each element contains all the information necessary to initiate and process a single request. Foreground requests are added to an I/O queue in front of background requests. However, a foreground request cannot replace an active background request (the current queue element).

C.1.3.1 **I/O Queue Element** - Once a device handler is in memory, any .READ/.WRITE programmed request for the corresponding device is interpreted by the monitor and translated into a call to the I/O device handler. To facilitate the overlapping of I/O and computation, all I/O requests in RT-11 are processed through an I/O queue.

The RT-11 I/O queue is made up of one linked list of queue elements for each resident device handler. I/O queue elements are seven words long for SJ and FB systems, and ten 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. This is sufficient for any program that uses wait mode I/O (.READW/.WRITW). However, for maximum throughput, the .QSET programmed request should be used at the beginning of a program to create one additional queue element for each asynchronous I/O request that can be outstanding. Then, asynchronous I/O should be used.

If an I/O transfer is requested and a queue element is not available, RT-11 must wait until an element is free before it can queue the request. This obviously slows program execution. If the program requires asynchronous I/O, it must allocate extra queue elements. It is always sufficient to allocate N new queue elements, where N is the maximum number of pending requests that can be outstanding at any time in a particular program. This produces a total of N+1 available elements, since the element in the job's impure area is added to the list of available elements.

Figure C-2 shows the format of an I/O queue element and the meaning of each entry. The .QELDF macro defines symbolic names for the offsets from the beginning of the I/O queue element and a symbolic name for the size of the queue element. Figure C-2 also shows the offsets and the symbolic name that is associated with each offset.

Note that .QELDF defines offsets from the beginning of the queue element. From within a device handler, the pointer to the current queue element points to the third word of the element. Therefore, the offsets from .QELDF cannot be used directly to access words in the queue element. The following example from the PC handler illustrates a construction that is typically used in handlers to account for this discrepancy:

```
BUFF = Q.BUFF - Q.BLKN
```

<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 C-7)</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
2 = FG</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>
</table>

Figure C-2 I/O Queue Element Format

Name	Offset	Contents
Q.WCNT	12	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.
Q.COMP	14	Completion routine code: if 0, this is wait mode I/O if 1, just queue the request and return if even, completion routine address
Q.PAR	16	PAR1 Relocation Bias (XM only)
		reserved (XM only)
		reserved (DECnet)

Figure C-2 I/O Queue Element Format (Cont.)

Q.LINK, the link to the next queue element, points to the third word of the next queue element, not to its first word.

Q.LINK and Q.CSW are 16-bit physical addresses. They are always used in kernel mode, and therefore must always be in the lower 28K words of memory.

In XM systems, Q.BUFF is always an address between 20000 and 37777. To access the byte in the user's physical memory, the monitor loads PAR1 (Page Address Register 1 of the KT11 memory management hardware) with the Q.PAR values and then uses Q.BUFF as a pointer to the correct byte.

C.1.3.2 **Timer Queue Element** - Another queue maintained by the monitor is the timer queue. This queue is used to implement the .MRKT time and .TIMIO requests, which schedule completion routines to be entered after a specified period of time.

Figure C-3 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 2-word number, a modified expression of the number of ticks until the timed wait expires. (There are sixty ticks per second when 60 Hz power is used, and 50 ticks per second when 50 Hz power is used.) The timer queue elements are stored in the queue in order of their expiration times. An optional sequence number can be added to the request to distinguish it from others issued by the same job.

The monitor uses the timer queue internally to implement the .TWAIT programmed request. The .TWAIT request causes the issuing job to be suspended. A timer request is placed in the queue with the .RSUM programmed request logic as the completion routine. This causes execution to wait until the desired time has elapsed. Then execution resumes when the monitor itself issues the .RSUM programmed request.

A range of owner's sequence number IDs is reserved for use by DIGITAL software. All values in the range from 177400 through 177777 are reserved for DIGITAL. These values should not be used by customer programs.

There are several uses for system timer elements. If C.SYS is -1, the element is being used for either multi-terminal time-out support, or for device handler time-out support. If C.SYS is -3, the element is being used to implement a .TWAIT request in the XM monitor.

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 user's 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.

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

Figure C-3 Timer Queue Element Format

C.1.3.3 **Completion Queue Element** - The FB and XM monitors maintain one queue of I/O completion requests for each job. When an I/O transfer completes, the I/O queue element indicates whether or not a completion routine was specified in the request. If the seventh word of the I/O queue element is even and nonzero, a completion routine was requested. The queue completion logic in the monitor transfers the I/O request queue element to the completion queue. It places the channel status word and channel offset in the element. This has the effect of serializing completion routines, rather than nesting them. Elements are also added to this queue when a timer request expires and when a .SYNCH request is issued. The completion queue is a first-in/first-out queue. The completion routines are entered at priority level 0 rather than at interrupt level. In SJ, completion routines can interrupt each other. In FB and XM, no other code except interrupts can execute when a completion routine is running.

Note that completion routines are not serialized in the SJ environment, because there is no completion queue in SJ. Completion routines in SJ do not run in a first-in/first-out order. They are executed as soon as the I/O or timer request is complete.

Figure C-4 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.

Name	Offset	Contents
Q.LINK	0	Link to next queue element; 0 if none
	2	Undefined
	4	Undefined
	6	Undefined
Q.BUFF	10	Channel status word
Q.WCNT	12	Channel offset
Q.COMP	14	Completion routine address

Figure C-4 Completion Queue Element Format

C.1.3.4 **Synch Queue Element** - In the FB and XM monitors the .SYNCH request makes use of the completion queue. When the .SYNCH programmed request is entered, the 7-word area supplied with the request is linked into the head of the completion queue, where it appears to be a request for a completion routine. The .SYNCH request then does an interrupt exit. The completion queue manager next calls the code following the .SYNCH request at priority level 0 (after a possible context switch to bring in the correct job). To prevent the .SYNCH block from the user's program from being linked in the queue of available queue elements after the routine exits, the sixth word is set to -1. The completion queue manager checks the sixth word before linking a queue element back into the list of available elements, and skips elements with -1 there.

In the SJ monitor, the .SYNCH request sets up the registers, drops priority to 0, and calls the code following the request as a co-routine. When the co-routine returns, the .SYNCH logic does an interrupt exit.

Figure C-5 shows the format of a synch queue element. It includes the symbolic names and offsets as well as the contents of each word in the data structure.

Name	Offset	Contents
Q.LINK	0	Link to next queue element; 0 if none
Q.CSW	2	Job number
Q.BLKN	4	Undefined
Q.FUNC	6	Undefined
Q.BUFF	10	Synch ID
Q.WCNT	12	-1
Q.COMP	14	Synch routine address

Figure C-5 Synch Queue Element Format

C.1.3.5 **Fork Queue Element** - The RT-11 system maintains one fork queue. Its root is in the Resident Monitor. The device handler must provide a 4-word fork block that will be used as the fork queue element. Section 1.4.4.1 in this manual describes the .FORK macro.

Figure C-6 shows the format of a fork queue element. It includes the symbolic names and offsets as well as the contents of each word in the data structure.

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

Figure C-6 Fork Queue Element Format

C.1.4 I/O Channel Format

Figure C-7 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.

Name	Offset	Contents	
	0	Channel status word	
C.SBLK	2	Starting block number of this file (0 if nonfile structured)	
C.LENG	4	Length of file (if opened by .LOOKUP); Size of empty area (if opened by .ENTER)	
C.USED	6	Actual data length (if .LOOKUP); Highest block written (if .ENTER)	
C.DEVQ	10	Device unit number	Number of requests pending on this channel

Figure C-7 I/O Channel Description

Figure C-8 shows the significant bits in the channel status word.

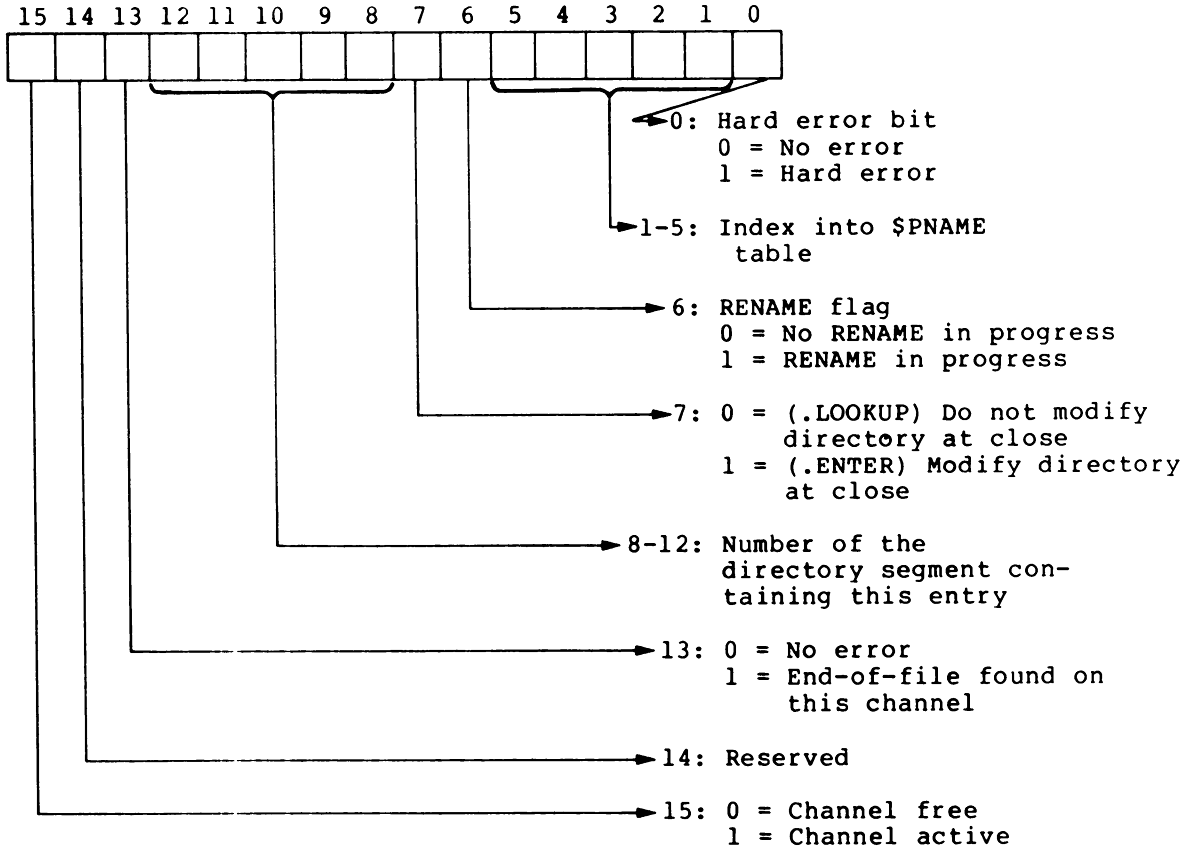


Figure C-8 Channel Status Word

C.2 Flow of Events in I/O Processing

Figure C-9 shows a simplified diagram of the flow of events involved in an I/O transfer. The following example, a read request to the RK disk handler, shows the relationship between the application program and the queue elements, and between the queue elements and the device handler. The flow of events for a non-DMA device is slightly different. (Figure C-12 shows a device handler for a non-DMA device, the paper tape reader and punch.)

This simplified diagram assumes that no other interrupts occur during this processing, and that the FB monitor is being used.

Event	[Processor priority]
Application program issues: 1. .READW or 2. .READC or 3. .READ	[PR0]
Monitor processes the programmed request in the EMT processor section.	[PR0]
Monitor takes a queue element from the list of available elements for this job. Priority 7 during this procedure ensures that the queue of available elements remains stable.	[PR7]
Monitor places the element on the handler's queue. The monitor holds the handler during this procedure to ensure that the handler's queue remains stable.	[PR0]
If handler is not busy, the monitor calls it at the sixth word (start of the the I/O initiation section).	[PR0]
Handler computes disk address, determines type of operation, starts device, and returns to monitor.	[PR0]
1. If .READW issued, monitor waits for I/O to complete before returning to the application program. 2. Program runs if .READC issued. 3. Program runs until .WAIT if .READ issued.	[PR0]

Meanwhile, the device is performing the I/O transfer.

until...

Event	[Processor priority]
Device interrupts through vector; new PC and PS start execution in handler at the interrupt entry point. Handler calls monitor through $INPTR to $INTEN.	[PR7]

Figure C-9 Flow of Events in I/O Processing

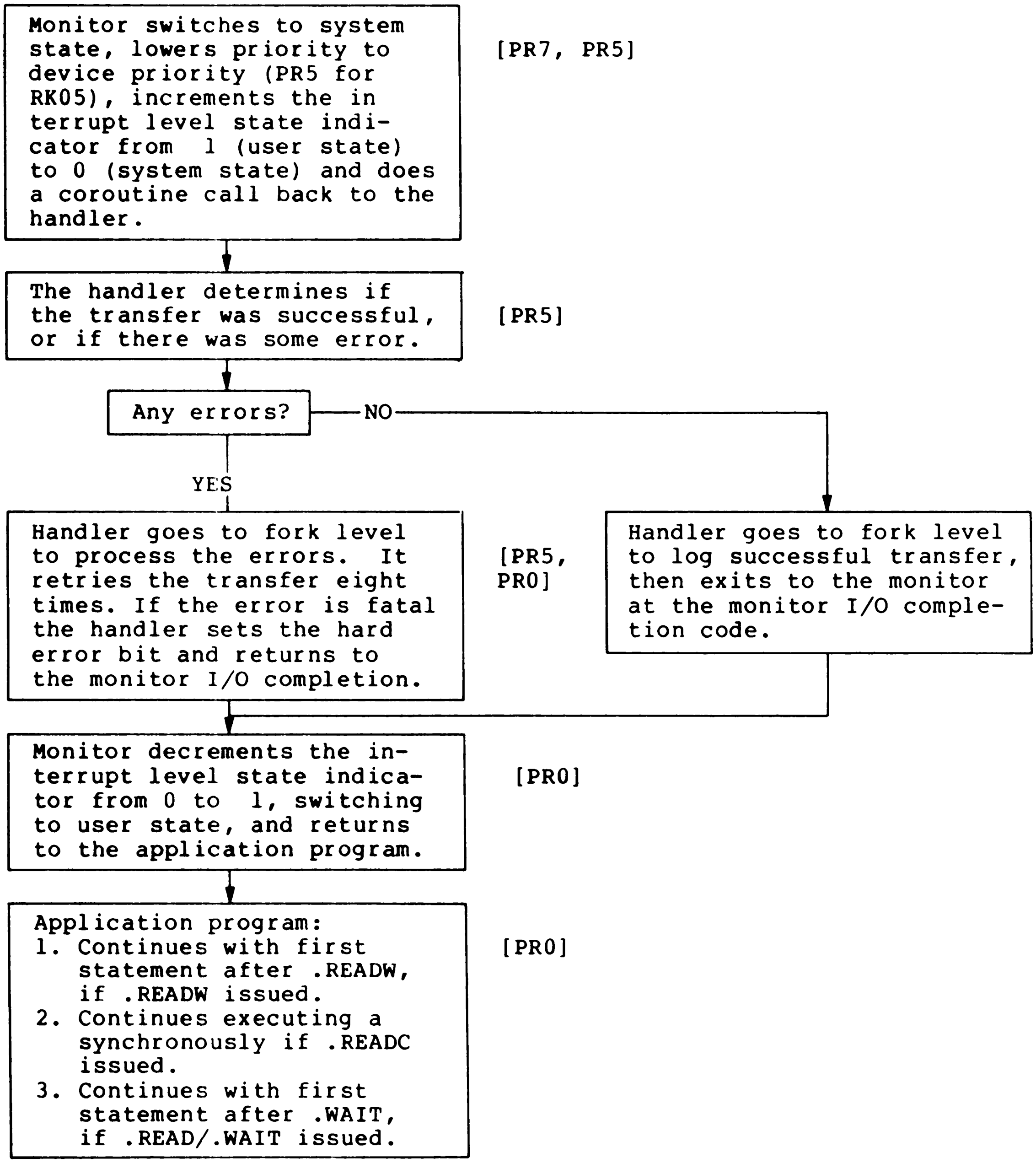


Figure C-9 Flow of Events in I/O Processing (Cont.)

C.3 Study of the RK05 Handler

Figure C-10 provides a listing of the assembled RK05 handler file. The comments give a detailed explanation of the handler. The RK05 handler was chosen as a representative handler for a random access disk that can be a system device. For this example, the RK handler was assembled as a data device only. See Section C.4 for information on system device handlers.

In Figure C-10, black ink is used for text and comments. Red ink is used for the actual computer output of the RK05 handler assembly listing.

Device handlers are written in position independent code, called PIC. The PDP-11 processors offer addressing modes that make it possible to write instructions that are not dependent on the virtual addresses to which they are linked. A body of such code is termed position independent, and can be loaded and executed at any virtual address. (See Appendix G, "Writing Position Independent Code", in the PDP-11 MACRO-11 Language Reference Manual, order number AA-5075A-TC.) Throughout the RK05 handler listing, coding constructions that were used specifically to make the handler position independent are marked as [PIC].

This listing was produced by assembling the conditional file RKCND.MAC together with the RK handler source file, RK.MAC. The command strings to produce this assembly and the listing file RK.LST are as follows:

Keyboard monitor command:

```
.MACRO/LIST:RK.LST/NOOBJECT/SHOW:ME:MEB:TTM RKCND.MAC+RK.MAC
```

MACRO program commands:

```
.R MACRO
*,RK.LST/L:ME:MEB:TTM=RKCND.MAC,RK.MAC
```

The first file listed below, RKCND.MAC, was created especially for this example. It was assembled together with the handler source file, RK.MAC, to produce code for the three system generation conditions: memory management, error logging, and device time-out. Normally, a device handler is assembled with the system conditional file, SYCND.MAC, to ensure that the handler has the same system generation parameters as does the current monitor.

This listing was produced by assembling the conditional file RKCND.MAC together with the RK handler source file, RK.MAC. The command strings to produce this assembly and the listing file RK.LST are as follows:

Keyboard monitor command:

```
.MACRO/LIST:RK.LST/NOOBJECT/SHOW:ME:MEB:TTM RKCND.MAC+RK.MAC
```

MACRO program commands:

```
.R MACRO
*,RK.LST/L:ME:MEB:TTM=RKCND.MAC,RK.MAC
```

The first file listed below, RKCND.MAC, was created especially for this example. It was assembled together with the handler source file, RK.MAC, to produce code for the three system generation conditions: memory management, error logging, and device time-out. Normally, a device handler is assembled with the system conditional file, SYCND.MAC, to ensure that the handler has the same system generation parameters as does the current monitor.

```
RK05  V03.01 MACRO V03.02B6-SEP-78 11:55:53 PAGE 1

                ;CONDITIONAL FILE FOR RK HANDLER EXAMPLE
                ;
                ;RKCND.MAC
                ;
                ;9/1/78 JAD
                ;
                ;ASSEMBLE WITH RK.MAC TO TURN ON 18-BIT I/O,
                ;TIME-OUT SUPPORT, AND ERROR LOGGING FOR
                ;RK HANDLER
                ;
        000001  MMG$T    = 1           ;TURN ON 18-BIT I/O
        000001  ERL$G    = 1           ;TURN ON ERROR LOGGING
        000001  TIM$IT   = 1           ;TURN ON TIME-OUT SUPPORT
```

The listing of the RK handler source file that follows is current for RT-11 V03B; it includes one source patch. Comments that are part of the source file itself are all upper-case characters and begin with a semicolon (;). Comments that were added as documentation in this appendix are upper- and lower-case characters.

Figure C-10 RK05 Handler Listing

```
RK05  V03.01 MACRO V03.02B6-SEP-78 11:55:53 PAGE 2

                        ;RK EDIT LEVEL 0
                        .TITLE  RK05  V03.01
                        .IDENT  /V03.01/
                        ; RT-11 DISK (RK11) HANDLER
                        ;
                        ; COPYRIGHT (C) 1978
                        ;
                        ; DIGITAL EQUIPMENT CORPORATION
                        ; MAYNARD, MASSACHUSETTS  01754
                        ;
                        ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY
                        ; ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY 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 EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO
                        ; AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP
                        ; OF THE SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.
                        ;
                        ; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO
                        ; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
                        ; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
                        ;
                        ; DEC ASSUMES NO RESPONSIBILITY FOR THE USE
                        ; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
                        ; WHICH IS NOT SUPPLIED BY DEC.
```

```
RK05  V03.01 MACRO V03.02B6-SEP-78 11:55:53 PAGE 3

                        .ENABL LC
```

****

The device handler Preamble Section starts here.

****

```
                        .MCALL  .DRBEG,.DREND,.FORK,.DRAST,.DRFIN,.QELDF
```

Each macro that is used in the handler requires the .MCALL statement, as shown above. The .QELDF, .DRBEG, .DRAST, .DRFIN, and .DREND macros are provided in the system macro library in order to simplify writing a device handler.

Figure C-10 RK05 Handler Listing (Cont.)

```
                        ; SYSTEM GENERATION OPTIONS:
```

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 options are present in the system. When a system is generated, the handler files are assembled together with SYCND.MAC, the system conditional file, so that the correct conditionals are defined in the handler files. If a handler is to be added to an existing system, it should be assembled with the same conditional file that was used for the rest of the system. If there is no conditional file assembled with the handler file, the conditionals are turned off by the following three lines of code (for the purpose of this example, the three following conditionals were set to 1 by the preceding file, RKCND.MAC):

```
                        .IIF NDF TIM$IT,TIM$IT=0 [No device time-out support]
                        .IIF NDF MMG$T, MMG$T=0  [No memory management]
                        .IIF NDF ERL$G, ERL$G=0  [No error logging]

                        .NLIST CND
```

For the purpose of this listing, printing of conditional source lines is suppressed within the expansion of system macros. This is accomplished by the .NLIST CND and .LIST CND pair of directives.

```
   000000               .QELDF
```

The .QELDF macro defines symbolic offsets into the I/O queue elements. See Figure C-2 above for a diagram of the I/O queue element.

```
        000000  Q.LINK=0       [Link to next queue element]
        000002  Q.CSW=2.       [Pointer to channel status word]
        000004  Q.BLKN=4.      [Physical block number]
        000006  Q.FUNC=6.      [Special function code]
        000007  Q.JNUM=7.      [Job number]
        000007  Q.UNIT=7.      [Device unit number]
        000010  Q.BUFF=^O10    [User virtual memory buffer address]
        000012  Q.WCNT=^O12    [Word count]
        000014  Q.COMP=^O14    [Completion routine code]
        000016  Q.PAR=^O16     [PAR1 relocation bias]
        000024  Q.ELGH=^O24    [End of queue element, used to find length]
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                .LIST CND

```

The following direct assignment statements are required only if the handler can be a system device. For this example the RK handler was assembled as a mass storage device only, and not as a system device. Therefore, the symbol $RKSYS in SYCND.MAC was not set to 1. It does not cause a problem to leave the assignment statements in place if the handler is being assembled only as a storage device and not as a system device. The globals being defined here are the entry points for all the other system devices in the RT-11 system.

```
        000000  DTSYS   == 0                    ;THIS IS RK HANDLER
        000000  DLSYS   == 0
        000000  DSSYS   == 0
        000000  DXSYS   == 0
        000000  DPSYS   == 0
        000000  RFSYS   == 0
        000000  DMSYS   == 0
        000000  DYSYS   == 0

                ; RK CONTROL DEFINITIONS:
```

The next two statements define the vector and CSR addresses for the RK device, if they have not already been defined in the system conditional file, SYCND.MAC. The default vector is 220; the default CSR address is 177400.

```
                .IIF NDF RK$VEC, RK$VEC == 220
                .IIF NDF RK$CSR, RK$CSR == 177400
```

The following group of direct assignment statements set up the device control registers. The device control register names, locations, and operation codes can be found in the PDP-11 Peripherals Handbook and in the hardware manual for the device.

```
        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 Register]
        177410  RKBA    = RKDS+10   [Current Bus Address Register]
        177412  RKDA    = RKDS+12   [Disk Address Register]
                [RKDB, the Data Buffer Register, is not used]

```

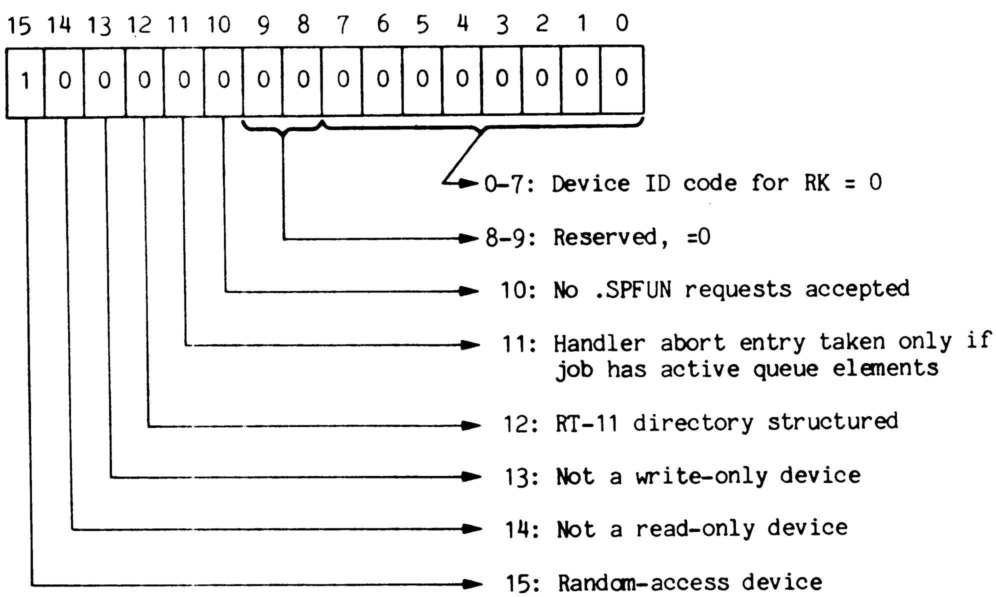
Figure C-10 RK05 Handler Listing (Cont.)

The symbol RKCNT represents the number of times to retry an I/O transfer should an error occur.

```
        000010  RKCNT   = 10                    ;# ERROR RETRYS

```

The device status word RKSTS and the device size word RKDSIZ are set up here. The information they contain is used by the .DSTATUS programmed request, which returns the information to a running program. See Figure C-1 for the format of the device status word. The diagram below shows how the code 100000 was selected for the RK device status word.




```
        100000  RKSTS   = 100000                ;DEVICE SYSTEM STATUS
                                                ;WORD ($STAT)
        011300  RKDSIZ  = 11300                 ;DEVICE BLOCK SIZE ($DVSIZ)
```

The next four direct assignment statements are for error logging.

```
        000000  RKIDEN  = 0                     ;RK11 ID = 0 IN HIGH BYTE
                                                ;FOR ERROR LOG
        000377  RKIDS   = 377                   ;RK11 DEVICE ID = 0 IN HIGH
                                                ;BYTE
                                                ;-1 IN LOW BYTE FOR I/O
                                                ;SUCCESS TO ERROR LOG
```

Figure C-10 RK05 Handler Listing (Cont.)

```
        004000  RKRCNT  = 4000                  ;I/O RETRY COUNT IN HIGH BYTE
        000007  RKNREG  = 7                     ;# OF REGISTERS TO READ
                                                ;FOR ERROR LOG
```

****

The device handler Header Section begins here.

****

```
                        ; START OF DRIVER
                        .NLIST CND
000000                          .DRBEG  RK,RK$VEC,RKDSIZ,RKSTS
```

The .DRBEG macro generates the following block of code (up to the next .LIST CND directive):

```
000000          .ASECT  [Stores information in block 0 of handler]
        000052  . = 52
                .GLOBL  RKEND
000052  000550          .WORD   <RKEND - RKSTRT>
000054  011300          .WORD   RKDSIZ
000056  100000          .WORD   RKSTS
```

The three words shown above are extracted by the bootstrap.

Normally, determining the size of the device for the xxDSIZ word, above, is a simple matter. However, some device handlers can control devices that permit two different size volumes to be used. An example of this is the DM handler, which can access either RK06 or RK07 disks through a single controller. Such handlers should place the size of the smaller volume in the xxDSIZ word, above. If necessary, the handler can permit a running program to issue an .SPFUN programmed request to determine the size of the volume that is currently mounted. Bit 10 (SPFUN$) of the device status word must be set by the handler at assembly time to indicate that .SPFUN requests are allowed.

The DM handler, for example, handles I/O to the RK06 and RK07 disks as follows. First, it selects a unit (0 through 7) of the device by placing opcode 01 in RKCS1 (the RK06/07 Control and Status Register 1). Then it gets the value of bit 8 from RKDS (Drive Status Register). A value of 0 means that the selected unit is an RK06. A value of 1 indicates RK07. Next, the handler puts this value, the 0 or 1, into bit 10 of RKCS1. Finally, it is ready to calculate the correct disk address and do a data transfer.

Figure C-10 RK05 Handler Listing (Cont.)

```
000000          .CSECT  [Returns to the unnamed .PSECT]
000000          RKSTRT::
                .GLOBL  RKINT
```

The first word of the handler, RK$VEC, contains the vector address for the device:

```
000000 000220          .WORD   RK$VEC
```

The second word of the handler, shown below, 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. The abort entry point is the word preceding the RKINT label.

```
000002 000172          .WORD   RKINT -
```

The third word of the handler, shown below, 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. However, 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 into the PS word of the vector and set in the PS when the device interrupts using that vector.

The monitor also uses the third word of the handler as a flag area in order to hold the handler. When the monitor needs to manipulate the I/O queue of a handler while I/O is active, or if it must abort the handler, it prevents the handler from completing a transfer and releasing a queue element by setting bit 15 of this word. It actually does this by rotating the C bit into bit 15. If the handler does a .DRFIN operation while it is held, the monitor shifts word 3 right again, effectively setting bit 14, and returns without affecting the queue. When the handler is freed later, the monitor checks to see if bit 14 was set, indicating that the handler tried to return a queue element while it was held. If that is so, monitor routine COMPLT is called for the handler to return the queue element and start an I/O operation on the next queue element.

```
000004 000340          .WORD   ^O340

000006          RKSYS::[Required if the device can be a system device]
```

Figure C-10 RK05 Handler Listing (Cont.)

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 RMON 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
     45                         .LIST CND
```

****

The handler I/O Initiation Section begins here.

****

```
     46                         .IF EQ MMG$T
```

Most of the code in the handler is assembled based on the value of certain conditionals, such as MMG$T. The IF statement above controls the assembly of the code in this handler. If the handler is assembled with MMG$T = 1 (that is, with extended memory support enabled), code following the .IFF statements is assembled. If the handler does not have extended memory support enabled (that is, if MMG$T = 0), code following the .IFT statements is assembled. Code following the .IFTF statements is always assembled, regardless of the value of MMG$T.

```
     47                         .IFTF
```

The next statement is the first executable statement of the handler code. 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 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 prior element was released. This section initiates the I/O transfer.

Figure C-10 RK05 Handler Listing (Cont.)

The I/O initiation code is executed at priority 0 in system state. This 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 to use in this section. At the end of the section, control is returned to the monitor with an RTS PC. The I/O queue guarantees that transfers will be serialized. Because of this, RT-11 device handlers are not re-entrant. To minimize their size, they are not written as pure code and data segments.

```
000012  012727          MOV     #RKCNT,(PC)+     ;SET ERROR RETRIES
        000010
```

The MOV statement above 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 knows that it has a brand new queue element, and that a retry is not in progress.

```
000016  000000  RETRY:  0                        ;HIGH ORDER BIT USED FOR
                                                 ;RESET IN PROGRESS FLAG
```

If bit 15 of the word at RETRY: is 1 (that is, if the word is negative), then a retry is in progress.

```
000020  016705          MOV     RKCQE,R5         ;GET Q PARAMETER POINTER
```

RKCQE points to the block number Q.BLKN in the I/O queue element.

```
        177764
000024  011502          MOV     @R5,R2           ;R2 = BLOCK NUMBER
000026  016504          MOV     2(R5),R4         ;R4 = UNIT NUMBER
        000002                                   [The controller requires
                                                 the unit number in the top
                                                 three bits of the word
                                                 loaded into RKDA.]
000032  006204          ASR     R4               ;ISOLATE UNIT BITS IN
                                                 ;HIGH 3 BITS
000034  006204          ASR     R4
000036  006204          ASR     R4
000040  000304          SWAB    R4
000042  042704          BIC     #^C<160000>,R4
        017777
000046  000404          BR      2$               ;ENTER COMPUTATION LOOP
```

The device unit number and block number are known; the disk address for a read or write request must be calculated. Once calculated, the disk address is stored in DISKAD in case it must be used

Figure C-10 RK05 Handler Listing(Cont.)

again during retries. 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     #177760,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')=16+S'=4+S
        000004
000104  060304  3$:     ADD     R3,R4           ;R4 NOW CONTAINS RK ADDRESS
000106  010467          MOV     R4,DISKAD       ;SAVE DISK ADDRESS
```

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, sector. The next statement points R5 to a queue element, since perhaps this is a retry and R5 is not already set up.

```
        000016
000112  016705  AGAIN:  MOV     RKCQE,R5        ;POINT R5 TO Q ELEMENT
        177672
000116  012703          MOV     #103,R3         ;ASSUME A WRITE
        000103
```

The operation code for a write with interrupt enabled is 103. This information is in the PDP-11 Peripherals Handbook.

```
000122  012704          MOV     #RKDA,R4        ;POINT TO DISK ADDRESS REG
        177412
000126  012714          MOV     (PC)+,@R4       ;PUT IN ADDRESS  UNIT SELECT
```

In the statement above, (PC)+ refers to DISKAD:.

```
000130  000000  DISKAD: 0                       ;SAVED COMPUTED DISK ADDRESS
```

The following statement adds 4 to R5, so that R5 points to Q.BUFF in the queue element.

Figure C-10 RK05 Handler Listing (Cont.)

```
78 000132  022525          CMP     (R5)+,(R5)+     ;ADVANCE TO BUFFER ADDRESS
                                                   ;IN Q ELEMENT
79                 .IFT
80                         MOV     (R5)+,-(R4)     ;PUT IN BUFFER ADDRESS
81                 .IFF
82 000134  004777          JSR     PC,@$MPPTR      ;CONVERT TO PHYSICAL ADDR
```

In the line above, $MPPTR is a pointer to the monitor routine $MPPHY. See Section 1.4.4.5 of this manual for information on the $MPPHY monitor routine. 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. Section 1.4.4.5 explains how to use the routine, and lists the calling conventions, required inputs, and the outputs of the routine.

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 (non-NPR) devices. NPR devices such as the RK must convert this pair of words into an 18-bit physical address consisting of a 16-bit low part and a two-bit extension part. The extension bits are in positions 4 and 5 for use with UNIBUS controllers. The routine $MPPHY is called through the pointer $MPPTR to do this address conversion. The extension bits must be ORed into the command word being built for RKCS (see statement number 93, below).

```
           000370
83 000140  012644          MOV     (SP)+,-(R4)     ;MOVE LO 16 BITS INTO PLACE
84                 .IFTF
```

The next statement moves the word count Q.WCNT from the queue element into RKWC, the device word count register. (Note that Q.WCNT is a word count.) If the device is character oriented, the word count must be shifted left to change it to a byte count (the same as multiplying it by 2). RT-11 can transfer up to 32767 words per operation. However, it can never transfer an odd number of bytes.

```
85 000142  012544          MOV     (R5)+,-(R4)     ;PUT IN WORD COUNT
86 000144  001406          BEQ     7$              ;0 COUNT => SEEK
87 000146  100402          BMI     5$              ;NEGATIVE => WRITE
```

The RK controller requires that all word counts be negative.

```
88 000150  005414          NEG     @R4             ;POSITIVE => READ.
                                                   ;FIX FOR CONTROLLER
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                    ;       ADD     #2,R3           ;START UP A READ
```

The statement above was replaced by the following statement as a result of a source patch to the V03B handler source file. The following statement converts a write operation code to a read operation code by adding 2 to it. The operation code 105 is for a read operation with interrupt enabled.

```
   000152  122323           CMPB    (R3)+,(R3)+     ;CHANGE COMMAND CODE TO READ
   000154           5$:
                    .IFF
```

The following operation is necessary for the creation of an 18-bit physical address. The 2-bit extension must be ORed into the command word being built for RKCS.

```
   000154  052603           BIS     (SP)+,R3        ;SET IN HI ORDER ADDRESS BITS
                    .IFTF
```

The next statement starts the operation, whatever it is, by moving the operation code to RKCS, the device control and status register.

```
   000156  010344   6$:     MOV     R3,-(R4)        ;START THE OPERATION
```

The next statement returns control to the monitor. The I/O transfer continues concurrently.

```
   000160  000207           RTS     PC              ;AWAIT INTERRUPT
```

The next statement is reached if the operation is a seek. The operation code for a seek with interrupt enabled is 111.

```
   000162  012703   7$:     MOV     #111,R3         ;START UP A SEEK
           000111
                    .IFF
   000166  005016           CLR     (SP)            ;NO HI ORDER MEMORY ADDRESS
                                                    ;ON SEEK
                    .IFT
   000170  000771           BR      5$              ;AWAIT INTERRUPT
```

Figure C-10 RK05 Handler Listing (Cont.)

****

The handler Asynchronous Trap Entry Section begins here.

****

The following code is reached when an interrupt occurs.

```
                        ; ASYNCHRONOUS TRAP ENTRY POINT TABLE

                        .NLIST CND
    000172                      .DRAST  RK,5
```

The .DRAST macro generates the following block of code (up to the next .LIST CND directive):

```
                        .GLOBL  $INPTR
      000172  000207    RTS     %7
```

The abort entry point is the word preceding RKINT:. Since no abort entry point was specified in the .DRAST macro, above, RTS PC was generated.

Disks are always allowed to complete an I/O transfer attempt. Aborting them in the middle of an operation is not necessary, and can possibly corrupt the disk. It is not practical to try to stop a disk during an I/O transfer. So, abort requests are ignored by doing an RTS PC. (In contrast, see the corresponding section of the PC handler in Section C.5 of this appendix. The PC handler has an abort entry point because the paper tape reader or punch must be stopped to abort an I/O transfer.)

```
      000174  004577  RKINT:: JSR     %5,@$INPTR
              000344
      000200  000100          .WORD   ^C<5*^040>^0340

                        .LIST CND
```

If the handler is for a system device, the bootstrap fills in vector 220 and the pointers to the fixed offsets in the Resident Monitor. (The bootstrap also relocates the pointers, which are actually set up by defining the values at assembly time.) Otherwise, the information is filled in when the handler is made resident by .FETCH or LOAD.

At interrupt time, the new PC (RKINT:) and new PS (340) are used. The handler calls the monitor through $INPTR in the handler to

Figure C-10 RK05 Handler Listing (Cont.)

$INTEN in the monitor. The monitor lowers priority from 7 to 5, switches to system state, and calls the handler back.

```
                         ; INTERRUPT ENTRY POINT
```

The monitor calls the handler back at this point. Execution is at priority 5 and is in system state. The hardware has now finished the I/O operation, and the handler must determine if the transfer was successful or if there was an error.

```
000202  012705      MOV    #RKER,R5       ;POINT TO ERROR STATUS
                                          ;REGISTER
        177402
000206  012504      MOV    (R5)+,R4       ;SAVE ERRORS IN R4,
                                          ;POINT TO RKCS
```

The value of RETRY is negative if a drive reset was just done. (Bit 15 is the retry flag.)

```
000210  005767      TST    RETRY          ;WERE WE DOING A DRIVE RESET?
        177602
000214  100013      BPL    NORMAL         ;NO-NORMAL OPERATION
000216  005715      TST    @R5            ;YES-ANY ERROR?
```

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  100411      BMI    NORMAL         ;YES-HANDLE NORMALLY
```

R5 points to RKCS, the device control and status register.

```
000222  032715      BIT    #20000,@R5     ;RESET COMPLETE?
        020000
```

The RK device interrupts twice during a drive reset. The first interrupt should be ignored.

```
000226  001474      BEQ    RTSPC          ;NO-DISMISS INTERRUPT-RK11
                                          ;WILL INTERRUPT AGAIN
                                          ;WHEN RESET COMPLETE
```

Figure C-10 RK05 Handler Listing (Cont.)

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.

```
     000230                      .FORK   RKFBLK          ;DO  RETRIES AT
                                                         ;FORK LEVEL
     000230  004577              JSR     %5,@$FKPTR
             000312
     000234  000244              .WORD   RKFBLK - .      [PIC]


     000236  105067    RKRETR:   CLRB    RETRY+1         ;YES-CLEAR RESET FLAG
             177555
     000242  000723              BR      AGAIN           ;AND RETRY OPERATION AT
                                                         ;FORK LEVEL

     000244  022527    NORMAL:   CMP     @R5,#310        ;IS THIS FIRST OF TWO
                                                         ;INTERRUPTS CAUSED BY SEEK?
             000310
```

The RK device interrupts twice for a seek operation. The first interrupt should be ignored by the handler. The seek is complete after the second interrupt has occurred.

```
     000250  001463              BEQ     RTSPC           ;YES-IGNORE IT.RK WILL
                                                         ;INTERRUPT AGAIN
                                                         ;WHEN SEEK 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, the device control and status register, is an error summary bit. If RKCS is negative, there was an error in the I/O transfer.

```
     000252  005715              TST     @R5             ;ANY ERRORS?
     000254  100067              BPL     DONE            ;NO-OPERATION COMPLETE
```

The errors are processed at fork level, priority 0.

```
     000256                      .FORK   RKFBLK          ;PROCESS ERRORS AT FORK LEVEL.
     000256  004577              JSR     %5,@$FKPTR
             000264
     000262  000216              .WORD   RKFBLK - .      [PIC]
```

Figure C-10 RK05 Handler Listing (Cont.)

The following block of code (up to the next .ENDC statement) is generated if the system supports error logging:

```
                              .IF NE ERL$G
```

Register 4 contains errors from RKER, the device error register. Unrecoverable errors that do not indicate hardware faults are not logged.

```
000264  032704            BIT    #62340,R4        ;TEST FOR USER TYPE ERRORS
        062340
000270  001031            BNE    RKERR            ;DON'T LOG THEM
                                                  ;SOFT ERROR.
```

The other types of errors are logged:

```
000272  010705            MOV    PC,R5            ;GET ADDRESS TO SAVE
                                                  ;SAVE REGISTERS
000274  062705            ADD    #RKRBUF-.,R5     [PIC]
        000214
000300  010502            MOV    R5,R2            ;SAVE ADDRESS IN R2 FOR EL
000302  012703            MOV    #RK$CSR,R3       ;R3 = ADDRESS OF
                                                  ;REGISTER TO READ
        177400
000306  012704            MOV    #RKNREG,R4       ;R4 = # OF REGISTERS TO READ
        000007
000312  012325  RKRREG:   MOV    (R3)+,(R5)+      ;MOVE REGISTERS TO BUFFER
000314  005304            DEC    R4               ;TEST IF DONE
000316  001375            BNE    RKRREG           ;NO
000320  012703            MOV    #RKNREG,R3       ;R3 = # OF REGISTERS
                                                  ;IN LOW BYTE
        000007
000324  062703            ADD    #RKRCNT,R3       ;R3 = TOTAL RETRY COUNT
                                                  ;IN HIGH BYTE
        004000
000330  016705            MOV    RKCQE,R5         ;POINT R5 AT 3RD WORD OF Q.
        177454
000334  116704            MOVB   RETRY,R4         ;SET R4=0 IN HIGH BYTE
                                                  ;FOR FORK ID
        177456
                                                  ;AND RETRY COUNT IN LOW BYTE
000340  005304            DEC    R4               ;RETRY COUNT VALUE AFTER
                                                  ;IT IS DECREMENTED
000342  004777            JSR    PC,@$ELPTR       ;CALL ERROR LOGGER.
        000172
000346  012705            MOV    #RKER,R5         ;RESET R5,R4 ON RETURN.
        177402
000352  012504            MOV    (R5)+,R4
                .ENDC
```

Figure C-10 RK05 Handler Listing (Cont.)

The next section of code retries both soft (such as checksum) and hard (hardware malfunction) errors. R5 points to RKCS, the device control and status register.

```
000354  012715  RKERR:  MOV     #1,@R5          ;YES-RESET CONTROL
        000001
```

When the controller is ready, it sets bit 7 of the low byte of RKCS.

```
000360  105715  3$:     TSTB    @R5             ;WAIT
000362  100376          BPL     3$              [loop until ready]
000364  105367          DECB    RETRY           ;DECREASE RETRY COUNT
        177426
000370  001414          BEQ     HERROR          ;NONE LEFT-HARD ERROR
000372  032704          BIT     #110000,R4      ;SEEK INCOMPLETE OR
                                                ;DRIVE ERROR?
        110000
```

Both seek incomplete and drive error require a drive reset before the operation can be retried.

```
                                                ; 100000=DRIVE ERROR
                                                ; 010000=SEEK ERROR
```

Common errors for which the I/O transfer operation should be retried are checksum errors, data late errors, and timing errors.

```
000376  001717          BEQ     RKRETR          ;NO-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. The operation code for a drive reset with interrupt enabled is 115.

```
000400  016737          MOV     DISKAD,@#RKDA   ;YES-RESELECT DRIVE
        177524
        177412
000406  012715          MOV     #115,@R5        ;START A DRIVE RESET
        000115
```

Figure C-10 RK05 Handler Listing (Cont.)

The flag in RETRY is set here so that on the next pass the handler will know that a drive reset, and not an I/O transfer, was the last operaton done.

```
    000412  052767          BIS     #100000,RETRY   ;SET FLAG
            100000
            177376
```

The next statement returns control to the monitor to wait for the drive reset or seek to finish.

```
    000420  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.

```
    000422  016705  HERROR: MOV     RKCQE,R5        ;GET POINTER TO Q ELEMENT
            177362
```

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 channel status word.

```
    000426  052755          BIS     #1,@-(R5)       ;GIVE OUR USER AN
                                                    ;ERROR IN CHANNEL
            000001
                    .IF NE  ERL$G
    000432  000411          BR      RKEXIT          ;HARD ERROR,BR TO EXIT.
```

The following section is reached after a successful transfer. Successful transfers are logged at fork level, priority 0.

```
    000434          DONE:   .FORK   RKFBLK          ;CALL ERROR LOG AT FORK
                                                    ;LEVEL FOR SUCCESS
    000434  004577          JSR     %5,@$FKPTR
            000106
    000440  000040          .WORD   RKFBLK - .
    000442  012704          MOV     #RKIDS,R4       ;SUCCESSFUL I/O,SET R4=0
                                                    ;IN HIGH BYTE FOR RK,
            000377
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                                                ;-1 IN LOW BYTE FOR SUCESS.
    000446  016705           MOV     RKCQE,R5     ;POINT R5 AT 3RD WORD OF Q.
            177336
    000452  004777           JSR     PC,@$ELPTR   ;CALL ERROR LOGGER.
            000062
                                                ;ON RETURN EXIT.
                    .IFF
                    DONE:                       [If no error logging]
                    .ENDC
    000456  005067  RKEXIT: CLR     RETRY        ;CLEAR ANY FLAGS
            177334
```

****

The handler I/O Completion Section begins here.

****

```
                    .NLIST CND
    000462                  .DRFIN  RK           ;EXIT TO COMPLETION
```

The .DRFIN macro generates the next block of code (up to the next .LIST CND directive). This section lets the monitor know that the I/O operation is complete so that the queue element can be returned to the free element list. Control returns to the monitor with the JMP statement. 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
    000462  010704           MOV     %7,%4
    000464  062704           ADD     #RKCQE-.,%4  [PIC]
                                                [Point to address of CQE]
            177324
    000470  013705           MOV     @#^054,%5    [Base of RMON]
            000054
    000474  000175           JMP     @^0270(5)    [Fixed offset in RMON]
                                                [Go to I/O completion code
                                                in the monitor]
            000270
                    .LIST CND

                    .ENDC

    000500  000000  RKFBLK: .WORD   0,0,0,0      ;FORK QUEUE BLOCK
    000502  000000
    000504  000000
    000506  000000
```

Figure C-10 RK05 Handler Listing (Cont.)

```
                   .IF NE ERL$G
    000510         RKRBUF: .BLKW   RKNREG          ;ERROR LOG STORAGE
                                                   ;FOR REGISTERS
                   .ENDC
```

```
*********************************************
```

The handler Termination Section begins here.

```
*********************************************
```

The .DREND macro generates the block of code up to the .LIST CND directive.

```
                   .NLIST CND
    000526                 .DREND  RK
            000000 ...V2=0
            000002 ...V2=...V2+2.
```

If the handler is for a system device, the bootstrap fills in the following table of pointers. Otherwise, it is filled in when the handler is made resident by .FETCH or by LOAD. The pointers are to fixed offsets in the Resident Monitor. Some of the following pointers are optional, and their assembly depends on which system conditionals are defined. See Section C.4 of this appendix for a more detailed explanation of the .DREND macro.

```
    000526  000000  $RLPTR:: .WORD  0
    000530  000000  $MPPTR:: .WORD  0
    000532  000000  $GTBYT:: .WORD  0
    000534  000000  $PTBYT:: .WORD  0
    000536  000000  $PTWRD:: .WORD  0
            000003  ...V2=...V2+1
    000540  000000  $ELPTR:: .WORD  0
            000007  ...V2=...V2+4.
    000542  000000  $TIMIT:: .WORD  0
    000544  000000  $INPTR:: .WORD  0
    000546  000000  $FKPTR:: .WORD  0
                    .GLOBL  RKSTRT
            000550' RKEND == .
    000060          .ASECT
            000060  .=60
    000060  000007          .WORD   ...V2 [Summary of SYSGEN options]
    000550          .CSECT                [Return to unnamed .PSECT]
                    .LIST CND

            000001  .END
```

Figure C-10 RK05 Handler Listing (Cont.)

The symbol table is generated at the end of the assembly listing.

```
SYMBOL TABLE

AGAIN     000112R       Q.PAR = 000016          RKRBUF   000510R
DISKAD    000130R       Q.UNIT= 000007          RKRCNT=  004000
DLSYS  =  000000 G      Q.WCNT= 000012          RKRETR   000236R
DMSYS  =  000000 G      RETRY   000016R         RKRREG   000312R
DONE      000434R       RFSYS = 000000 G        RKSTRT   000000RG
DPSYS  =  000000 G      RKBA  = 177410          RKSTS =  100000
DSSYS  =  000000 G      RKCNT = 000010          RKSYS    000006RG
DTSYS  =  000000 G      RKCQE   000010RG        RKWC  =  177406
DXSYS  =  000000 G      RKCS  = 177404          RK$CSR=  177400 G
DYSYS  =  000000 G      RKDA  = 177412          RK$VEC=  000220 G
ERL$G  =  000001        RKDS  = 177400          RTSPC    000420R
HERROR    000422R       RKDSIZ= 011300          TIM$IT=  000001
MMG$T  =  000001        RKEND = 000550RG        $ELPTR   000540RG
NORMAL    000244R       RKER  = 177402          $FKPTR   000546RG
Q.BLKN=   000004        RKERR   000354R         $GTBYT   000532RG
Q.BUFF=   000010        RKEXIT  000456R         $INPTR   000544RG
Q.COMP=   000014        RKFBLK  000500R         $MPPTR   000530RG
Q.CSW  =  000002        RKIDEN= 000000          $PTBYT   000534RG
Q.ELGH=   000024        RKIDS = 000377          $PTWRD   000536RG
Q.FUNC=   000006        RKINT   000174RG        $RLPTR   000526RG
Q.JNUM=   000007        RKLQE   000006RG        $TIMIT   000542RG
Q.LINK=   000000        RKNREG= 000007          ...V2 =  000007

. ABS.    000062     000
          000550     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1248 WORDS  ( 5 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  71 PAGES
,RK.LST/L:ME:MEB:TTM=RKCND.MAC,RK.MAC
```

Figure C-10 RK05 Handler Listing (Cont.)

C.4 System Device Handlers

The monitor and device handlers reside on the system device. The device must be block-replaceable (random access), and have read/write capability. Writing a device handler for a system device requires very little extra work once the basic device handler is written. (The RK handler in Section C.3 is a good example of a random access device handler.) The programmer simply defines the symbol $SYSDV. The system macros then expand properly, generating all the required code for a system device handler.

C.4.1 Assembling A System Device Handler

The following list shows the steps required to assemble a device handler as a system device handler.

1. The file SYCND.MAC must be edited to set the symbol $xxSYS to 1. For the RK handler, for example, the statement is as follows:

   ```
   $RKSYS = 1
   ```

2. The file SYSDEV.MAC must be included in the assembly. This file contains the single line:

   ```
   $SYSDV = 1
   ```

3. The handler, called MYFILE in this example, should be assembled together with the three system files, as shown:

   ```
   MACRO/LIST xx+SYCND+SYSDEV+MYFILE/OBJECT
   ```

   In the line above, xx represents SJ, FB, or XM. The correct macro source file for the corresponding monitor should be used. The resulting object file is MYFILE.OBJ.

   (To assemble a handler as a data device only, the SYSDEV file should be omitted.)

C.4.2 System Device Handler Requirements

The following list outlines the special requirements for a system device handler. These requirements are filled automatically by the system macros .DRBEG, .DRAST, .DRFIN, and .DREND.

1. Entry points of all current system devices (except for this handler) must be referenced in a global statement, and all must be equated to 0.

2. The handler size must be global, and must be called $SYHSZ.

3. The handler entry point must be tagged xxSYS (xx represents the device name). It must also be global. The xxSYS label is provided by the .DRBEG macro.

4. The handler must be a .PSECT named SYSHND. This .PSECT is defined by the .DRBEG macro.

5. The handler must terminate with a table of pointers to monitor routines. These global routine names are resolved when the handler is linked to the monitor, instead of being filled in by the fetch code at load time. The conditionals that are defined for the handler must match the conditionals defined for the monitor. The .DREND macro provides the table of pointers.

C.4.3 The .DRBEG and .DREND Macros

Figure C-11 shows the .DRBEG and .DREND macros. Appendix B of this manual provides complete listings of all the system macros. In Figure C-11, black ink is used for text and comments. Red ink is used for the actual source listing of the macro files.

```
.MACRO  .DRBEG  NAME,VEC,DSIZ,DSTS,VTBL

.IF NDF $SYSDV              If the handler is not for a system device, the lines
                            up to the .IFF statement are assembled.
.ASECT
. = 52
.GLOBL  NAME'END            This is global so that the handler can be broken
                            into two separately assembled modules.
                            (The RT-11 magtape handler is an example.)
                            .DRBEG can be put in the first module, and
                            .DREND can be put in the last module.

        .WORD   <NAME'END - NAME'STRT>
        .WORD   DSIZ
        .WORD   DSTS
.CSECT
.IFF                        If the handler is for a system device, the next two
                            lines are assembled.

$SYDSZ == DSIZ              This is global because it gets linked into the USR
                            for use by the .DSTATUS request.

.PSECT  SYSHND              The .PSECT is named SYSHND for the system handler.
.ENDC
NAME'STRT::
.IF B VTBL
.GLOBL  NAME'INT            This is for a device with a single vector.

        .WORD   VEC
        .WORD   NAME'INT - .
.IFF
.GLOBL  VTBL,NAME'INT       This is for a device with more than one vector.

        .WORD   <VTBL-.>/2. -1 + ^O100000
        .WORD   NAME'INT - .
.ENDC
        .WORD   ^O340
NAME'SYS::                  This is used only by a system handler.
NAME'LQE::      .WORD   0
NAME'CQE::      .WORD   0
.ENDM


.MACRO  .DREND  NAME
...V2=0                     This bit mask is an accumulation of SYSGEN options.
                            As each option is defined, a bit is added to this
                            word.

.IF NE MMG$T                (For XM handler)
...V2=...V2+2
```

Figure C-11 The .DRBEG and .DREND Macros

```
.IF DF $SYSDV
.GLOBL  $RELOC,$MPPHY,$GETBYT,$PUTBYT,$PUTWRD

$RLPTR:: .WORD  $RELOC  These pointers are for use in XM only. The system
$MPPTR:: .WORD  $MPPHY  handler must have this table with these names.
$GTBYT:: .WORD  $GETBYT The boot relocates the pointers appropriately.
$PTBYT:: .WORD  $PUTBYT
$PTWRD:: .WORD  $PUTWRD
.IFF
$RLPTR:: .WORD  0       Handlers for nonsystem devices do not need names
$MPPTR:: .WORD  0       in this table because the .FETCH code sets them
$GTBYT:: .WORD  0       up when the handler is made resident.
$PTBYT:: .WORD  0
$PTWRD:: .WORD  0
.ENDC
.ENDC                   (End of XM conditional)
.IF NE ERL$G
...V2=...V2+1
.IF DF $SYSDV
.GLOBL  $ERLOG
$ELPTR:: .WORD  $ERLOG  Pointer for error logging for system devices.
.IFF
$ELPTR:: .WORD  0       Pointer for error logging for nonsystem devices.
.ENDC
.ENDC
.IF NE TIM$IT
...V2=...V2+4
.IF DF $SYSDV
.GLOBL  $TIMIO
$TIMIT:: .WORD  $TIMIO  Pointer for time-out support for system devices.
.IFF
$TIMIT:: .WORD  0       Pointer for time-out support for nonsystem devices.
.ENDC
.ENDC
.IF DF $SYSDV
.GLOBL  $FORK,$INTEN
$INPTR: .WORD   $INTEN  Pointers for system devices.
$FKPTR:: .WORD  $FORK
.IFF
$INPTR:: .WORD  0       Pointers for nonsystem devices.
$FKPTR:: .WORD  0
.IFTF
.GLOBL  NAME'STRT       These globals allow the handler to be broken into
                        modules.
NAME'END == .
.IFT
$SYHSZ == NAME'END - NAME'STRT  This must be in all system handlers.
                                It defines the size of the handler in bytes.
```

Figure C-11 The .DRBEG and .DREND Macros (Cont.)

```
.IFF
.ASECT
.=60
        .WORD   ...V2       This is the SYSGEN options word. It is placed in
                            location 60 in block 0 of the handler. It must
                            match the SYSGEN fixed offset in RMON. It is used
                            for nonsystem handlers only.
.CSECT
.ENDC
.ENDM
```

Figure C-11 The .DBREG and .DREND Macros (Cont.)

C.5 Study of the PC Handler

Figure C-12 provides detailed comments on a listing of the PC handler. The comments do not duplicate those in the RK handler example; comments are provided only for those features that are different in the PC handler, such as multi-vectored format. Figure C-12 illustrates handler techniques for a serial, character-oriented (non-NPR) device with two vectors. The PC handler can be used for the paper tape reader alone as well as for the combined paper tape reader and punch devices.

In Figure C-12, black ink is used for text and comments. Red ink is used for the actual device handler assembly listing.

```
PC    V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 1

                        ;CONDITIONAL FILE FOR PC HANDLER EXAMPLE
                        ;
                        ;PCCND.MAC
                        ;
                        ;9/1/78 JAD
                        ;
                        ;ASSEMBLE WITH PC.MAC TO TURN ON 18-BIT I/O,
                        ;TIME-OUT SUPPORT, AND ERROR LOGGING FOR
                        ;PC HANDLER
                        ;
                000001  MMG$T   = 1             ;TURN ON 18-BIT I/O
                000001  ERL$G   = 1             ;TURN ON ERROR LOGGING
                000001  TIM$IT  = 1             ;TURN ON TIME-OUT SUPPORT

PC    V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 2

                        .TITLE PC   V03.01
                        .IDENT  /V03.01/
                        ; RT-11 HIGH SPEED PAPER TAPE PUNCH AND READER (PC11) HANDLER
                        ;
                        ; COPYRIGHT (C) 1978
                        ;
                        ; DIGITAL EQUIPMENT CORPORATION
                        ; MAYNARD, MASSACHUSETTS 01754
                        ;
                        ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY
                        ; ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY 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 EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO
                        ; AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP
                        ; OF THE SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.
                        ;
                        ; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO
                        ; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
                        ; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
                        ;
                        ; DEC ASSUMES NO RESPONSIBILITY FOR THE USE
                        ; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
                        ; WHICH IS NOT SUPPLIED BY DEC.
```

Figure C-12 PC Handler Listing

```
PC   V03.01  MACRO V03.02B12-SEP-78 15:29:52 PAGE 3

**************************************************

The device handler Preamble Section starts here.

**************************************************

                    .MCALL  .DRBEG,.FORK,.DREND,.DRAST,.DRFIN,.QELDF

                    .IIF NDF PR11$X, PR11$X=0        [0=punch and reader;
                                                       1=reader only]
                    .IIF NDF MMG$T, MMG$T=0
                    .IIF NDF ERL$G, ERL$G=0
                    .IIF NDF TIM$IT, TIM$IT=0

                    .NLIST CND
  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=^010
             000012 Q.WCNT=^012
             000014 Q.COMP=^014
             000016 Q.PAR=^016
             000024 Q.ELGH=^024
                    .LIST CND
```

The following three lines are commonly used offsets in the queue element:

```
             177776 CSTAT    = Q.CSW-Q.BLKN
             000006 BYTCNT   = Q.WCNT-Q.BLKN
             000004 BUFF     = Q.BUFF-Q.BLKN

                    ; PAPER TAPE PUNCH CONTROL REGISTERS
             000074 .IIF NDF PC$VEC, PP$VEC == 74          ;PUNCH VECTOR ADDR
                    .IIF NDF PP$CSR, PP$CSR == 177554      ;PUNCH CONTROL
                                                           ;REGISTER
             177556 PPB      = PP$CSR+2              ;PUNCH DATA BUFFER

             000000 PRDSIZ   = 0                     ;PP DEVICE SIZE (0 => NON-
                                                     ;FILE STRUCTURED)
                    .IF EQ PR11$X
             000007 PRSTS    = 7                     ;PP-PR DEVICE STATUS WORD
                    .IFF
                    PRSTS    = 40007                 ;READER ONLY
```

Figure C-12 PC Handler Listing (Cont.)

```
                .ENDC

                ; PAPER TAPE READER CONTROL REGISTERS
                .IIF NDF PR$CSR, PR$CSR == 177550       ;CONTROL REGISTER
        177552  PRB     == PR$CSR+2                     ;DATA REGISTER
                .IIF NDF PR$VEC, PR$VEC == 70           ;READER VECTOR ADDR

        000001  PRGO    = 1                     ;READER ENABLE BIT
        000101  PINT    = 101                   ;INTERRUPT ENABLE BIT
                                                ;AND GO BIT

                ; CONSTANTS FOR MONITOR COMMUNICATION
        000001  HDERR   = 1                     ;HARD ERROR BIT  [for CSW]
        020000  EOF     = 20000                 ;END OF FILE BIT  [for CSW]
```

****

The device handler Header Section begins here.

****

```
                ; LOAD POINT

                .IF EQ PR11$X                   [If both reader and punch:]
                .NLIST CND
 000000                 .DRBEG  PR,PR$VEC,PRDSIZ,PRSTS,PRTAB
```

PRTAB in the line above is the vector table.

```
000000          .ASECT
        000052  . = 52
                .GLOBL  PREND
000052  000334          .WORD   <PREND - PRSTRT>
000054  000000          .WORD   PRDSIZ
000056  000007          .WORD   PRSTS
000000          .CSECT
000000          PRSTRT::
                .GLOBL  PRTAB,PRINT
```

This references the table for a multi-vectored device:

```
000000  100034          .WORD   <PRTAB-.>/2. -1 + ^O100000
000002  000160          .WORD   PRINT - .
000004  000340          .WORD   ^O340
000006          PRSYS::
000006  000000  PRLQE:: .WORD   0
000010  000000  PRCQE:: .WORD   0
```

Figure C-12 PC Handler Listing (Cont.)

```
                  .LIST CND
                  .IFF                                  [If reader only:]
                  .NLIST CND
                          .DRBEG  PR,PR$VEC,PRDSIZ,PRSTS
                  .LIST CND
                  .ENDC

```

****

The device handler I/O Initiation Section begins here.

****

```
                  ; ENTRY POINT

   000012  016704 PP:     MOV     PRCQE,R4        ;R4 POINTS TO CURRENT Q ENTRY
           177772
   000016  006364         ASL     BYTCNT(R4)      ;WORD COUNT TO BYTE COUNT
           000006
   000022  103007         BCC     PR              ;BRANCH => READ
```

The routine for the punch:

```
                  .IF EQ PR11$X                     [Both reader and punch:]
   000024  012767         MOV     #PP$CSR,PRCSR   ;SAVE CSR FOR ABORT.
           177554
           000256
   000032  052737         BIS     #100,@#PP$CSR   ;CAUSES INTERRUPT,
                                                  ;STARTING TRANSFER
           000100
           177554
   000040  000207         RTS     PC
                  .IFF                              [Reader only:]
                          BR      PPERR           ;NO PUNCH,ERROR.
                  .ENDC

```

The routine for the reader:

```
   000042  001505 PR:     BEQ     PRDONE          ;A REQUEST FOR 0 BYTES IS
                                                  ;A SEEK, EXIT.
```

Even though a seek is not a reasonable operation for paper tape, the handler provides for it as part of RT-11's device independence.

Figure C-12 PC Handler Listing (Cont.)

```
000044  012737            MOV     #PR$CSR,PRCSR     ;SAVE CSR FOR ABORT.
        177550
        000256
000052  005737            TST     @#PR$CSR          ;IS READER READY?
        177550
000056  100000            BPL     PRGORD            ;YES, START TRANSFER
000060  052744            BIS     #EOF,@-(R4)       ;IMMEDIATE EOF IF NOT READY
        020000
000064  000474            BR      PRDONE            ;SET EOF BIT,
                                                    ;COMPLETE OPERATION

                  ; PUNCH-READER VECTOR TABLE

000066            PRTAB:
                  .IF EQ PR11$X
000066  000070            .WORD   PR$VEC            ;READER VECTOR
000070  000072            .WORD   PRINT-.           ;READER ISR OFFSET
000072  000340            .WORD   340               ;STATUS
000074  000074            .WORD   PP$VEC            ;PUNCH VECTOR
000076  000010            .WORD   PPINT-.           ;PUNCH ISR OFFSET
000100  000340            .WORD   340               ;STATUS
000102  000000            .WORD   0                 ;END OF TABLE
                  .ENDC
```

****

The device handler Asynchronous Trap Entry Section begins here.

****

```
                  ; PUNCH INTERRUPT SERVICE

                  .IF EQ PR11$X
                  .NLIST CND
000104                    .DRAST  PP,4,PRDONE
```

PRDONE is the abort entry point. An abort can be requested by any of the following means: typing double CTRL/C, issuing the .HRESET programmed request, any type of I/O error, traps to 4 and 10, and any other condition that causes a MON-F- type of fatal error message to appear. In the event that an abort is requested, is necessary to stop the device. This is not necessary for a disk, but it is important for a character-oriented device like paper tape, in order to prevent a tape runaway condition.

```
                  .GLOBL  $INPTR
000104  000454    BR      PRDONE
```

Figure C-12 PC Handler Listing (Cont.)

```
   000106  004577  PPINT:: JSR     %5,@$INPTR
           000216
   000112  000140          .WORD   ^C<4*^040>^0340
                   .LIST CND
   000114  016704          MOV     PRCQE,R4         ;R4 POINTS TO CURRENT Q ENTRY
           177670
   000120  005737          TST     @#PP$CSR         ;ERROR?
```

Bit 15 in PP$CSR is the error bit. The possible errors for paper tape devices are device out of tape, and tape jammed.

```
           177554
   000124  100412          BMI     PPERR            ;YES-PUNCH OUT OF PAPER
   000126  005764          TST     BYTCNT(R4)       ;ANY MORE CHARS TO OUTPUT?
```

The transfer is done if the required number of bytes is transferred without error.

```
           000006
   000132  001451          BEQ     PRDONE           ;NO-TRANSFER DONE
   000134  005264          INC     BYTCNT(R4)       ;DECREMENT BYTE COUNT
                                                    ;(IT IS NEGATIVE)
           000006
                   .IF EQ MMG$T
                           MOVB    @BUFF(R4),@#PPB  ;PUNCH CHARACTER
                           INC     BUFF(R4)         ;BUMP POINTER
                   .IFF
```

$GTBYT is a pointer to the monitor $GETBYT routine. See Section 1.4.4.5 of this manual for a description of the routine.

```
   000140  004777          JSR     PC,@$GTBYT       ;GET A BYTE FROM USER BUFFER
           000152
   000144  112637          MOVB    (SP)+,@#PPB      ;PUNCH IT
           177556
                   .ENDC
   000150  000207          RTS     PC
                   .ENDC
```

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.

Figure C-12 PC Handler Listing (Cont.)

```
000152  052754  PPERR:  BIS     #HDERR,@-(R4)    ;SET HARD ERROR BIT
        000001
000156  000437          BR      PRDONE           ;GO TO I/O COMPLETION

                ; READER INTERRUPT SERVICE

                .NLIST CND
000160                  .DRAST  PR,4,PRDONE      ;DEFINE AST ENTRY POINTS
                .GLOBL  $INPTR
000160  000436  BR      PRDONE
000162  004577  PRINT:: JSR     %5,@$INPTR
        000142
000166  000140          .WORD   ^C<4*^O40>^O340
                .LIST CND
000170  016704          MOV     PRCQE,R4         ;R4 POINTS TO Q ENTRY
        177614
                .IF EQ MMG$T
                        ADD     #BUFF,R4         ;POINT R4 TO BUFFER ADDRESS
                .ENDC
000174  005737          TST     @#PR$CSR         ;ANY ERRORS?
        177550
000200  100413          BMI     PREOF            ;YES-TREAT AS EOF
                .IF EQ MMG$T
                        MOVB    @#PRB,@(R4)      ;PUT CHAR IN BUFFER
                        INC     (R4)+            ;BUMP BUFFER POINTER
                        DEC     @R4              ;DECREASE BYTE COUNT
                .IFF
000202  113745          MOVB    @#PRB,-(SP)      ;GET A CHARACTER
        177552
000206  004777          JSR     PC,@$PTBYT       ;MOVE IT TO USERS BUFFER
        000105
000212  005364          DEC     BYTCNT(R4)       ;DECREASE BYTE COUNT
        000005
                .ENDC
000216  001417          BEQ     PRDONE           ;IF ZERO,WE ARE DONE
000220  052737  PRGORD: BIS     #PINT,@#PR$CSR   ;ENABLE READER INTERRUPT,
                                                 ;GET A CHARACTER
        000101
        177550
000226  000207          RTS     PC
```

Stop the device if there are errors or if the end of tape is reached:

```
000230  005037  PREOF:  CLR     @#PR$CSR         ;DISABLE INTERRUPTS
        177550
000234                  .FORK   PRFBLK           ;REQUEST SYSTEM PROCESS
000234  004577          JSR     %5,@$FKPTR
        000072
000240  000040          .WORD   PRFBLK - .
                .IF EQ MMG$T
```

Figure C-12 PC Handler Listing (Cont.)

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, etc.). 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.

```
                      PRE01:  CLRB    @(R4)             ;CLEAR REMAINDER OF BUFFER
                              INC     (R4)              ;BUMP BUFFER ADDRESS.
                              DEC     BYTCNT-BUFF(R4)   ;TEST IF DONE.
                              BNE     PRE01             ;BRANCH IF MORE.
                      .IFF
    000242  005046    PRE01:  CLR     -(SP)
    000244  004777            JSR     PC,@$PTBYT        ;CLEAR A BYTE IN USER BUFFER
            000050
    000250  005364            DEC     BYTCNT(R4)        ;DECREMENT BYTE COUNT
            000006
    000254  001372            BNE     PRE01             ;BR IF MORE
                      .ENDC

                      ; OPERATION COMPLETE
```

If the operation is complete or if it cannot complete because of an error, it is necessary to turn off the device:

```
    000256  005077    PRDONE: CLR     @PRCSR            ;TURN OFF THE READER/PUNCH
                                                        ;INTERRUPT
            000026
                                                        ;IN CASE WE GET AN ERROR LATER
                      .NLIST CND
```

****

The handler I/O Completion Section begins here.

****

```
    000262            PRFIN:  .DRFIN  PR                ;GO TO I/O COMPLETION
                      .GLOBL  PRCQE
    000262  010704            MOV     %7,%4
    000264  062704            ADD     #PRCQE-.,%4
            177524
    000270  013705            MOV     @#^O54,%5
            000054
    000274  000175            JMP     @^O270(5)
            000270
```

Figure C-12 PC Handler Listing (Cont.)

```
                    .LIST CND


    000300  000000  PRFBLK: .WORD   0,0,0,0         ;FORK QUEUE BLOCK
    000302  000000
    000304  000000
    000306  000000
    000310  000000  PRCSR:  .WORD   0               ;ADDRESS OF DEVICE TO STOP.

                    .NLIST CND
```

****

The handler Termination Section begins here.

****

```
    000312                  .DREND  PR
            000000  ...V2=0
            000002  ...V2=...V2+2.
    000312  000000  $RLPTR:: .WORD  0
    000314  000000  $MPPTR:: .WORD  0
    000316  000000  $GTBYT:: .WORD  0
    000320  000000  $PTBYT:: .WORD  0
    000322  000000  $PTWRD:: .WORD  0
            000003  ...V2=...V2+1
    000324  000000  $ELPTR:: .WORD  0
            000007  ...V2=...V2+4.
    000326  000000  $TIMIT:: .WORD  0
    000330  000000  $INPTR:: .WORD  0
    000332  000000  $FKPTR:: .WORD  0
                    .GLOBL  PRSTRT
            000334' PREND == .
    000060          .ASECT
            000060  .=60
    000060  000007          .WORD   ...V2
    000334          .CSECT
                    .LIST CND

            000001  .END
```

Figure C-12 PC Handler Listing (Cont.)

```
SYMBOL TABLE

BUFF   = 000004       PREND = 000334RG        Q.CSW = 000002
BYTCNT= 000006        PREOF   000230R         Q.ELGH= 000024
CSTAT = 177776        PREO1   000242R         Q.FUNC= 000006
EOF   = 020000        PRFBLK  000300R         Q.JNUM= 000007
ERL$G = 000001        PRFIN   000262R         Q.LINK= 000000
HDERR = 000001        PRGO  = 000001          Q.PAR = 000016
MMG$T = 000001        PRGORD  000220R         Q.UNIT= 000007
PINT  = 000101        PRINT   000162RG        Q.WCNT= 000012
PP      000012R       PRLQE   000006RG        TIM$IT= 000001
PPB   = 177556        PRSTRT  000000RG        $ELPTR  000324RG
PPERR   000152R       PRSTS = 000007          $FKPTR  000332RG
PPINT   000106RG      PRSYS   000006RG        $GTBYT  000316RG
PP$CSR= 177554 G      PRTAB   000066RG        $INPTR  000330RG
PP$VEC= 000074 G      PR$CSR= 177550 G        $MPPTR  000314RG
PR      000042R       PR$VEC= 000070 G        $PTBYT  000320RG
PRB   = 177552 G      PR11$X= 000000          $PTWRD  000322RG
PRCQE   000010RG      Q.BLKN= 000004          $RLPTR  000312RG
PRCSR   000310R       Q.BUFF= 000010          $TIMIT  000326RG
PRDONE  000256R       Q.COMP= 000014          ...V2 = 000007
PRDSIZ= 000000

. ABS.  000062    000
        000334    001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  1276 WORDS  ( 5 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  71 PAGES
,PC.LST/L:ME:MEB:TTM=PCCND.MAC,PC.MAC
```

Figure C-12 PC Handler Listing (Cont.)

C.6 RT-11 File Formats

C.6.1 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 program, one module is defined as the unit of code enclosed by the .TITLE and .END pair of MACRO directives. The module name is taken from the .TITLE statement. Object modules are produced by language processors, such as MACRO and FORTRAN, and are processed by the linker to become runnable programs (in SAV, LDA, or REL format, discussed later). Object files can also be processed by the librarian to produce library OBJ files, which are then used by the linker.

Many different object modules can be combined to form one file. Each object module remains complete and independent. However, object modules combined into a library by the librarian are no longer independent. They are concatenated and become part of the library's structure. The modules are concatenated 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 second module is added to the library behind the first module. The first byte of the second module is positioned as the high order byte of the last word of the first module. The result of this procedure is that one byte is saved in the library.

To understand byte concatenation, it is most helpful to think of the modules as a stream of bytes, rather than as a stream of 2-byte words. Figure C-13 shows how two 5-byte modules would be concatenated. Module 1 and module 2 are shown both as bytes and as words.

Module 1:

Bytes:
1
2
3
4
5

Words:	
2	1
4	3
	5

Module 2:

Bytes:
1
2
3
4
5

Words:	
2	1
4	3
	5

Concatenated modules, Module 1 followed by Module 2:

Bytes:

Module 1:	1
	2
	3
	4
	5
Module 2:	1
	2
	3
	4
	5

Words:

Module 1:	2	1
	4	3
Module 2:	1	5
	3	2
	5	4

Figure C-13 Modules Concatenated by Byte

If RT-11 is to begin execution of a program within a particular object module of a program, the information on where to start is given as the transfer address. The first even transfer address encountered by the linker is passed to RT-11 as the program's start address. Whenever the resulting program is executed the start address is used to indicate the first executable instruction. If no transfer address is given (if, for example, none is specified with the .END directive in a MACRO program) or if all are odd, the resulting program does not self-start when run.

Object modules are made up of formatted binary blocks. A formatted binary block is a sequence of 8-bit bytes (stored in an RT-11 file, on paper tape, or by some other means) and is arranged as shown in Figure C-14.

Byte containing octal value 1
Byte containing octal value 0
Low order byte of length
High order byte of length
Data bytes
Checksum byte

Figure C-14 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 data portion of each formatted binary block contains the actual object module information. 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.

If the first two bytes of a formatted binary block (the 1 and 0 bytes) are discarded, and if the checksum byte is discarded, the remainder of the block is compatible with RSX-11M formatted binary blocks. The length bytes indicate the length of the RSX binary block. RT-11 formatted binary blocks are a proper subset of the RSX binary blocks. See Appendix B, "Task Builder Data Formats", in the RSX-11M Task Builder Reference Manual, order number AA-2588D-TC, for detailed information on the many types of formatted binary blocks.

C.6.2 Library File Format (OBJ and MAC)

A library file contains concatenated modules and some additional information. RT-11 supports both object and macro libraries. Object libraries usually have an .OBJ file type; macro libraries usually have a .MAC file type. The modules in a library file are preceded by a Library Header Block and Library Directory, and are followed by the Library End Block, or trailer. Figure C-15 shows the format of a library file.

Library Header
Directory
Concatenated modules (starts on a block boundary)
Library End Trailer Block

Figure C-15 Library File Format

Diagrams of each component in the library file structure are included here. See Chapter 12 of the RT-11 System User's Guide for information on using the librarian.

C.6.2.1 **Library Header Format** - The library header describes the status of the file. There is a different header for object libraries and for macro libraries. The contents of the object library header are shown in Figure C-16. The contents of the macro library header are shown in Figure C-17.

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.

Offset	Contents	Description
0	1	Library header block code
2	42	
4	7	Librarian code
6	305	Library version number
10	0	Reserved
12		Date in RT-11 format (0 if none)
14		Time expressed in two words
16		
20	0	Reserved
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

Figure C-16 Object Library Header Format

Offset	Contents	Description
0	1001	Library type and ID code
2	305	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 C-17 Macro Library Header Format

C.6.2.2 **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 directory (see Figure C-18) is composed of 4-word entries that contain information related to all modules in the library file. Note that if the librarian /N option is used for object libraries to include module names, bit 15 of the relative block number word is set to 1. If the librarian is invoked with the keyboard monitor LIBRARY command, module names are never included.

Symbol characters 1-3 (Radix 50)	
Symbol characters 4-6 (Radix 50)	
	Block number relative to start of file
Reserved (7 bits)	Relative byte in block (9 bits)

Figure C-18 Library Directory Format

In the library directory, the symbol characters represent the entry point or 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 accomodate the exact number of modules in the library. Space for the object library 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, those not used by the directory, are zero-filled. Modules follow the directory. 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. This can be changed by 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 the default directory size is used.

Modules in libraries are concatenated by byte. (See Figure C-13 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.

C.6.2.3 **Library End Block Format** - Following all modules in the library is a specially coded Library End Block, or trailer, which signifies the end of the file (see Figure C-19).

	1	Data block header
	10	Data block length
	10	Library End Block code
	0	Reserved, must be 0
	357	Checksum byte

Figure C-19 Library End Block Format

C.6.3 Absolute Binary File Format (LDA)

The linker /L option, or the keyboard monitor LINK command /LDA option, produces output files in a paper tape compatible binary format.

Paper tape format, shown in Figure C-20, is a sequence of data blocks. 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: 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 6 bytes.

First data block:

Field	Description
1	
0	
BCL	Low order 8 bits of byte count
BCH	High order 8 bits of byte count
ADL	Low order byte of absolute load address of data bytes in the block
ADH	High order byte of load address
Data bytes . . .	
Checksum byte	

Intermediate data blocks:

Field	Description
1	
0	
BCL	This pattern is repeated for all intermediate blocks
BCH	
ADL	
ADH	
Data bytes . . .	
Checksum byte	

Last data block:

Field	Description
1	
0	
6	
0	
JL	Low byte of start address, or odd number
JH	High byte of start address, or odd number
Checksum byte	

Figure C-20 Absolute Binary Format (LDA)

LDA format files are used 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 usual procedure for loading a program that will execute in a stand-alone environment is as follows:

1. Toggle the BIN loader into memory.
2. Load the Absolute Loader into memory.
3. Load the LDA file into memory and begin execution.

LSI computer systems have console microcode that makes steps 1 and 2 above unnecessary.

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.

C.6.4 Save Image File Format (SAV)

Save image format is used for programs that are to be run in the SJ environment, or in the background in the FB and XM environments. Save image files normally have a .SAV file type. This format is essentially an image of the program as it would appear in memory. (Block 0 of the file corresponds to memory locations 0-776, block 1 to locations 1000-1776, and so forth.) See Table C-2 for the contents of block 0. See also Section 11.5.2 of the RT-11 System User's Guide for more information on the load modules created by the linker.

Table C-2
Information in Block 0

Offset	Contents
0	Reserved
2	Reserved
4	Reserved
6	Reserved
10	Reserved
12	Reserved
14	XM BPT trap (XM only)
16	XM BPT trap (XM only)
20	XM IOT trap (XM only)
22	XM IOT trap (XM only)
24	Reserved

(continued on next page)

Table C-2 (Cont.)
Information in Block 0

Offset	Contents
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 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	Reserved
66	Reserved
.	Reserved
.	Reserved
.	Reserved
360-377	Bitmap area

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. This information is used by the monitor when loading the program.

The keyboard monitor commands R and RUN cause a program stored in a SAV file to be loaded and started. (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. Locations that are protected, such as location 54 and the device interrupt vectors, are not loaded. The locations that are not protected are loaded 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, the corresponding block of the SAV file is loaded 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 transfer control to RMON. The parts of the program, if any, that are stored in SWAP.SYS are read 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 into memory. (The R command does not use SWAP.SYS.) The monitor keeps track of the fact that KMON and USR are swapped out, and execution of the program begins.

C.6.5 **Relocatable File Format (REL)**

A foreground job is linked using 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.

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 users to use .ASECT directives to store information in locations 0 through 777 in REL files. All global references have been resolved. The REL file is not relocated at link time; relocation information is included 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.

C.6.5.1 **REL Files without Overlays** - A REL file for a program without overlays appears as shown in Figure C-21.

Block 0	Program text	Relocation information

Figure C-21 REL File Without Overlays

Block 0 (relative to the start of the file) contains the information shown in Table C-2. 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. Block 0 of the file is read into an internal monitor buffer.
2. The amount of memory required for the job is obtained from location 52 of block 0 of the file, and the space in memory is allocated by moving KMON and the USR down.
3. The program text is read into the allocated space.
4. The relocation information is read into an internal buffer.
5. The locations indicated in the relocation information area are relocated 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. For each relative address in the list, the monitor 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 C-22.

15	14 ... 0
	Relative word offset
Original contents	
	Relative word offset
Original contents	
.	
.	
	.
.	
-2	

Figure C-22 Relocation Information Format

In Figure C-22, bits 0-14 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 is used to indicate 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. The value 177776, or -2, terminates the list of relocation information. The original contents is a full 16-bit word.

C.6.5.2 **REL Files with Overlays** - When overlays are included in a program, the file is similar to that of a nonoverlaid program. However, in addition to the root segment, the overlay segments must also be relocated. 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 an overlaid file is run with FRUN. The relocation information for overlay 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 resident in memory. Figure C-23 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 the overlay region. This size is added to the size of the program base segment (in location 52) to allocate space for the job.

After the program base (root) code has been relocated, each existing overlay is read into the program overlay region in memory, relocated using the overlay relocation information, and then written 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 the same as in the 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.

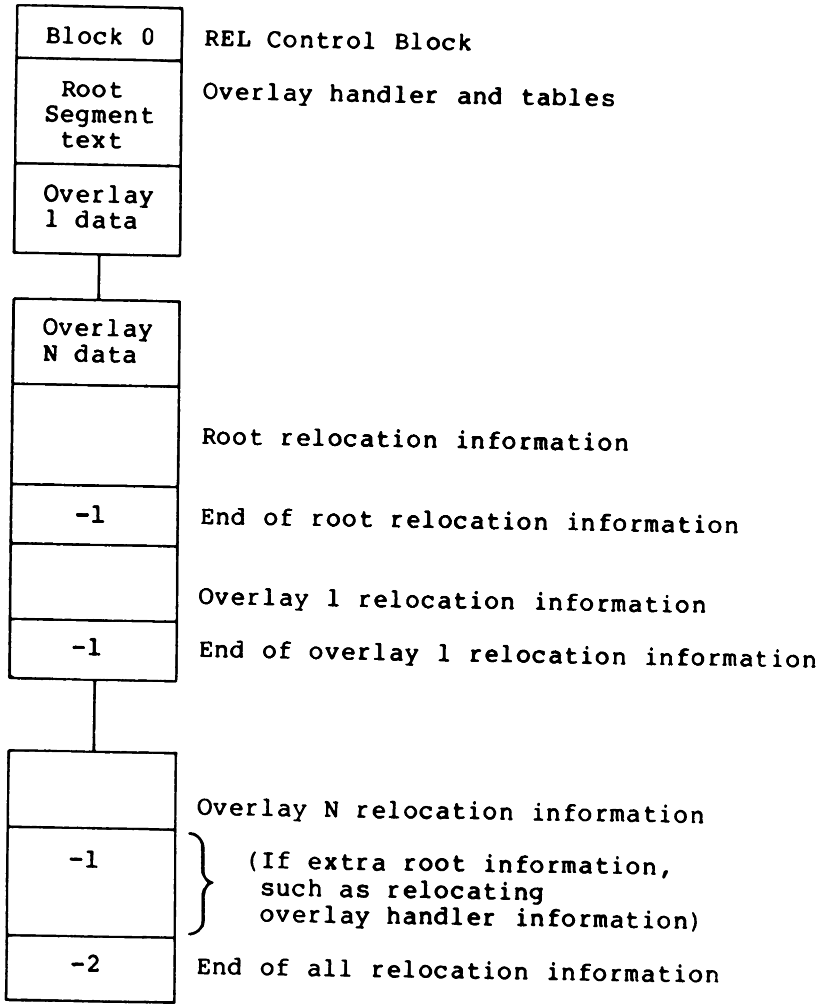


Figure C-23 REL File with Overlays

C.7 The Device Directory

The device directory begins at physical block 6 of any directory-structured device and consists of a series of directory segments that contain the names and lengths of the files on that device. The directory area is variable in length, from 1 to 31 (decimal) directory segments. DUP allows specification of the number of segments when the directory is initialized. The default value varies from device to device. See Chapter 8 of the RT-11 System User's Guide for a table of the default directory segments. Each directory segment is made up of two physical blocks. Thus, a single directory segment is 512 words, or 1024 bytes in length. Figure C-24 shows the general format of the device directory.

5 header words
file entries . . .
End of segment marker

Figure C-24 Device Directory Format

C.7.1 RT-11 File Storage

It is important for users to understand how RT-11 stores files on a device. All RT-11 files must reside on blocks that are contiguous on the device. Because the blocks are located in order, one after the other, the overhead of having pointers in each block to the next block is eliminated. Figure C-25 shows a simplified diagram of a file-structured device with two files stored on it.

file 80 blocks	empty 150 blocks	file 119 blocks	...

Figure C-25 File-Structured Device

When a file is created in RT-11, the size for the file must be allocated in the .ENTER programmed request. If the actual size is not known, as is often the case, the size allocated should be large enough to accomodate all the data possible. There are two special cases for the .ENTER request. A length argument of 0 allocates for the file either one-half the largest space available, or the second largest space, whichever is bigger. A length argument of -1 allocates the largest space possible on the device.

A tentative entry is then created on the device with the length allocated. The tentative entry is always followed by an empty entry. This is in order to account for unused space if the actual data written to the file is smaller than the size originally allocated. Figure C-26 shows an example of a tentative entry whose allocated size is 100 blocks.

file 80 blocks	tentative 100 blocks	empty 50 blocks	file 119 blocks

Figure C-26 Tentative Entry

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 C-27. Note that every tentative entry must be followed by an empty entry.

file 80 blocks	tentative 100 blocks	empty 0 blocks	tentative 25 blocks	empty 25 blocks	file 119 blocks

Figure C-27 Two Tentative Entries

When a program finishes writing data to the device, it closes the tentative file with the .CLOSE programmed request. The tentative entry is made permanent. Its length is the actual size of the data that was written. The size of the empty entry is its original size plus the difference between the tentative file size and the permanent file size.

Figure C-28 shows the same example after both tentative files were closed. The first file's actual length is 75 blocks, and the second file's length is 10 blocks. Note that the total number of blocks associated with entries in Figure C-28, including empty entries, is equal to the total number of blocks in Figure C-26.

file 80 blocks	permanent 75 blocks	empty 25 blocks	permanent 10 blocks	empty 40 blocks	file 119 blocks

Figure C-28 Permanent Entries

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 bigger from within a program, it is necessary to read the existing file, allocate a new, larger tentative entry, and then write both the old and the new data to the new file. The old file can then be deleted.

The DUP utility program provides an easy way to extend the size of an existing file. The /T option does this, providing that there exists an empty entry with sufficient space in it immediately after the data file.

C.7.2 Directory Header Format

Each directory segment contains a 5-word header, leaving 507 (decimal) words for directory entries. The contents of the header words are described in Table C-3.

Table C-3
Directory Header Words

Word	Contents
1	The number of segments available for entries. This number can be given to DUP when the device is initialized and must be in the range from 1 to 31 (decimal). Or, DUP can use the default value for the device.
2	Segment number of the next logical directory segment. The directory is a linked list of segments. This word is the link word between logically contiguous segments; if it is equal to 0, there are no more segments in the list. See Section C.7.4 for more details.
3	The highest segment currently open (each time a new segment is created, this number is incremented). This word is updated only in the first segment and is unused in any but the first segment.
4	The number of extra bytes per directory entry. This number can be specified when the device is initialized with DUP. Currently, RT-11 does not allow direct manipulation of information in the extra bytes.
5	Block number on the device where entries (files, tentatives, or empties) in this segment begin.

C.7.3 Directory Entry Format

The remainder of the segment is filled with directory entries. An entry has the format shown in Figure C-29.

Status word	
Name (chars 1-3) (in Radix-50)	
Name (chars 4-6) (in Radix-50)	
File type (1 to 3 characters) (in Radix-50)	
Total file length	
Job #	Channel #
Date	
Optional extra words	
. . .	

Figure C-29 Directory Entry Format

C.7.3.1 **Status Word** - The status word is broken down into two bytes of data, as shown in Figure C-30.

Type of entry	Reserved

Figure C-30 Status Word

Table C-4 lists the valid entry types.

Table C-4
Entry Types

Value	Type of Entry
1	Tentative file. (One that has been .ENTERed but not .CLOSEd.) Files of this type are deleted if not eventually .CLOSEd and are listed by DIR as <UNUSED> files.
2	An empty file. The name, file type, and date fields are not used. DIR lists an empty file as <UNUSED> followed by the length of the unused area.
4	A permanent entry. A tentative file that has been .CLOSEd is a permanent file. The name of a permanent file is unique; there can be only one file with a given name and file type. If another exists before the .CLOSE is done, it is deleted by the monitor as part of the .CLOSE operation.
10	End-of-segment marker. RT-11 uses this to determine when the end of the directory segment has been reached during a directory search.

Note that an end-of-segment marker can appear as the 512th word of a segment. It does not have to be followed by a name, type, or other data.

C.7.3.2 **Name and File Type** - These three words, in Radix-50, contain the symbolic name and file type assigned to a file. These words are usually unused for empty entries. However, the DIR utility program /Q option (or the keyboard monitor command DIRECTORY with the /DELETED option) lists the names and file types of deleted files.

C.7.3.3 **Total File Length** - The file length consists of the number of blocks taken up by the entry. Attempts to read or write outside the limits of the file result in an end-of-file error.

C.7.3.4 **Job Number and Channel Number** - A tentative file is associated with a job in one of two ways:

1. In the SJ environment, 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 entry for the channel when the .CLOSE is given. The channel number is loaded into the even byte of the sixth word.

2. In the FB and XM environments, the channel number is put into the even byte of the sixth word. In addition, the number of the job that is opening the file is put into the odd byte of the sixth word. The job number is required to uniquely identify the correct tentative file during the .CLOSE operation. It is also necessary because both jobs can have files open on their respective channels. The job number (0 for background, 2 for foreground) differentiates the tentative files.

NOTE

This sixth word (job number and channel number word) is used only when the file is marked as tentative. Once the entry becomes permanent, the word becomes unused. The function of the sixth word while the entry is permanent permanent is reserved for future use by DIGITAL software.

C.7.3.5 **Date** - When a tentative file is created by means of .ENTER, the system date word is put into the creation date slot for the file. The date word format is shown in Figure C-31. Bit 15 is reserved for future use by DIGITAL. This word is 0 if no date has been entered with the DATE keyboard monitor command.

15	14 13 12 10	9 8 7 6 5	4 3 2 1 0
	Month (1-12) (decimal)	Day (1-31) (decimal)	Year - 110 (octal)

Figure C-31 Date Word

C.7.3.6 **Extra Words** - The number of extra words is determined by specifying an option to DUP at initialization time. This choice is reflected by the number of extra bytes per entry in the header words. Although DUP provides for allocation of extra words, RT-11 provides no direct facility for manipulating this extra information. Any user program that needs to access these words must perform its own direct operations on the RT-11 directory.

Figure C-32 illustrates a typical RT-11 directory segment.

Header block:	4	Four segments available
	0	No next segment
	1	Highest open is #1
	0	No extra words per entry
	16	Files start at block 16 (octal)
File entries:	2000	Permanent entry
	71105	Radix-50 for RKM
	54162	Radix-50 for NFB
	75273	Radix-50 for SYS
	42	File is 42 (octal) blocks long (34 decimal)
	0	Used only for tentative entries
	0	No creation date
	1000	An empty entry
	0	(The name and file type of an
	0	empty entry are not significant.)
	0	
	100	100 (octal) blocks long (64 decimal)
	0	Used only for tentative entries
	0	No creation date
	2000	Permanent entry
	62570	Radix-50 for PIP
	0	Radix-50 for spaces
	50553	Radix-50 for MAC
	11	11 (octal) blocks long (9 decimal)
	0	Used only for tentative entries
	0	No creation date
	400	Tentative file on channel 1
	62570	Radix-50 for PIP
	0	Radix-50 for spaces
	50553	Radix-50 for MAC
	20	20 (octal) blocks long (16 decimal)
	1	Job 0 (BG); channel 1
	0	No creation date
	1000	(Every tentative entry must be
	0	followed by an empty entry.)
	0	
	0	
	1020	1020 (octal) blocks long (528 decimal)
	0	Used only for tentative entries
	0	No creation date
	4000	End of directory segment

Figure C-32 RT-11 Directory Segment

When the tentative file PIP.MAC is closed by the .CLOSE programmed request, the permanent file PIP.MAC is deleted.

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 given in the fifth word of that directory segment and add to it the length of each file in the directory before the desired file. For example, in Figure C-32, the permanent file PIP.MAC will begin at block number 160 (octal).

C.7.4 Size and Number of Files

The number of files that can be stored on an RT-11 device depends on the number of segments in the device's directory and the number of extra words per entry. The maximum number of directory segments on any RT-11 device is 31 (decimal). The following formula can be used to calculate the theoretical maximum number of directory entries.

$$31 * \frac{512-6}{7 + N} - 2$$

In the formula shown above,

N equals the number of extra information words per entry. If N is 0, the maximum is 2232 (decimal) entries.

Note that all divisions are integer. That is, the remainder should be discarded. No cancelling is valid.

In the formula shown above, the -2 is required for two reasons. First, in order to enter a file, the tentative entry must be followed by an empty entry. 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 files are added sequentially (that is, one immediately after another) without deleting any files, roughly one-half the total number of entries will fit on the device before a directory overflow occurs. This situation results from the way filled directory segments are handled.

When a directory segment becomes full and it is necessary to open a new segment, approximately one half the entries of the filled segment are moved to the newly-opened segment. Thus, when the final segment is full, all previous segments have approximately one half their total capacity.

If files are continually added to a device and the SQUEEZE keyboard monitor command is not issued, the maximum number of entries can be computed from the following formula:

$$(M-1) * \frac{S}{2} + S$$

In the formula shown above,

M equals the number of directory segments

S can be computed from the following formula:

$$S = \frac{512 - 5}{7 + N} - 2$$

N equals the number of extra information words per entry.

The theoretical total of directory entries (see the first formula, above) can be realized by compressing the device (by 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.

C.7.5 Directory Segment Extensions

RT-11 allows a maximum of 31 (decimal) directory segments. This section covers the processing of a directory segment. For illustrative purposes, the following symbols are used:

```
n !.    This represents a directory segment with some
  !     directory entries. The segment number is shown as n.
  !
  !
```

```
n !.    This represents a segment that is full. That is, no more
  !.    entries will fit in the segment.
  !.
  !.
```

The directory starts out with entries entered into segment 1:

```
1 !.
  !
  !
  !
```

As entries are added, segment 1 fills:

```
1 !.
  !.
  !.
  !.
```

When this occurs and an attempt is made to add another entry to the directory, the system must open another directory segment. If another segment is available, the following occurs:

1. One half of the entries from the filled segment are put into the next available segment and the header words of the new segment are filled with the correct information.

2. The shortened segment is rewritten to the disk.

3. The directory segment links are set.

4. The file is entered in either the shortened or the newly created segment, depending on which segment has the an emtpy entry of the required size.

NOTE

If the last segment becomes full and an attempt is made to enter another file, a fatal error occurs and an error message is generated.

Thus, in the normal case, the segment appears as:

```
       1  !.   Before extension. Segment 1 is full.
          !.
          !.
          !.

       1  !.   After extension. Half the entries are in the
          !.   new segment, half in the old. Segment 1 is
          !    linked to segment 2.
  1       !
Links
to 2
       2  !.
          !.
          !
          !
```

If many more files are entered, they fill up the second segment and overflow into the third segment, if it is available.

```
       1  !.
          !.
          !
  1       !
Links
to 2
       2  !.
          !.
          !
  2       !
Links
to 3
       3  !.
          !.
          !
          !
```

In this case, the segments are contiguous. However, the links between them are still required by the USR. The links are also required when the segments are not contiguous. For example, if a large file were deleted from segment 2 and many small files were entered, it would then be possible to overflow segment 2 again. If this occurred and a fourth segment existed, the directory would appear as follows:

```
       1  !.   In this case, segment 2 overflows into
          !.   segment 4 and the links are used to link
  1       !    logical pieces rather than physical
Links          pieces.
to 2
       2  !.
          !.
          !
  2       !
Links
to 4
       3  !.
          !.
          !
  4       !
Links
to 3
       4  !.   Segment 4 is linked to segment 3 because segment
          !.   2 was previously linked to 3.
          !
          !
```

C.8 Magtape Structure

This section covers the magtape file structure as implemented in RT-11 V03 and V03B. RT-11 V03 and V03B can read magtapes created under RT-11 V02C. RT-11 magtapes use a subset of the VOL1, HDR1, and EOF1 ANSI standard labels. RT-11 automatically writes magtapes with ANSI standard labels. 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 over onto 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 that the hardware magtape handler (as opposed to the file-structured magtape handler) can read data in any format at all. Or, users can make use of .SPFUN programmed requests and the file-structured magtape handler to read tapes whose data is in a non-standard format. 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 provides no volume protection by checking access fields.

In the diagrams shown below, an asterisk (*) represents a tape mark. 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). Programmers should consult the hardware manual for their particular tape devices if the format of the tape mark is important to them.

A file stored on magtape has the following structure:

```
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, the file MBOOT.BOT must be used to copy the primary bootstrap. The primary bootstrap is represented by BOOT in the diagram above. 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, the file MBOOT.BOT should be copied to the tape, as a file. (This is in addition to copying it into the boot block at the beginning of the tape.) Instructions for building bootable magtapes are in the RT-11 System Generation Manual.

Each label on the tape, as shown in the diagrams above, 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 C-5 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 C-5 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

Table C-5
ANSI Magtape Labels in RT-11

Volume Header Label (VOL1)			
CP	Field Name	L	Content
1-3	Label identifier	3	VOL
4	Label number	1	1
5-10	Volume identifier	6	Volume Label. If no volume ID is specified by the user 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 CP40-50 = Owner Name. Maximum is ten 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)			
CP	Field Name	L	Content

(continued on next page)

Table C-5 (Cont.)
ANSI Magtape Labels in RT-11

1-3	Label identifier	3	HDR
4	Label number	1	1
5-21	File identifier	17	The 6-character ASCII file name (spaces can be used to pad the file name to six characters; the dot can be written without the padding), dot, 3-character file type. 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:			
CP	Field Name	L	Content
1-3	Label identifier	3	EOF
55-60	Block count	6	Number of data blocks since the preceding HDR1 label, unless a .SPFUN operation is done. If .SPFUNs are issued, the block count is 0. However, if only 256-word .SPFUN writes are done, block count is accurate.

C.9 Cassette Structure

A blank, newly initialized TU60 cassette appears in the format shown in Figure C-33.

Clear leader	Extended file gap	Sentinel file	Unpredictable information

32 bytes (decimal)

Figure C-33 Initialized Cassette Format

A cassette with a file on it appears as shown in Figure C-34.

Clear leader	Extended file gap	Header block	Block gap	Data block	Block gap	Data block	File gap	Sentinel file
		32 bytes (decimal)		128 bytes (decimal)				

Figure C-34 Cassette With Data

Files normally have data written in 128-byte (decimal) blocks. This can be altered by writing cassettes while in hardware mode. In hardware mode, the user program must handle the processing of any headers and sentinel files. In software mode, the handler automatically does this.

Figure C-34 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 C-35.

Block gap	Data block	Block gap	Clear trailer

or as:

Block gap	Data block	Block gap	Data block	Clear trailer
			(Partially written)	

Figure C-35 Physical End of Cassette

In the latter case, 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 C-6. The data in Table C-6 is binary (that is, 0 equals a byte of 0) unless it is specified to be ASCII.

Table C-6
Cassette File Header Format

Byte Number	Contents
0-5	File name in ASCII characters (ASCII is assumed to imply a 7-bit code).
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 multi-volume file; successive numbers are used for continuations.
13	Level 1; this byte is a 1. This byte must be changed to 0 if CAPS-11 will be used to load files. See the RT-11 System Generation Manual for details.
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
29-31	Reserved for user

INDEX

RT-11 Advanced
Programmers Guide
AD-5280B-T3

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 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____________

Street____________

City____________ State____________ Zip Code or Country____________

Please cut along this line.

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.