# RT–11 System Macro Library Manual

Order Number AA–PD6LA–TC

**August 1991**

This manual contains current reference data about system macros used to call routines in the RT–11 Monitor that performs program requests.

**Revision/Update Information:** This information was previously published, along with reference data about the system subroutine library, as part of the *RT–11 Programmer's Reference Manual,* AA–H378E–TC.

**Operating System:** RT–11 Version 5.6

**Digital Equipment Corporation**
**Maynard, Massachusetts**

**First Printing, August 1991**



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



S1435

This document was prepared using VAX DOCUMENT, Version 1.2.

# Contents


## Preface ix

## Chapter 1 Introduction to Advanced RT–11 Programming

1.1 Programmed Requests . . . 1–1
1.1.1 Operating System Features . . . 1–2
1.1.1.1 RT–11 Monitors . . . 1–2
1.1.1.2 System Job Environment . . . 1–4
1.1.1.3 Multiterminal Operation . . . 1–4
1.1.1.4 System Communication Areas . . . 1–4
1.2 Programmed Request Implementation . . . 1–5
1.2.1 EMT Instructions . . . 1–5
1.2.2 System Control Path Flow . . . 1–6
1.3 System Conventions . . . 1–7
1.3.1 Program Request Format . . . 1–8
1.3.2 Blank Arguments . . . 1–11
1.3.3 Addressing Modes . . . 1–12
1.3.4 Keyword Macro Arguments . . . 1–13
1.3.5 Channels and Channel Numbers . . . 1–14
1.3.6 Device Blocks . . . 1–14
1.3.7 Programmed Request Errors . . . 1–15
1.3.8 User Service Routine (USR) Requirement . . . 1–15
1.4 Using Programmed Requests . . . 1–18
1.4.1 Initialization and Control . . . 1–18
1.4.2 Examining System Information and Reporting Status . . . 1–21
1.4.3 Command Interpretation . . . 1–21
1.4.4 File Operations . . . 1–22
1.4.5 Input/Output Operations . . . 1–24
1.4.5.1 Completion Routines . . . 1–24
1.4.5.2 Multiterminal Requests . . . 1–26
1.4.6 Foreground/Background Communications . . . 1–26
1.4.7 Timer Support . . . 1–27
1.4.8 Program Termination or Suspension . . . 1–28
1.4.9 Job Communications . . . 1–29
1.4.10 Mapped and Unmapped Regions . . . 1–29
1.4.11 Extended Memory Functions . . . 1–29
1.4.12 Interrupt Service Routines . . . 1–31
1.4.13 Device Handlers . . . 1–31
1.4.14 Logical Name Translation Bypass . . . 1–32

1.4.15 Consistency Checking . . . . . . . . . . 1–33
1.5 Programmed Request Summary . . . . . . . . . . 1–33

## Chapter 2 Programmed Request Description and Examples

.ABTIO . . . . . . . . . . 2–3
.ADDR . . . . . . . . . . 2–4
.ASSUME . . . . . . . . . . 2–5
.BR . . . . . . . . . . 2–6
.CALLK . . . . . . . . . . 2–7
.CALLS . . . . . . . . . . 2–9
.CDFN . . . . . . . . . . 2–11
.CHAIN . . . . . . . . . . 2–13
.CHCOPY . . . . . . . . . . 2–16
.CKXX . . . . . . . . . . 2–19
.CLOSE . . . . . . . . . . 2–22
.CLOSZ . . . . . . . . . . 2–24
.CMAP/.CMPDF/.GCMAP . . . . . . . . . . 2–26
.CMKT . . . . . . . . . . 2–32
.CNTXSW . . . . . . . . . . 2–34
.CRAW . . . . . . . . . . 2–36
.CRRG . . . . . . . . . . 2–40
.CSIGEN . . . . . . . . . . 2–41
.CSISPC . . . . . . . . . . 2–47
.CSTAT . . . . . . . . . . 2–51
.CTIMIO . . . . . . . . . . 2–53
.DATE . . . . . . . . . . 2–55
.DEBUG/.DPRINT . . . . . . . . . . 2–57
.DELETE . . . . . . . . . . 2–60
.DEVICE . . . . . . . . . . 2–61
.DRAST . . . . . . . . . . 2–64
.DRBEG . . . . . . . . . . 2–67
.DRBOT . . . . . . . . . . 2–69
.DRDEF . . . . . . . . . . 2–70
.DREND . . . . . . . . . . 2–75
.DREST . . . . . . . . . . 2–77
.DRFIN . . . . . . . . . . 2–83
.DRINS . . . . . . . . . . 2–84
.DRPTR . . . . . . . . . . 2–86
.DRSET . . . . . . . . . . 2–88
.DRSPF . . . . . . . . . . 2–89
.DRTAB . . . . . . . . . . 2–93
.DRUSE . . . . . . . . . . 2–95
.DRVTB . . . . . . . . . . 2–97
.DSTAT . . . . . . . . . . 2–98

.ELAW . . . . . . 2–100
.ELRG . . . . . . 2–101
.ENTER . . . . . . 2–102
.EXIT . . . . . . 2–106
.FETCH/.RELEAS . . . . . . 2–108
.FORK . . . . . . 2–110
.FPROT . . . . . . 2–112
.GCMAP . . . . . . 2–114
.GFDAT . . . . . . 2–115
.GFINF . . . . . . 2–117
.GFSTA . . . . . . 2–120
.GMCX . . . . . . 2–123
.GTIM . . . . . . 2–124
.GTJB . . . . . . 2–126
.GTLIN . . . . . . 2–129
.GVAL/.PVAL . . . . . . 2–132
.HERR/.SERR . . . . . . 2–136
.HRESET . . . . . . 2–141
.INTEN . . . . . . 2–142
.LOCK/.UNLOCK . . . . . . 2–144
.LOOKUP . . . . . . 2–147
.MAP/.UNMAP . . . . . . 2–153
.MFPS/.MTPS . . . . . . 2–155
.MRKT . . . . . . 2–158
.MSDS . . . . . . 2–161
.MTATCH . . . . . . 2–162
.MTDTCH . . . . . . 2–165
.MTGET . . . . . . 2–167
.MTIN . . . . . . 2–171
.MTOUT . . . . . . 2–173
.MTPRNT . . . . . . 2–175
.MTPS . . . . . . 2–176
.MTRCTO . . . . . . 2–177
.MTSET . . . . . . 2–178
.MTSTAT . . . . . . 2–180
.MWAIT . . . . . . 2–181
.PEEK . . . . . . 2–183
.POKE . . . . . . 2–185
.PRINT . . . . . . 2–187
.PROTECT/.UNPROTECT . . . . . . 2–188
.PURGE . . . . . . 2–191
.PVAL . . . . . . 2–192
.QELDF . . . . . . 2–193
.QSET . . . . . . 2–194
.RCTRLO . . . . . . 2–196

.RCVD/.RCVDC/.RCVDW . . . . . . . . . . 2–197
.RDBBK . . . . . . . . . . 2–205
.RDBDF . . . . . . . . . . 2–206
.READ/.READC/.READW . . . . . . . . . . 2–207
.RELEAS . . . . . . . . . . 2–218
.RENAME . . . . . . . . . . 2–219
.REOPEN . . . . . . . . . . 2–221
.RSUM . . . . . . . . . . 2–222
.SAVESTATUS . . . . . . . . . . 2–223
.SCCA . . . . . . . . . . 2–226
.SDAT/.SDATC/.SDATW . . . . . . . . . . 2–228
.SDTTM . . . . . . . . . . 2–236
.SERR . . . . . . . . . . 2–238
.SETTOP . . . . . . . . . . 2–239
.SFDAT . . . . . . . . . . 2–242
.SFINF . . . . . . . . . . 2–243
.SFPA . . . . . . . . . . 2–246
.SFSTAT . . . . . . . . . . 2–248
SOB . . . . . . . . . . 2–253
.SPCPS . . . . . . . . . . 2–254
.SPFUN . . . . . . . . . . 2–257
.SPND/.RSUM . . . . . . . . . . 2–261
.SRESET . . . . . . . . . . 2–263
.SYNCH . . . . . . . . . . 2–264
.TIMIO . . . . . . . . . . 2–267
.TLOCK . . . . . . . . . . 2–269
.TRPSET . . . . . . . . . . 2–271
.TTYIN/.TTINR . . . . . . . . . . 2–273
.TTYOUT/.TTOUTR . . . . . . . . . . 2–275
.TWAIT . . . . . . . . . . 2–278
.UNLOCK, .UNMAP, .UNPROTECT . . . . . . . . . . 2–280
.WAIT . . . . . . . . . . 2–281
.WDBBK . . . . . . . . . . 2–282
.WDBDF . . . . . . . . . . 2–283
.WRITE/.WRITC/.WRITW . . . . . . . . . . 2–284

Appendix A Summary of Added and Changed Functionality

Index

## Figures


| | | |
|---|---|---|
| 1–1 | RT–11 Monitors | 1–3 |
| 1–2 | System Flow During Programmed Request Execution | 1–7 |
| 1–3 | EMT 374 Argument | 1–9 |
| 1–4 | Stack Set by .CSIGEN Programmed Request | 1–10 |
| 1–5 | EMT 375 Argument Block | 1–10 |


## Tables


| | | |
|---|---|---|
| 1–1 | EMT Codes | 1–6 |
| 1–2 | Programmed Requests Requiring the USR | 1–16 |
| 1–3 | Values Used with .MRKT/.CMKT, .TIMIO/.CTIMIO | 1–28 |
| 1–4 | Programmed Requests for RT–11 Environments | 1–33 |
| 1–5 | Multijob or Mapped Program Requests | 1–40 |
| 2–1 | Change Mapping Context (I.CMAP) Word Bits | 2–27 |
| 2–2 | Timer Block Format | 2–53 |
| 2–3 | Device-Identifier Byte Values | 2–71 |
| 2–4 | System Service | 2–75 |
| 2–5 | Special Functions for the TYPE=n Parameter | 2–90 |
| 2–6 | Soft Error Codes (.SERR) | 2–138 |
| A–1 | Summary of Added and Changed Functionality | A–1 |

# Preface

## Description of the Manual

This manual provides reference data about system macros used to call routines in the RT–11 Monitor that perform program requests. These system macros are defined in system macro library SYSMAC.SML stored on the system volume.

Reference data about the system subroutine library is now contained in a separate manual, *RT–11 System Subroutine Library Manual*. See *Associated Documents*.

## Document Structure

***Chapter 1 — Introduction to RT–11 Programming***
Describes the effective use of programmed requests and subroutines in RT–11 programs; provides examples that demonstrate their flexibility and value in working programs.

***Chapter 2 — Programmed Request Description and Examples***
Programmed requests arranged in alphabetical sequence; detailed description and example of its use in a program; reference to related programmed requests.

***Appendix A — Summary of Added and Changed Functionality***
Lists program requests implemented in previous major releases of RT–11. Summarizes changes made to the program requests from version to version.

## Intended Audience

This information is intended for use of advanced RT–11 MACRO–11 assembly language programmers.

## Associated Documents

The RT–11 Documentation Set consists of the following associated documents:

Basic Books

- *Introduction to RT–11*
- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*
- *RT–11 Commands Manual*

- *RT–11 Quick Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

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

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 System Internals Manual*
- *RT–11 Device Handlers Manual*
- *RT–11 Volume and File Formats Manual*
- *DBG–11 Symbolic Debugger User's Guide*

## Conventions

The following conventions are used in this manual:

| Convention | Meaning |
|---|---|
| UPPERCASE characters | In programmed request examples, uppercase characters representing the MACRO element of the command should be entered exactly as given. |
| Lowercase characters | In programmed request syntax examples, lowercase characters represent arguments to the MACRO element of the command for which you provide a value. For example:<br>.MTDTCH area,unit |
| Black print | In examples, black print indicates output lines or prompting characters that the system displays. |
| [ ] | Square brackets in a command string indicates optional parameters, qualifiers, or values. |
| RET | RET in examples represents the RETURN key. |
| CTRL/*x* | CTRL/*x* indicates a control key sequence. While pressing the key labeled Ctrl, press another key. For example: CTRL/C |

# Chapter 1
# Introduction to Advanced RT–11 Programming

This chapter describes programmed requests and subroutines and recommends how to use them effectively in your programs. Examples are provided to demonstrate their flexibility and value in working programs. Programmed requests and system subroutines, available as part of the RT–11 operating system, aid you in writing reliable and efficient programs and provide a number of services to application programs. These requests call routines in the RT–11 monitor that perform these services. System macros are defined in SYSMAC.SML, a system macro library stored on the system volume. The library also contains macro routines you use to write device handlers and interrupt service routines.

Although the SYSMAC.MAC file is not provided on the RT–11 distribution kit, you will need this file if you want to modify the system macro library. Create SYSMAC.MAC from the distributed file SYSMAC.SML by running the SPLIT utility. Type the following CCL command to create the file SYSMAC.MAC on your default device.

```
.SPLIT  ,SYSMAC.MAC=SYSMAC.SML/B:..SYSM
```

The variable ..SYSM represents the boundary along which to split SYSMAC.SML. Refer to the file CUSTOM.TXT on your distribution kit for the value to substitute for ..SYSM in the command line.

If you are a FORTRAN programmer, you can access the RT–11 monitor services through calls to routines in system subroutine library SYSLIB.OBJ, stored on the system volume. A character string manipulation package and two-word integer support routines are included in this library. SYSLIB subroutines enable you to write almost all application programs in FORTRAN without having to do any assembly language coding. For information about the system subroutine library, refer to *RT–11 System Subroutine Library Manual*.

If you are a C-language programmer, you can access RT–11 monitor services by using RTSYS.H in conjunction with SYSLIB.OBJ.

## 1.1 Programmed Requests

When you require a certain monitor service, you issue a programmed request in your source program. The programmed request in your source program expands into the appropriate machine language instructions during assembly time. When the program executes, these instructions request the resident monitor to supply the service represented by the programmed request.

Monitor services consist of the following processes:

- Initialization and control of operating system characteristics
- Allocation of system resources and reporting status
- Command interpretation
- File operations
- Input/output operations
- Interjob communications
- Timer support
- Program termination or suspension
- Extended memory functions

The system macro library (SYSMAC.SML) also contains several macros which are not programmed requests; they are described in Chapter 2 along with the programmed requests. These macros are provided to aid you in writing:

- Interrupt service routines
- Device handlers
- Memory management control blocks
- Consistency checking routines

### 1.1.1 Operating System Features

The RT–11 operating system features enhanced monitors, system job support and multiterminal operation support.

#### 1.1.1.1 RT–11 Monitors

The RT–11 monitors, built from one set of common sources, offer the following variety of operational configurations. See Figure 1–1.

- Single-job and multijob unmapped monitors
- Single-job and multijob single-mapped monitors
- Single-job and multijob fully mapped monitors

**Single-Job Unmapped Monitors**

There are three single-job unmapped RT–11 monitors:

- SB replaces the single job (SJ) monitor, and supports most programmed requests. SB supports program requests that manipulate files, perform input and output, set timer routines, check system resources and status, and terminate program operations.

**Figure 1–1: RT–11 Monitors**

| Monitors/Modes Supported | Unmapped Monitors K, I | Single Mapped U–K, I | Fully Mapped U–S–K, I–D |
|---|---|---|---|
| Single Job | SB | XB | ZB |
| Multijob (2) | FB | –– | –– |
| System Job (8) | –– | XM | ZM |
| Other Single–Job Monitors | MT (used with MDUPs only)<br>AI (used for installations) | | |

- MT is used only with MDUPs.
- AI is used only with automatic installation.

### Single-job Mapped Monitors

Two single-job mapped RT–11 monitors, XB and ZB, provide programmed requests and features in addition to those provided by the FB monitor:

- XB is a single-mapped monitor that supports User and Kernel modes.
- ZB is a fully-mapped monitor that supports I and D space for User, Supervisor, and Kernel modes.

### Multijob Unmapped Monitors

FB is the unmapped multijob monitor. Multijob monitors support program requests in addition to those supported for the single-job monitor. Some programmed requests are provided for the multijob monitor only. Multijob monitors enable a program to set timer routines, suspend and resume jobs, and send messages and data between foreground and background jobs.

### Multijob Mapped Monitors

Mapped monitors extend RT–11's memory support capability beyond the 28K-word (plus I/O page) restriction imposed by the 16-bit address size. Mapped monitors program requests extend a program's effective logical addressing space (See Table 1–5).

There are two multijob mapped monitors:

- XM is a single-mapped monitor that supports User and Kernel modes.
- ZM is a fully-mapped monitor that supports I and D space for User, Supervisor, and Kernel modes.

#### 1.1.1.2 System Job Environment

Programmed requests in the system job environment enable programs to:

- Copy channels from other jobs
- Obtain job status information about jobs
- Send messages and data between jobs

Programmed requests perform most system resource control and interrogation functions; however, some communication is accomplished by directly accessing two memory areas:

- System communication area
- Monitor fixed-offset area

Of all the distributed RT–11 monitors, only XM and ZM let you run programs in the system job environment. This system job support enables you to run up to eight user programs in single- or fully-mapped memory environment. RT–11 is distributed with the following programs that can be run as system jobs:

- Error logger (ERRLOG)
- Device queue program (QUEUE)
- Transparent spooler package (SPOOL)
- Communication package (VTCOM)
- Keypad editor (KEX)
- Virtual index (INDEXX)
- Resident monitor (RTMON)

#### 1.1.1.3 Multiterminal Operation

The multiterminal feature of RT–11 enables your program to perform character input/output on up to 17 terminals. Programmed requests are available to perform input/output, attach and detach a terminal for your program, set terminal and line characteristics, and return system status information.

#### 1.1.1.4 System Communication Areas

**System Area**

The system communication area resides in locations 40 to $57_8$ and contains parameters that describe and control execution of the current job. This area holds information such as the Job Status Word, job starting address, User Service Routine (USR) swapping address, and the resident monitor's start address. Your program provides some of this information, but other data provided by the monitor may not be changed.

**Fixed Offset Area**

The second memory communication area, the fixed-offset area, is accessed by a fixed-address offset from the start of the resident monitor. This area contains system values that control monitor operation. Your program can examine or modify these values to determine the condition of the operating environment while a job is running. Digital recommends this area be accessed, using only .GVAL, .PVAL, .PEEK, .POKE. The *RT–11 System Internals Manual* contains details about the system communication area and the fixed-offset area.

This manual describes programmed requests specifically for RT–11 Version 5. For information about programmed requests for earlier versions of RT–11 and guidelines for their conversion, refer to Appendix A.

## 1.2 Programmed Request Implementation

### 1.2.1 EMT Instructions

A programmed request is a macro call followed by the necessary number of arguments. The macro definition corresponding to the macro call of a programmed request is *expanded* by the MACRO assembler whenever the programmed request appears in your program. The expansion arranges the arguments of the programmed request for the monitor and generates the hardware *emulator trap* instruction. However, some macros like .DRxxxx do not generate EMTs. EMT instructions should never appear in your programs, except through programmed requests.

When an EMT instruction is executed, control passes to the monitor. The low-order byte of the EMT code provides the monitor with the information that tells it what monitor service is being requested. The execution of the EMT generates a trap through vector location 30, which is loaded at boot time with the address of the EMT processor in the monitor.

Table 1–1 lists codes that may appear in the low-order byte of an EMT instruction and gives the monitor's interpretation of these codes.

**Table 1–1: EMT Codes**

| Low-Order Byte | Interpretation |
|---|---|
| 377 | Reserved; RT–11 ignores this EMT by returning control to the user program immediately. |
| 376 | Reserved; used internally by the RT–11 monitor. Your programs should not use this EMT since its use would lead to unpredictable results. |
| 375 | Programmed request with several arguments; R0 points to a block of arguments that supports the user request. |
| 374 | Programmed request with one argument; R0 contains a function code in the high-order byte and a channel code in the low-order byte. |
| 373 | Program request to call Kernel routines. |
| 360-372 | Reserved; used internally by the RT–11 monitor; your programs should never use these EMT codes since their use would lead to unpredictable results. |
| 340-357 | Programmed requests with the arguments on the stack and/or in R0. |
| 0-337 | RT–11 version 1 programmed requests with arguments both on the stack and in R0. They are supported only for binary compatibility with Version 1 programs. |

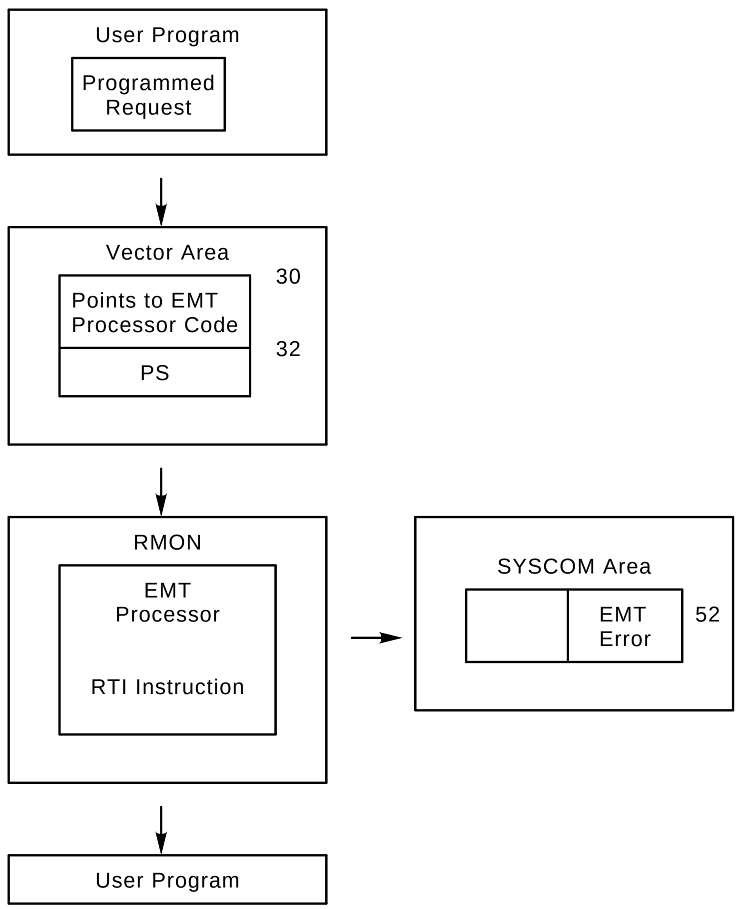
### 1.2.2 System Control Path Flow

Figure 1–2 shows system flow when a programmed request in an application (or system utility) program is implemented with an EMT instruction. When your program is executed, the following occurs:

1. The EMT instruction transfers control to the EMT processor code in the monitor.
2. The user program counter (PC) and processor status word (PS) are pushed onto the stack, and the contents of location 30 are placed in the program counter.
3. Location 30 points to the EMT processor code in the monitor. Location 32 contains the PSW for the EMT trap.
4. The monitor loads byte 52 of the system communication area with an error code if the monitor detects any errors during EMT processing.
5. The EMT processor uses R0 to pass back information to the program. All other registers are preserved. Unlike other EMTs, .CSIGEN and .CSISPC return information on the stack, thereby modifying the stack pointer.
6. Request blocks passed to EMTs are accessed, but not modified by the monitors. Parameters pushed onto the stack by standard macro definitions are popped from the stack by the monitor through standard EMT processing.

The monitor either processes a programmed request entirely when it is issued or performs partial processing and queues the request for further processing. For information about requests that perform I/O operations, see Section 1.4.5. When a request results in an error prior to its being queued, the completion routine is not entered, and the monitor returns to the user program with the carry bit set. If the request is queued, the completion routine is entered upon completion of further processing, regardless of the outcome.

**Figure 1–2: System Flow During Programmed Request Execution**




## 1.3 System Conventions

This section describes system conventions that must be followed to ensure correct operation of programmed requests.

### 1.3.1 Program Request Format

To issue programmed requests from assembly language programs, you must set up the arguments in correct order and issue the appropriate EMT instruction. Macros have been created to help you do this. They are contained in the system macro library named SYSMAC.SML. Their use is recommended for maintaining program compatibility with future releases of RT–11 and for program readability. Most names for definitions in SYSMAC.SML, except SOB, start with a period (.) to distinguish them from symbols and macros you define.

Most arguments provided to a programmed request must be valid assembler expressions because the arguments are used as source fields in the instructions (such as a MOV instruction) when macros are expanded at assembly time. Each programmed request in your program must appear in a .MCALL directive to make the macro definition available from system macro library, SYSMAC.SML. Alternatively, you can enable the automatic .MCALL feature of MACRO by using the .ENABL MCL directive. (However, you cannot use .ENABL MCL to automatically .MCALL .PRINT.)

Because there are various ways to set up the argument block and specify arguments to a programmed request, you should read the sections on programmed request format and on blank arguments to be sure you understand programmed request operation. Program requests have two acceptable formats:

**FORMAT 1**

The first format specifies the programmed request, followed by the arguments required by the request.

Form:

**.PRGREQ arg1,arg2,...,argn**

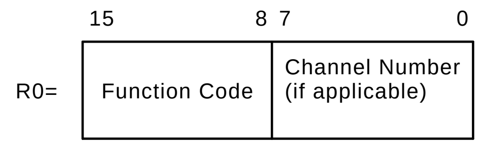
where:

| | |
|---|---|
| **.PRGREQ** | is the name of the programmed request |
| **arg1,arg2...,argn** | are arguments used with the request. |

Programmed requests using this format generate either an EMT 374 instruction or EMT 340 through 357 instructions.

Programmed requests that use an EMT 374 instruction set up R0 with the channel number in the even (low-order) byte and the function code in the odd (high-order) byte, as shown in Figure 1–3.

**Figure 1–3: EMT 374 Argument**




For example, the programmed request .DATE generates an EMT 374. The macro for this programmed request appears in the system macro library as:

```
        .TITLE  EXDATE.MAC

.MACRO  .DATE
        MOV     #10.*^o400,R0
        EMT     ^o374
.ENDM
```

The *function code*, in this case $10_{10}$, is placed in the high-order byte of R0. A 0 is placed in the low-order byte since .DATE does not reference a channel.

Any arguments for EMT 340 through 357 would be placed either on the stack, in R0, or in R0 and on the stack.

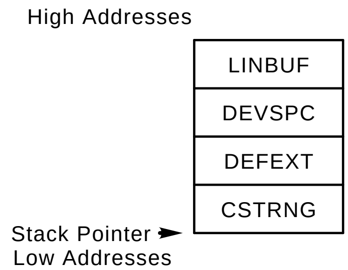
.CSIGEN is an example of a programmed request that generates an EMT 344. A simplified macro expansion of this programmed request is:

```
        .TITLE  EXCSIG.MAC

.MACRO  .CSIGEN   DEVSPC,DEFEXT,CSTRNG,LINBUF
.IF NB  LINBUF
        MOV     LINBUF,-(SP)
.ENDC
        MOV     DEVSPC,-(SP)
.IF NB  LINBUF
        INC     @SP
.ENDC
        MOV     DEFEXT,-(SP)
.IF B   CSTRNG
        CLR     -(SP)
.IFF
.IF IDN CSTRNG,#0
        CLR     -(SP)
.IFF
        MOV     CSTRNG,-(SP)
.ENDC
.ENDC
        EMT     ^o344
.ENDM
```

When this programmed request is executed, all the specified arguments are placed on the User stack. The EMT processor then uses these arguments in performing the function of the programmed request .CSIGEN. See Figure 1–4.

**Figure 1–4: Stack Set by .CSIGEN Programmed Request**




**FORMAT 2**

The second format specifies the programmed request, the address of the argument block, and the arguments that will be contained in the argument block.

Form:

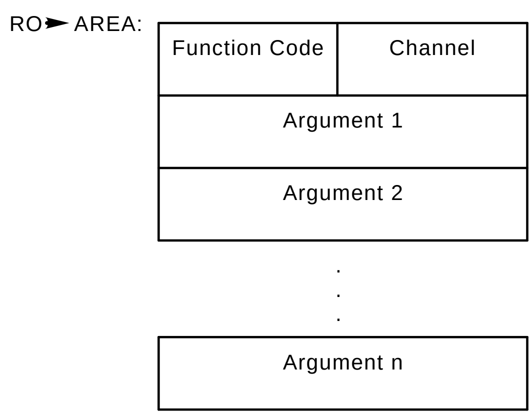
**.PRGREQ area,arg1,arg2,...,argn**

where:

| | |
|---|---|
| **.PRGREQ** | is the name of the programmed request |
| **area** | is the address of an argument block |
| **arg1,arg2,...,argn** | are arguments that will be contained in the argument block. |

This format generates an EMT 375 instruction. Programmed requests that call the monitor, via an EMT 375, use R0 as a pointer to an argument block. See Figure 1–5.

**Figure 1–5: EMT 375 Argument Block**




The programmed request format uses *area* as a pointer to the argument block containing the arguments *arg1* through *argn*.

Form:

**.PRGREQ area,arg1,...,argn**

Blank fields are permitted; however, if the *area* argument is empty, the macro assumes that R0 points to a valid argument block. If any of the fields *arg1* to *argn* are empty, the corresponding entries in the argument list are left untouched. For example,

**.PRGREQ area,arg1,arg2**

points R0 to the argument block at *area*, fills in the first word (function code and channel number) and fills in the first and second arguments, while

**.PRGREQ area**

points R0 to the block and fills in the first word (function code and channel number) without filling in any other arguments. Arguments left blank are discussed in the next section.

### 1.3.2 Blank Arguments

Any programmed request that uses an argument block assumes that any argument left blank has been previously loaded by your program into the appropriate memory location (exceptions to this are the .CHCOPY and .GTJB requests). For example, when the programmed request

**.PRGREQ area, arg1, arg2**

is assembled, R0 will point to the first word of the argument block. The first word has the function code in the high-order byte and the channel number in the low-order byte. *arg1* is in the second word of the argument block (that is, pointed to by the contents of address R0 plus 2), while *arg2* is in R0 plus 4.

There are two ways to account for arguments:

- Let the MACRO assembler generate the instructions needed to fill up the argument block at run time.
- Write these instructions in your program, leaving the arguments in the programmed request blank for those that you have written in.

Digital recommends that you let SYSMAC.SML macro definitions generate the instructions, both for program clarity and for reduced chance of programming error.

The next three examples are all equivalent because the arguments have been accounted for either in the program instructions or in the programmed request. The second example sets up all the arguments for the programmed request, prior to executing the programmed request:

```
.TITLE  EXPRGA.MAC

MOV     #ARG1,AREA+2
MOV     #ARG2,AREA+4
.PRGREQ #AREA

.TITLE  EXPRGB.MAC
```

```
        MOV     #AREA,R0
        .PRGREQ ,,#ARG1,#ARG2

        .TITLE  EXPRGC.MAC

        MOV     #AREA,R0
        MOV     #CODE*400!CHANNEL,@R0
        MOV     #ARG1,2(R0)
        MOV     #ARG2,4(R0)
        .PRGREQ
```

The next example demonstrates how arguments are specified to the .TWAIT programmed request:

```
        .TITLE  EXWAIT.MAC

        .MCALL  .PRINT,.TWAIT
START:
WAIT:   .TWAIT  #AREA,#TIME
        .PRINT  #MSG
        BR      WAIT
AREA:   .BLKW   2
TIME:   .WORD   0,10.*60.
MSG:    .ASCIZ  /Print this every 10 seconds/
.END    START
```

The .TWAIT programmed request suspends a program and requires two arguments:

- The first argument, *area*, is replaced by the address of a two-word EMT argument block.
- The second argument, *time*, is replaced by two words of time—high-order first, low-order second, expressed in ticks.

In the example, AREA is specified as an argument with the programmed request that points to the address of the EMT argument block. The first word of the argument block has a zero stored in the low-order byte representing the channel number and a function code of 24 stored in the high-order byte. The second word contains a pointer to the location (the second argument), which specifies the amount of time that the program will be suspended. It is defined as two words and, in this example, represents a 10-second interval. When run, the example program prints its message every ten seconds.

### 1.3.3 Addressing Modes

You must make certain that the arguments specified are valid source fields and that the address accurately represents the value desired. If the value is a constant or symbolic constant, use the immediate addressing mode (#), but if the value is in a register, use the register symbol (Rn). If the value is in memory, use the label of the location whose value is the argument.

A common error is to use *n* rather than #*n* for numeric arguments. When a direct numerical argument is required, the immediate mode causes the correct value to be put in the argument block; for example,

```
        .TITLE  EXPRGD.MAC
        .PRGREQ #AREA,,#4
```

is correct, while:

```
        .TITLE  EXPRGE.MAC
        .PRGREQ #AREA,,4
```

is not correct, because the contents of location 4, instead of the desired value 4, are placed into the argument block. However, the form in the next example is correct, because the contents of *list* is the argument block pointer and the contents of *number* is the data value:

```
        .TITLE  EXPRGF.MAC
        .PRGREQ LIST,,NUMBER
        ...
        .PSECT  DATA
LIST:   .WORD   AREA
NUMBER: .WORD   4
```

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. Also, with the exception of calls to the Command String Interpreter (.CSIGEN/.CSISPC), the position of the stack pointer is preserved across a programmed request.

Be sure that addressing mode provided to the macro generates the correct value as a *source operand* in a MOV instruction. Check the programmed request macro in the Macro Library (SYSMAC.SML) and manually expand the programmed request or use the macro assembler (by using the .LIST MEB directive) to be sure of correct results.

### 1.3.4 Keyword Macro Arguments

The RT–11 MACRO assembler supports keyword macro arguments. All the arguments used in programmed request calls can be encoded in their keyword form. See the *PDP–11 MACRO–11 Language Reference Manual* for details.

In EMT 375 programmed requests, the high byte of the first word 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 it 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 set to the appropriate code, 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. This enables you 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.

### 1.3.5 Channels and Channel Numbers

A channel is a data structure that is a logical connection between your program and a file on a mass storage device or on a serial device such as the line printer or terminal. The system provides $16_{10}$ channels by default. When a file is opened on a particular device, a channel number is assigned to that file. The channel number can have an octal value from 0 to 376 (0 to 254 decimal). Your program first opens a channel through a programmed request by specifying the device and/or file name, file type, and a channel number to the monitor, then refers to that file or device in all subsequent I/O operations by the assigned channel number. You can specify a device (non-file-structured) or a device and file name (file-structured).

Channel $255_{10}$ is reserved for system use. Channel $15_{10}$ is used by the system's overlay handler (if the program is overlaid).

### 1.3.6 Device Blocks

A device block is a four-word block of Radix–50 information that you set up to specify a physical or logical device name, file name, and file type for use with a programmed request. When your program opens a file, this information is passed to the monitor which uses the information to locate the referenced device and the file name in the corresponding directory. For example, a device block representing the file FILE.TYP on device DK might be written as:

```
        .TITLE  EDBLK1.MAC

;        123456
.RAD50  /DK /          ;device
.RAD50  /FILE  /       ;name
.RAD50  /TYP/          ;type
```

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

```
        .TITLE  EDBLK2.MAC

;        123456789ABC
.RAD50  /DK FILE  TYP/  ;complete DBLK
```

Spaces must fill out each field. Also, the colon and period separators must not appear in the string since they are only used by the Command String Interpreter to delimit the various fields.

If the first word of a device block is the name of a mass-storage device such as a disk and the second word of the block is 0, the device block refers to an entire volume of the mass storage device in a non-file-structured manner.

### 1.3.7 Programmed Request Errors

Programmed requests use three methods of reporting errors detected by the monitor:

- Setting the carry bit of the processor status word (PSW)
- Reporting the error code in byte 52 of the system communications area
- Generating a monitor error message

If a programmed request has been executed unsuccessfully, the monitor returns to your program with the carry bit set. The carry bit is returned clear after the normal termination of a programmed request. Almost all requests should be followed by a Branch Carry Set (BCS) or Branch Carry Clear (BCC) instruction to detect a possible error.

Because some programmed requests have several error codes, byte 52 in the system communications area is used to receive the error code. Therefore, when the carry bit is set, check byte 52 to identify the kind of error that occurred in the programmed request. The meanings of values in the error byte are described individually for each request. The error byte is always zero when the carry bit is clear. Your program should reference byte 52 with absolute addressing. Always address location 52 as a byte, *never as a word*, because byte 53 has a different use. The following example shows how byte 52 can be tested for the error code:

```
       .TITLE  ERRBYT.MAC

$ERRBY  =:      52              ;(.SYCDF) error byte
        .PRGREQ #AREA,ARG1,...,ARGN
        BCS     ERROR
        ...
ERROR:  TSTB    @#$ERRBY
        ...
.END
```

Error messages generated by the monitor are caused by fatal errors, which cause your program to terminate immediately. Some fatal errors can be intercepted and have their values returned in byte 52 (See the .HERR/.SERR programmed requests).

### 1.3.8 User Service Routine (USR) Requirement

The USR is always resident in mapped monitors; therefore, the following discussion of programmed request requirements for the USR is applicable only to unmapped monitors. Programmed requests that require USR to be in memory require a fresh copy of the USR to be read in because the code to execute them resides in the USR buffer area. Since the buffer area gets overlaid by data used to perform other system functions, the USR must be read in from the system device even if there is a copy of the USR presently in memory. In mapped monitors, USR is always in memory. The SB monitor will verify the swap address is even and within the job. Table 1–2 lists programmed requests that require the USR and notes any exceptions to the requirement.

**Table 1–2: Programmed Requests Requiring the USR**

| | | |
|---|---|---|
| .CHAIN[4] | .FPROT | .RELEASE |
| .CLOSE[1] | .GFDATE[1] | .RENAME[1] |
| .CLOSZ[1] | .GFINFO[1] | .SFDATE[1] |
| .CSIGEN | .GFSTAT[1] | .SFINFO[1] |
| .CSISPC | .GTLIN | .SFSTAT[1] |
| .DELETE[1] | .HRESET[4] | .SRESET[4] |
| .DSTATUS | .LOCK[5] | .TLOCK[5] |
| .ENTER[1] | .LOOKUP[1] | |
| .EXIT[4] | .PURGE[1,2] | |
| .FETCH | .QSET[3] | |

[1]Special directory operations always require the USR. Or, if the channel has been opened with a non-file-structured open, the USR is not required.

[2]If the channel has not been opened on a special directory device, the USR is not required.

[3]Requires a fresh copy of the USR to be read into memory.

[4]If the job FETCHed a handler which specified a *RELEASE=routine* in .DRPTR or any handler currently in memory requested notification of job exit.

[5]Ensures the USR is in memory.

USR requirements for programmed requests differ among the monitors as shown in the table. The .CLOSE programmed request on non-file-structured devices, such as a line printer or terminal, does not require the USR under any monitor.

Because USR is not reentrant, only one job at a time can use the USR. This is particularly important for concurrent jobs when a magnetic tape device is active. For example, file operations on tape devices require a sequential search of the tape. When you issue a USR file operation from a background program to a magtape, USR locks out the foreground job until the background job is complete. Special function request SF.USR provided in file structured magtape handlers can be used to perform asynchronous directory operations on tape.

In multijob environments, jobs can use the .TLOCK request to check USR availability. If the USR is not available, control returns immediately with the C bit set, indicating that the .LOCK request (that attempts to gain ownership of USR) has failed. In this way a job can perform critical tasks before losing control by being queued up for USR availability.

Any request that requires the USR to be in memory can also require that a portion of your program be saved temporarily in the system device swap file; that is, a portion of your job can be "swapped out" and stored in file SWAP.SYS to provide room for the USR. Then the USR is then read into memory. Although swapping is invisible in normal operation, you must consider it in your programming. For example, the argument block being passed to the USR must not be in the area that is swapped over. You can save time by optimizing your programs so that they require little or no swapping.

Consider the following items if a swap operation is necessary:

- **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 (not encountered in mapped monitors because the USR is always resident).

- **Value of Location 46**

  If you assemble an address into word 46 or move 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 zero, the USR will be at its normal location in high memory. If the USR does not require swapping, this value is ignored.

  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 will occur. The SET USR NOSWAP command makes the USR permanently resident.

  If you specify an alternate address in location 46, the SB monitor will verify the validity of the USR swap address. In previous versions of RT–11 the SJ monitor did not validate the address.

- **Monitor Offset 374**

  The contents of monitor offset 374 indicates the size of the USR in bytes.

  Programs should use this information to dynamically determine the size of the region needed to swap the USR.

- **Protecting Program Areas**

  Make sure the following areas of your program do not get overlaid when you swap in the USR:

  - Program stack
  - Any parameter block for calls to the USR
  - EMT instruction that invoked the USR
  - I/O buffers
  - Device handlers
  - Interrupt service routines
  - Queue elements
  - Defined channels
  - Completion routines in use when USR is called

The *RT–11 System Internals Manual* provides additional information on the USR.

## 1.4 Using Programmed Requests

This section describes how to implement programmed requests to access the various monitor services.

### 1.4.1 Initialization and Control

Typically, you use several programmed requests to control the operating environment in which your program is running. These requests can include control of:

- Memory allocation
- I/O access
- Devices
- Error processing

#### Memory Allocation

When loaded, a program occupies the memory specified by its image created at link time. A program's memory requirements are specified to the monitor by the .SETTOP request. To obtain more memory, execute a .SETTOP request with R0 containing the highest address desired. The monitor:

- Determines whether the address is valid.
- Returns the highest address available.
- Determines whether it is necessary to swap the USR.

Resident handlers or foreground jobs can restrict the amount of memory available to meet the amount requested for the program. The monitor retains the USR in memory, if possible, thereby reducing the amount of swapping. When this is not possible, the monitor will automatically remove the USR from memory and swap part of the user program to swap file (SWAP.SYS) on the system device whenever the USR must be reloaded to process a request. The .SETTOP request determines how much memory is available and controls monitor swapping characteristics. (See the .SETTOP programmed request in Chapter 2 for special optional features provided in an extended memory environment. Additional information on the .SETTOP request is also given in the *RT–11 System Internals Manual*.)

If a program needs so much memory that the USR must swap, swapping will automatically occur whenever a USR call is made, but when a program knows file operations are necessary so it can consolidate and perform these operations individually, system efficiency can be enhanced as follows:

- Request the USR to be swapped in.
- Have it remain resident while a series of consecutive USR operations is performed.
- Swap the USR out when the sequence of operations is completed.

Three programmed requests control USR swapping. The request .LOCK causes the USR to be made resident for a series of file operations:

- Requiring a portion of your program to be written to the swap blocks prior to reading in the USR.
- Requiring the USR to be read in, if it finds the USR is overwritten.

The request .UNLOCK swaps your program back in if it was swapped out, and the USR is overwritten; otherwise, no swapping occurs. The request .TLOCK makes the USR resident in multijob monitors only if the USR is not currently servicing another job's file requests at the time the .TLOCK request is issued. This check prevents a job from becoming blocked while the USR is processing another job's request. When a .TLOCK succeeds, the USR is ready to perform an operation immediately. In a single-job environment, the .TLOCK request performs exactly like the .LOCK request.

RT–11 provides $16_{10}$ channels as part of the job's impure area; that is, 16 files can be allocated at one time. Up to $255_{10}$ channels can be allocated with the .CDFN request. This request sets aside memory inside the job area to provide the storage required for the status information on the additional channels. Once the .CDFN request has been executed, as many channels as specified can be active simultaneously. Use the .CDFN request during the initialization phase of your program. The keyboard monitor command CLOSE does not work if you define new channels with the .CDFN programmed request.

The .CNTXSW request lets the job add memory locations to the list of items to be context-switched. The request itself does not cause a context switch to occur.

#### Input/Output Access

Each pending I/O, message, or timer request must be placed into one of the monitor queues. These are then processed by the monitor on a first-in first-out basis, by job priority, or by time of expiration. In RT–11, all I/O transfers are queued to allow asynchronous processing of the request. A queue is a list of elements, each element being seven words long in unmapped monitors, and ten decimal words long when using mapped monitors. When your program issues a data transfer programmed request, the information specifying the transfer is stored by the monitor in a queue element. This information is passed to the device handler, which then processes the I/O transfer.

Each job, whether background or foreground, initially has only a single queue element available. Additional queue elements may be set aside with a .QSET request. The .QSET request declares where in memory the additional queue elements will go and how many elements there will be. If you do not include a .QSET request in your program, the monitor uses the queue element set aside in the job's impure area. In this case, since only one element is available for each job, all operations would be synchronous. That is, any request issued when the available queue element list is empty has to wait for that element to become free. The number of queue elements necessary equals the number of asynchronous operations pending at any time.

### Devices

The .DEVICE request turns off any special devices that are being used by the running program upon program termination. This request lets you specify a set of device control register addresses and a value to be set in each register on job exit. When a job is terminated—either normally, by an error condition, or by a CTRL/C—the specified values are set in the specified locations.

Loading a background job with a GET, R, or RUN command, or loading a foreground or system job with a FRUN and SRUN command, respectively, alters most locations in the vector area 0 to 476. Virtual jobs do not load over the vector area. RT–11 automatically prevents alteration of locations used by the system, such as the clock, the console terminal, and all vectors used by handlers that are loaded. If a foreground job in a foreground/background environment accesses a device directly through an in-line interrupt service routine, the foreground job must notify the monitor that it must have exclusive use of the vectors. Using the .PROTECT programmed request lets the foreground job gain exclusive use of a vector. The .PROTECT request can also be used by either the foreground or background job, prior to setting the contents of a vector, to test whether the vectors are already controlled by a job. This serves as further protection against jobs interfering with each other. An .UNPROTECT programmed request relinquishes control of a vector, making the vector available to both the background and foreground jobs.

Special function requests (See listing in *RT–11 Device Handlers Manual*) are used for performing special functions on devices such as magnetic tape. .SPFUN requests are used for such functions as rewind or space-forward operations.

### Error Processing

During the course of program execution, errors can occur that cause the monitor to stop the program and print a *MON-F* error message, such as directory I/O errors, monitor I/O errors on the system device, or I/O requests to nonexistent devices. Some programs cannot let the monitor abort the job because of these errors. For example, in the case of RT–11 multi-user DIBOL, a directory I/O error affecting only one of the users should not cause the whole program to abort. For such applications, a pair of requests is provided—.HERR and .SERR:

- .HERR request (normal default) indicates that the monitor will handle severe errors and stop the job.
- .SERR request causes the monitor to return most errors to your program for appropriate action, setting an error code in byte 52.

In addition to processing I/O errors through .HERR and .SERR requests, you can also use the .TRPSET or .SFPA requests to handle certain fatal errors. Use these requests to prevent your program from aborting due to a trap to location 4 or $10_8$ or to the exception traps of the Floating Point Processor (FPP) or Floating Point Instruction Set (FIS). A .TRPSET request specifies the address of a routine that the monitor enters when a trap to location 4 or 10 occurs. A .SFPA request specifies the address of a floating-point exception routine called whenever an exception trap occurs.

### 1.4.2 Examining System Information and Reporting Status

Several programmed requests interrogate the system for specific details about a device or file that your program may be using:

| Request | Description |
|---|---|
| .CSTAT | Status information on a file: starting block, length, device location |
| .DATE | Obtains the system date, which then can be printed on a report or entered as a data record in a file. |
| .DSTAT | Status information on a device: block length, controller-assignment number |
| .GTIM | Obtains the time-of-day and is used in the same way as .DATE. |
| .GTJB | Obtains job information:<br>• Foreground or background information<br>• Memory limits<br>• Virtual high limit for a job created with the linker /V option (mapped monitors only)<br>• Unit number of the job's console terminal (if you are using the multiterminal feature)<br>• Address of the job's channel area<br>• Address of the job's impure area<br>• Logical job name (if you are using a monitor with the system job feature) |
| { .MTGET<br>.MTSTAT } | Multiterminal status information when using multiterminal feature |
| { .MFPS<br>.MTPS } | Read the priority bits and set the priority and T-bits in the processor status word (PS); let a program run without change on all processors, including those that support 177776 as PS. |
| .SDTTM | Sets the system date and/or time. Changing the date or time has no effect on any outstanding mark time or timed wait requests. |

### 1.4.3 Command Interpretation

Two of the most useful programmed requests are .CSIGEN and .CSISPC. These requests call the Command String Interpreter (CSI), which is part of the USR. They are used to process standard RT–11 command strings:

Form:

***Dev:Output[,size]/Option=Dev:Input/Option**

The monitor prints the asterisk on the terminal. The RT–11 system programs use the same command string. See the *RT–11 System Utilities Manual* for more detailed information.

The .CSIGEN request analyzes the string for correct syntax, automatically loads the required device handlers into memory, opens the files specified in the command, and returns to your program with option information. So, with one request, a language processor such as the FORTRAN compiler is ready to input from the source files and output the listing and binary files. You can specify options in the command string to control the operation of the language processor. The .CSIGEN request uses channels 0 through 2 to accommodate three output file specifications and channels 3 through $10_8$ to accommodate six input file specifications.

The .CSISPC request gets you the services of the command processor, but lets you do your own device file manipulation. When you use .CSISPC, the CSI:

1. Obtains a command string
2. Analyzes it syntactically
3. Places the results in a table
4. Passes the table to your program for appropriate action

The .GTLIN request obtains one line of input at a time instead of one character at a time. These three requests support the indirect file function and let your program obtain one line at a time from an indirect file. Therefore, if your program were started through an indirect file, the line would be taken from the indirect file and not from the terminal. The .GTLIN request has an optional argument which forces input to come from the terminal, a useful feature if your program requires information only available from the terminal.

### 1.4.4 File Operations

A device handler is the normal RT–11 interface between the monitor and the peripheral device on which file operations are performed. The console terminal handlers and the interjob message handlers are part of the resident monitor and require no attention on your part. All other device handlers are loaded into memory with either a .FETCH request from the program or a LOAD command from the keyboard, before any other request can access that device. (See Input/Output Operations section that describes the use of programmed requests for performing I/O operations. The *RT–11 System Internals Manual* describes how to write device handlers for RT–11.

Once the handler is in memory, a .LOOKUP request can locate existing files and open them for access. New files are created with an .ENTER request. Space for the file can be defined and allocated as:

- One-half the size of the largest unused space or all of the second largest space, whichever is larger (the default)
- A space of a specific size

- As much space as possible

The parameter you specify as the file size argument of the .ENTER request or as specified in a .CSIGEN command string affects the way the system allocates space.

When file operations are completed, issuing a .CLOSE request makes the new file permanent in the directory. Issuing a .CLOSZ request accomplishes the same thing, but it lets you specify the file length. A .PURGE request can free the channel without making the file permanent in the directory. Existing permanent files can be renamed with a .RENAME request or deleted with a .DELETE request.

Two other requests, .SAVESTATUS and .REOPEN, add to the flexibility of file operations:

- .SAVESTATUS stores the current status of a file that has been opened with a .LOOKUP request and makes the file temporarily inactive, thus freeing the channel for use by another file.
- .REOPEN causes the inactive file to be reactivated on the specified channel, and I/O continues on that channel. (You can open more files than there are channels.)

If you also lock the USR in memory, you can open all the files your program needs, while maintaining system swapping efficiency, by:

- Locking the USR in memory, and opening the files that are needed.
- Issuing the .SAVESTATUS request.
- Releasing the USR.
- Issuing a .REOPEN request each time a file is needed.
- Locking USR, and using the .CLOSE request to make the files permanent.

Because a .REOPEN request does not require any I/O, all USR swapping and directory motion can be isolated in the initialization code for an application, thereby improving program efficiency.

The following requests are useful for obtaining or modifying file information:

- .GFDAT provides file's creation date for file's directory entry.
- .GFINF provides word content of directory offset specified from file's directory entry.
- .GFSTA provides file status information from the file's directory entry.
- .SFDAT lets you change the date that appears in a file's directory entry listing. You may want to do this for a file that you update in place, for example, or if the original creation date was in error.
- .SFINF lets you change the contents of the directory entry offset specified in file's directory entry.
- .SFSTA lets you change the status information in a file's directory entry.

- .FPROT protects a file against deletion or removes protection so that a file can be deleted by a .DELETE, .ENTER or .RENAME request. The contents of a protected file are not protected against modification.

### 1.4.5 Input/Output Operations

You can perform I/O in three different modes that optimize the overlap of CPU and I/O processing:

- Synchronous I/O
- Asynchronous I/O
- Event-driven I/O

#### Synchronous I/O

The programmed requests .READW and .WRITW perform *synchronous* I/O; that is, the instruction following the request is not executed until the I/O transfer is completely finished; in this way the program and the I/O process are synchronized.

#### Asynchronous I/O

The program requests .READ, .WRITE, and .WAIT perform *asynchronous* I/O; that is, the .READ or .WRITE request adds the transfer request to the queue for the device:

- If the device is inactive, the request is placed at the beginning of the queue; the transfer begins; control returns to the user program before the transfer is completed.
- If the device is active, the request is queued; control returns to user before transfer is complete.

The .WAIT programmed request, however, blocks the program until the transfer is completed, enabling the I/O operation to be completed before any further processing is done. Asynchronous I/O is most commonly used for double buffering.

#### Event-Driven I/O

Program requests, such as .READC and .WRITC, perform *event-driven* I/O; that is, they initiate a completion routine when the transfer is finished. Event-driven I/O is practical for conditions where system throughput is important, where jobs are divided into overlapping processes, or where processing timings are random. The last condition is the most attractive case for using event-driven I/O because processor timing may range up to infinity in a process that is never completed.

Because completion routines are essential to event-driven I/O, the next section presents general guidelines for writing completion routines.

#### 1.4.5.1 Completion Routines

Completion routines are part of your program that execute following the completion of some external operation, interrupting the normal program flow. On entry to an I/O completion routine, R0 contains the contents of the Channel Status Word and R1 contains the channel number for the operation. The carry bit is not significant.

Completion routines are serialized (within a job's context, not between jobs) and run at priority 0. Completion routines do not interrupt one another but are queued; the next completion routine is not entered until the first is completed.

If the foreground job is running and a foreground I/O transfer completes and wants a completion routine, that routine is entered immediately if the foreground job is not already executing a completion routine. If it is executing a completion routine, that routine continues to termination, at which point other completion routines are entered in a first-in first-out order. If the background job is running (even in a completion routine) and a foreground I/O transfer completes with a specified completion routine, execution of the background job is suspended and the foreground routine is entered immediately.

Also, it is possible to request a completion routine from an in-line interrupt service routine through a .SYNCH programmed request. This enables the interrupt service routine to issue other programmed requests to the monitor.

You must observe the following restrictions when writing completion routines:

- Completion routines should never reside in 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 being the likely result.
- Registers other than R0 and R1 must be saved upon entry to completion routines and restored upon exiting. Registers cannot transfer data between the mainline program and the completion routine.
- Under mapped monitors, completion routines must remain mapped while the request is active and the routine can be called.
- The completion routine must exit with an RETURN instruction because the routine was called from the monitor with a CALL ADDR, where ADDR is the user-supplied entry point address. If you exit completion routines with an .EXIT request, your job will abort.

  However, if you generate the special .SPCPS support, you can exit from a completion routine by using an .SPCPS request to change the main line PC so it points to .EXIT in the main program. When all completion routines are done, the .EXIT will be executed.

With the exception of the .SYNCH request, completion routines are normally run in User mapping in the mapped monitor context.

Frequently, a program's completion routine needs to change the flow of control of the mainline code. For example, you may wish to establish a schedule among the various tasks of an application program after a certain time has elapsed or after an I/O operation is complete. Such an application needs to redirect the mainline code to a scheduling subroutine when the application's timer or read/write completion routine runs. An .SPCPS programmed request saves the mainline code program counter and processor status word, and changes the mainline code program counter to a new value. If the mainline code is performing a monitor request, that request finishes before derailing can occur.

#### Terminal Input/Output

Several programmed requests are available to provide an I/O capability with the terminal:

- .TTYIN request obtains a character from the console
- .TTYOUT request prints a character at the terminal

Programs can issue .TTINR and .TTOUTR requests, which indicate that a character is not available or that the output buffer is full. The program can then resume operation and try again at a later time.

The .PRINT request prints multiple characters and can print multiple lines.

A .RCTRLO request forces the terminal output to be reactivated after a CTRL/O has been typed to suppress it, so that urgent messages will be printed.

### 1.4.5.2 Multiterminal Requests

The RT–11 multiterminal feature enables your program to perform input/output on up to 17 terminals. There are several programmed requests you can use to perform I/O on these terminals. Before issuing any of these programmed requests to a terminal, you must issue the .MTATCH request, which reserves the specified terminal for exclusive use by your program. The terminal cannot then be used by any other job until you issue the .MTDTCH request to detach the terminal.

Multiterminal requests .MTPRNT and .MTRCT0 have the same functionality as .PRINT and .RCTRLO, except that .MTPRNT specifies which terminal to print on. Unlike TTYIN and TTYOUT, the .MTIN request can transfer one or more characters to the program; .MTOUT can print one or more characters at the terminal.

By setting terminal and line characteristics with the .MTSET request, you provide a four-word status block that contains the terminal status word, the character of the terminal requiring fillers and the number of fillers required for this character, the width of the carriage (80 characters by default), and system terminal status. The status of a terminal can be obtained by issuing the .MTGET request.

The .MTSTAT program provides information about the entire multiterminal system, not about an individual terminal in the system.

## 1.4.6 Foreground/Background Communications

Communication between foreground and background jobs is accomplished through programmed requests .SDAT and .RCVD. These requests also have three modes (synchronous, asynchronous, and event-driven) that enable buffer transfer between the two jobs as if I/O were being done. The sending job treats a .SDAT request as a write, and the receiving job treats .RCVD as a read. In the case of .RCVD requests, the receiving buffer must be one word longer than the number of words expected. When the data transfer is completed, the first word of the buffer contains the number of words actually sent.

Jobs receiving messages can be activated when messages are sent through .RCVDC completion routines, while the sending jobs use .SDATC completion routines. The .MWAIT request is used for synchronizing message requests. It is similar to the .WAIT request that is used for normal I/O.

If you want one job to read or write data in a file opened by another active job, use the .CHCOPY request. For example, when the background job is processing data that is being collected by the foreground job, the .CHCOPY request enables you to copy channel information from the foreground job and to use the channel information to control a read or write request.

The multijob monitors always cause a context switch of critical items such as machine registers, the job status area, and floating-point processor registers (only swapped or context switched if the job is using .SFPA), when a different job is scheduled to run because it has a higher priority, or because the current job is blocked and a lower priority job is runnable. When the monitor saves a job's context, it saves the job-dependent information on the job's stack so that this information can be restored when the job is runnable again.

### 1.4.7 Timer Support

Monitor timer support is provided through the .MRKT request. The SB monitor, as distributed, does not have timer support, but can be selected during SYSGEN. Use the .MRKT request to specify the address of a routine that will be entered after a specified number of clock ticks. Like I/O completion routines, .MRKT routines are asynchronous and independent of the main program. After the specified time elapses, the main program is interrupted, the timer completion routine executes, and control returns to the interrupted program.

Pending .MRKT requests contained within the queue are identified by number. Pending timer requests can be canceled with a .CMKT request. .MRKT requests schedule by timer completion routines.

The programmed requests .MRKT/.CMKT and .TIMIO/.CTIMIO require request identification words as an argument. Certain ranges of values are reserved for different uses as shown in the following table.

**Table 1–3: Values Used with .MRKT/.CMKT, .TIMIO/.CTIMIO**

| Range | Use |
|---|---|
| 1-176777 | For user applications with a .MRKT/.CMKT. Values in this range are canceled if a .CMKT request is issued with a value of 0. |
| 177000-177377 | For use in device handler .TIMIO/.CTIMIO requests.[1] |
| 177400-177477 | Reserved for multiterminal support. |
| 177500-177677 | Reserved. |
| 177700 | Used by the .TWAIT request. |
| 177701-177766 | Reserved. |
| 177767-177777 | DECnet. |

[1]To ensure a unique value for each handler, DIGITAL suggests that the value be assigned as 177000+devcod, where *devcod* represents the device identifier value used in the .DRDEF macro at the beginning of the handler.

Values in range 177700 to 177777 are automatically canceled whenever a program terminates or aborts; however, values in the range 177000 to 177677 must be canceled individually by the routine that issued the .TIMIO request. This would occur, for example, in handler abort code.

Use the .TWAIT request to suspend a job for a specified time interval. For example, the .TWAIT request will let a compute-bound job relinquish CPU time to the rest of the system, so other jobs can be run.

### 1.4.8 Program Termination or Suspension

Many jobs come to an execution point when there is no further processing necessary until an external event occurs. In the multijob environment such a job can issue a .SPND request to *suspend* the execution of that job. While the foreground job is suspended, the background job runs. When the desired external event occurs, it is detected by a previously requested completion routine, which executes a .RSUM request to *resume* the job at the point it was suspended.

When a job is ready to terminate or reaches a serious error condition, it can reset the job environment with the .SRESET and .HRESET requests:

- .SRESET is a soft reset; that is, it reinitializes the monitor data base for the job, but allows queued I/O to run to completion.
- .HRESET is a hard reset; it stops all I/O for the job by calls to the handlers. .HRESET performs the same functions as .SRESET and resets queued I/O.

Using the programmed request .EXIT in a background job terminates the program and returns control to the keyboard monitor:

- If R0 contains a zero upon execution of this request, this causes hard reset that disables the commands REENTER, CLOSE, and START.

- If R0 contains a nonzero value upon exit from your program, this causes a soft reset, and commands REENTER, CLOSE, and START are not disabled.

In a foreground job, an .EXIT programmed request stops the job, and may return control to the keyboard monitor. You can remove the job from memory by issuing the UNLOAD command.

You may initiate the execution of another program with a .CHAIN request from a background job. Files remain open across a .CHAIN request and data is passed in memory to the chained job, so that it can continue processing. In FORTRAN, channel information is stored in the job's impure area, and this information is not preserved across a .CHAIN request. Therefore, close any channels in the first program, and reopen them in the program being chained to.

### 1.4.9 Job Communications

System job support enables communications between any two jobs in the system by using a special .LOOKUP, .READx, and .WRITx requests. The background job, can send and receive messages between each other by using the .RCVD, .MWAIT and .SDAT programmed requests.

### 1.4.10 Mapped and Unmapped Regions

In multijob environments, communication between jobs is accomplished by the Message Handler (MQ) which performs like an ordinary RT–11 device handler in accepting and dispatching I/O requests from the queued I/O system. .READ and .WRITE requests are able to send messages between any two jobs as if they were data transfers to files. Both the sending and receiving job must issue a .LOOKUP request on a channel and use 'MQ' as the device specification and the logical job name of the job with which they are communicating as the file specification. In the case of .READ requests, the receiving buffer must be one word longer than the number of words expected. When the data transfer is completed, the first word of the buffer contains the number of words actually sent (identical to the .RCVD requests). This does not apply to the .WRITE requests; the first word of the sending buffer is the first word of the message to be sent.

When assigning logical job names to system jobs, programmed requests such as .LOOKUP, .CHCOPY, and .GTJB must use the job's current logical job name (See the *RT–11 Commands Manual*).

### 1.4.11 Extended Memory Functions

The RT–11 mapped monitors enable MACRO programs to access extended memory by mapping their virtual addresses to physical locations in memory. This is done in conjunction with Memory Management Unit (MMU), a hardware option that converts a 16-bit virtual address to an 18- or 22-bit physical address.

Using the Virtual Run Utility (VBGEXE) enables your programs to run faster and with less low-memory space than your programs would otherwise require. These performance improvements result from running the programs as virtual jobs. If you are running under a mapped monitor, but there is not enough memory for your program to execute, try using VBGEXE. For more information, refer to the *RT–11*

*System Utilities Manual*, VBGEXE section. See V, VRUN, and SET RUN VBGEXE commands in the RT–11 command monitior.

Use programmed requests to access extended memory in a program. When accessing extended memory, first establish window and region definition blocks, then specify the amount of physical memory the program requires and describe the virtual addresses you plan to use. Do this by creating regions and windows, then associate virtual addresses with physical locations by mapping the windows to the regions. You can remap a window to another region or part of a region or you can eliminate a window or a region. Once the initial data structures are set up, manipulate the mapping of windows to regions that best meet your requirements.

There are five types of extended memory programmed requests:

- General mapping control
- Region requests
- Window requests
- Map requests
- Status requests

Window and region requests have their own data structures. RT–11 macro .WDBBK creates a window definition block and macro .RDBBK creates a region definition block. Both macros automatically define offsets and bit names. Two other macros, .WDBDF and .RDBDF, define only the offsets and bit names.

The programmed request .CRAW is used to create a window. To eliminate a window, use the .ELAW request. A region is created using the .CRRG request. You return a region to the free list of memory with the .ELRG request.

You map a window to a region with the .MAP request. If a window is already mapped to a region, this window is unmapped and the new one is mapped. Use the .UNMAP request to unmap a window. You obtain the mapping status of a window with the .GMCX request.

Mapping context information must be saved when virtual and privileged jobs are swapped out of memory. The .CMAP request defines and saves these mapping structures in the mapping context area (MCA) until the monitor restores them. The .GCMAP request obtains the current memory mapping context.

Several programmed requests are restricted when they are in a mapped monitors environment. These programmed requests and their restrictions are as follows:

**.CDFN** All channels must be in the lower 28K of memory (but not in the PAR1 region, 20000-37776 octal).

**.QSET** All queue elements must be $10_{10}$ words long and in the lower 28K of memory (but not in the PAR1 region, 20000-37776 octal).

**.SETTOP** Effective only in the virtual address space that is mapped at the time the request is issued, unless the job was linked with the /V option (See the *RT–11 System Utilities Manual*)

**.CNTXSW** Not usable in virtual jobs.

Detailed information on programmed requests in an extended memory environment is given in the *RT–11 System Internals Manual*.

The UNIBUS Mapping Register Handler (UB) supports UNIBUS mapping registers on UNIBUS processors. The UB handler provides direct memory access (DMA) support for 22-bit memory addressing during I/O operations. UMR support is appropriate for device handlers that perform I/O operations and are capable of DMA. UMR lets the handler access computer memory beyond the 18-bit 256K-byte boundary during I/O operations. For more information, refer to the *RT–11 Device Handlers Manual*.

### 1.4.12 Interrupt Service Routines

Some macros in system macro library (SYSMAC.SML) are not programmed requests, but are used like programmed requests in interrupt service communication to the monitor.

.INTEN, the first macro call in every interrupt routine, causes the system to use the system stack for interrupt service and enables the monitor scheduler to make note of the interrupt. If device service is all the routine does, .INTEN is the only call it has to make.

You must issue the .SYNCH call whenever you need to issue from the interrupt service routine, one or more programmed requests, such as .READ or .WRITE. The .INTEN call switches execution to the system state and, since programmed requests can only be made in the user state, the .SYNCH call handles the switch back to the user state. The code following the .SYNCH call executes as a completion routine. When .SYNCH is finished, the completion routine can execute programmed requests, initiate I/O, and resume the mainline code. The first word after the .SYNCH call is the return address on error, while the second word is the return on success. The *RT–11 System Internals Manual* contains a detailed description of interrupt service routines.

### 1.4.13 Device Handlers

The system macro library (SYSMAC.SML) contains several macros that simplify the writing of a device handler. Device handler macros are described in Chapter 2. The *RT–11 Device Handlers Manual* also explains the use of these macros in writing a device handler.

### 1.4.14 Logical Name Translation Bypass

Some programmed requests let you specify a "logical name translation bypass" (bypass), a modified form of device lookup. If you specify this form, only physical names of devices will be searched for. For example, if *bypass* is specified and a device DL0 is on the system with the logical name TMP assigned to it, you can find it by specifying DL0, but not TMP. This modified form is an unsupported interface and requires hand-coding of the request blocks, only use it under very limited circumstances.

You can use bypass for nearly all requests that specify a *dblock* or *dev.* The following requests specify a *dblock* support bypass:

| | | |
|---|---|---|
| .DELETE | .GFINFO | .SFINFO |
| .ENTER | .GFSTAT | .SFSTAT |
| .GFDATE | .LOOKUP | .RENAME |

Although .FPROT and .SFDAT do not support bypass, you can simulate their functionality by issuing .SFSTAT and .SFINFO. The following code fragment illustrates the coding required for bypass:

```
;               xxxxxx is the request (e.g., LOOKUP)
        .LIBRARY "SRC:SYSTEM"
        .MCALL  ..xxxxxx        ;Request name with 2 dots (..LOOKUP)
DOC$UN=1                        ;Gen names of undocumented parts
        ..xxxxxx

        ...
        .xxxxxx args            ;Request name with 1 dot (.LOOKUP)
        .=.-2                   ;Crush EMT instruction for now
        BIS     #..PHYS,A.DBLK(R0) ;Make dblock addr odd
        MOV     #RETODD,A.URTN(R0) ;Supply bypass flag
        EMT     ...xxxxxx       ;Issue EMT (Name with 3 dots (...LOOKUP))

        ...
RETODD  =:      .+1             ;(Addr of RETURN)+1
        RETURN

AREA:   .BLKW   5.              ;Expanded request block
```

The following requests that specify a device support bypass are coded in the following manner:

```
..PHYS  =: 1            ;Bypass flag value

        .DSTATUS        #Reply!..PHYS,#dev
        .FETCH          #Addr!..PHYS,#dev
        .RELEASE        See below

        .FETCH          #..PHYS,#dev    ;use for .RELEASE
```

Note that .RELEASE is coded as a special form of .FETCH.

### 1.4.15 Consistency Checking

The .ASSUME macro tests the validity of the condition you specify. If the test result is false, MACRO generates an assembly error and displays an appropriate error message.

The .BR macro notifies you when program instructions that belong together are separated during assembly.

The .CKxx macro generates CK.Rn register checking macros; that is, when you specify a register(s) as an argument to .CKxx, CKxx generates checking macro CK.Rn for that register(s).

## 1.5 Programmed Request Summary

Many programmed requests operate only in a specific RT–11 environment, such as under a multijob monitor or when using a special feature such as multiterminal operation. Table 1–4 lists the programmed requests that can be used in RT–11 environments, including multiterminal operation. Table 1–5 lists the additional programmed requests that can be used under the multijob monitors and mapped monitors. The EMT and function code for each request are shown in octal. Although only the first six characters of the programmed request are significant to the Macro assembler, the longer forms are shown to provide a better understanding of the request function. Also, the purpose of each request is briefly described.

Macros used in interrupt service routines and in writing device handlers are listed since they are a part of the system macro library.

Table 1–4 summarizes the programmed requests that work in all RT–11 environments.

**Table 1–4: Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .ABTIO | 374 | 13 | Aborts I/O in progress on the specified channel |
| .ADDR | – | – | Computes a specified address in a position-independent manner |
| .ASSUME | – | – | Tests for a specified condition; if test is false, generates assembly error and prints descriptive message |
| .BR | – | – | Warns if code which belongs together is separated during assembly |
| .CALLK | 373 | – | Transfers control (and alters mapping) from the current mode to the specified virtual address in Kernel mode |
| .CALLS | – | – | Supports transfer of control to Supervisor mode; works with SHANDL Supervisor handler instructions |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .CDFN | 375 | 15 | Defines additional channels for I/O |
| .CHAIN | 374 | 10 | Chains to another program (in background job only) |
| .CKxx | – | – | Generates CK.Rn register checking macros for one or more registers. |
| .CLOSE | 374 | 6 | Closes the specified channel |
| .CLOSZ | 375 | 45 | Closes the channel opened by .ENTER; sets file size. |
| .CMKT | 375 | 23 | Cancels an unexpired mark time request (Timer support). |
| .CSIGEN | 344 | – | Calls the Command String Interpreter (CSI) in general mode |
| .CSISPC | 345 | – | Calls the Command String Interpreter (CSI) in the special mode |
| .CSTAT | 375 | 27 | Returns the status of the specified channel |
| .CTIMIO | – | – | Used within a device handler as a macro call to cancel a mark time request (device timeout support) |
| .DATE | 374 | 12 | Moves the current date information into R0 |
| .DEBUG | – | – | Sets up .DPRINT environment. |
| .DELETE | 375 | 0 | Deletes the file from the specified device |
| .DEVICE | 375 | 14 | Enables device interrupts to be disabled upon program termination |
| .DPRINT | 351 | – | Inserts run-time messages in programs |
| .DRAST | – | – | Used with device handlers to create the asynchronous entry points to the handler |
| .DRBEG | – | – | Used with device handlers to create a header and additional information in .ASECT locations |
| .DRBOT | – | – | Used with system device handlers to set up the primary driver |
| .DRDEF | – | – | Used with device handlers to set up handler parameters, call driver macros from the library, and define useful symbols |
| .DREND | – | – | Used with device handlers to generate the table of pointers into the resident monitor |
| .DREST | – | – | Places device-specific information in block 0 of device handler |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .DRFIN | – | – | Used with device handlers to generate the code required to exit to the completion code in the resident monitor |
| .DRINS | – | – | Sets up installation code area in block 0 of a device handler, and defines system and data device installation entry points |
| .DRPTR | – | – | Places pointers in block 0 of device handler; pointers refer to service routines at address in that handler |
| .DRSET | – | – | Used with device handlers to create the list of SET options for a device |
| .DRSPF | – | – | Defines special function codes supported by handler |
| .DRTAB | – | – | Establishes file address of list of Digital-defined handler data tables |
| .DRUSE | – | – | Establishes file address of user-defined handler data tables |
| .DRVTB | – | – | Used with multivector device handlers to generate a table that contains the vector location, interrupt entry point, and processor status word for each device vector |
| .DSTAT | 342 | – | Returns the status of a specified device |
| .ENTER | 375 | 2 | Creates a new file for output |
| .EXIT | 350 | – | Exits the user program and optionally passes a command to KMON |
| .FETCH | 343 | – | Loads a device handler into memory |
| .FORK | – | – | Generates a subroutine call in an interrupt service routine that permits long but not critical processing to be postponed until all other interrupts are dismissed |
| .FPROT | 375 | 43 | Sets or removes a file's protection |
| .GFDAT | 375 | 44 | Returns in R0 the creation date from a file's directory entry |
| .GFINF | 375 | 44 | Returns in R0 the word contents of the directory entry offset specified in file's directory entry |
| .GFSTA | 375 | 44 | Returns in R0 the status information from a file's directory entry |
| .GTIM | 375 | 21 | Gets the time of day |
| .GTJB | 375 | 20 | Gets parameters of a job |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .GTLIN | 345 | – | Accepts an input line from either an indirect file or the console terminal |
| .GVAL | 375 | 34 | Returns contents of a monitor fixed offset |
| .HERR | 374 | 5 | Specifies termination of a job on fatal errors |
| .HRESET | 357 | – | Terminates I/O transfers and does a .SRESET operation |
| .INTEN | – | – | Generates a subroutine call to notify the monitor that an interrupt has occurred, requests system state, and sets processor priority to the specified value |
| .LOCK | 346 | – | Makes the monitor User Service Routine (USR) permanently resident until an .EXIT or .UNLOCK is executed; the user program is swapped out, if necessary |
| .LOOKUP | 375 | 1 | Opens an existing file for input and/or output via the specified channel; opens a message channel to a specified job |
| .MACS | – | – | Selects EMT expansions compatible with most current version (only if you've previously specified ..V1 or ..V2) |
| .MFPS | – | – | Reads the priority bits in the processor status word, but does not read the condition codes |
| .MRKT | 375 | 22 | Marks time; sets an asynchronous routine to be entered after specified interval |
| .MTATCH | 375 | 37 | Attaches a terminal for exclusive use by the requesting job |
| .MTDTCH | 375 | 37 | Detaches a terminal from one job and frees it for use by other jobs |
| .MTGET | 375 | 37 | Returns the status of a specified terminal to the user |
| .MTIN | 375 | 37 | Operates as a .TTYIN request for a multiterminal configuration |
| .MTOUT | 375 | 37 | Operates as a .TTYOUT request for a multiterminal configuration |
| .MTPRNT | 375 | 37 | Operates as a .PRINT request for a multiterminal configuration |
| .MTPS | – | – | Sets the priority bits, condition codes, and T-bit in the processor status word |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .MTRCTO | 375 | 37 | Operates as a .RCTRLO request for a multiterminal configuration |
| .MTSET | 375 | 37 | Modifies terminal status in a multiterminal configuration |
| .MTSTAT | 375 | 37 | Provides multiterminal system status. |
| .PEEK | 375 | 34 | Examines memory locations |
| .POKE | 375 | 34 | Changes memory locations |
| .PRINT | 351 | – | Outputs an ASCII string terminated by a zero byte or a $200_8$ byte |
| .PROTECT | 375 | 31 | Requests that specified vectors in the area from 0 to $476_8$ be given exclusively to the current job |
| .PURGE | 374 | 3 | Clears out a channel for reuse |
| .PVAL | 375 | 34 | Replaces contents of a monitor fixed offset |
| .QELDF | | | Used with device handlers to define offsets in the I/O queue element |
| .QSET | 353 | – | Increases the size of the monitor I/O queue |
| .RCTRLO | 355 | – | Enables output to the terminal; overrides any previous CTRL/O |
| { .READ .READC .READW } | 375 | 10 | Transfers data on the specified channel to a memory buffer and returns control to the user program:<br>• For .READ, when the transfer request is entered in the I/O queue; no special action is taken upon completion of I/O<br>• For .READC, when the transfer request is entered in the I/O queue; upon completion of the read, control transfers asynchronously to the completion routine specified in the .READC request<br>• For .READW, when the transfer is complete |
| .RELEASE | 343 | – | Removes a device handler from memory |
| .RENAME | 375 | 4 | Changes the name of the indicated file to a new name; an invalid operation for magtape |
| .REOPEN | 375 | 6 | Restores the parameters stored via a .SAVESTATUS request and reopens the channel for I/O |
| .RSUM | 374 | 2 | Causes the mainline code of the job to be resumed after it was suspended by a .SPND request |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .SAVESTATUS | 375 | 5 | Saves the status parameters of an open file in user memory and frees the channel for use |
| .SCCA | 375 | 35 | Enables intercept of CTRL/C commands |
| .SDTTM | 375 | 40 | Sets the system date and/or time |
| .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 |
| .SFDAT | 375 | 42 | Changes a file creation date in a directory entry |
| .SFINF | 375 | 44 | Returns in R0 the word contents of the directory entry offset specified in file's directory entry |
| .SFPA | 375 | 30 | Sets user interrupt for floating-point processor exceptions |
| .SFSTA | 375 | 44 | Returns in R0 the status information from a file's directory entry |
| SOB | – | – | Simulates the SOB instruction |
| .SPCPS | 375 | 41 | Used in a completion routine to change the flow of control of the mainline code (special feature) |
| .SPFUN | 375 | 32 | Performs special functions on magtape, cassette, diskette, and some disk devices |
| .SPND | 374 | 1 | Causes the running job to be suspended |
| .SRESET | 352 | – | Resets all channels and releases the device handlers from memory |
| .SYNCH | – | – | Generates a subroutine call that enables your program to perform programmed requests from within an interrupt service routine |
| .TIMIO | – | – | Generates a subroutine call in a handler to schedule a mark time request (timeout support) |
| .TLOCK | 374 | 7 | Indicates if the USR is currently used by another job and performs exactly as a .LOCK request |
| .TRPSET | 375 | 3 | Sets a user intercept for traps to monitor to vectors 4 and $10_8$ |
| .TTINR/.TTYIN | 340 | – | Reads one character from the keyboard buffer |
| .TTYOUT/.TTOUTR | 341 | – | Transfers one character to the terminal input buffer |
| .TWAIT | 375 | 24 | Suspends the running job for a specified amount of time |

**Table 1–4 (Cont.): Programmed Requests for RT–11 Environments**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .UNLOCK | 347 | – | Releases the USR after execution of a .LOCK and swaps in the user program, if required |
| .UNPROTECT | 375 | 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 |
| { .WRITC .WRITE .WRITW } | 375 | 11 | Transfers data on the specified channel to a device and returns control to the user program:<br>• For .WRITC, when the transfer request is entered in the I/O queue; upon completion of the read, control transfers asynchronously to the completion routine specified in the .WRITC request<br>• For .WRITE, when the transfer request is entered in the I/O queue; no special action is taken upon completion of I/O<br>• For .WRITW, when the transfer is complete |

Table 1–5 lists program requests that can be used only in a multijob and mapped environment.

**Table 1–5: Multijob or Mapped Program Requests**

| Name | EMT | Code | Purpose |
|---|---|---|---|
| .CHCOPY[3] | 375 | 13 | Enables one job to access another job's channel |
| .CMAP[2] | 375 | 46 | Controls separate I/D space, Supervisor mapping |
| .CNTXSW[3] | 375 | 33 | Requests that the indicated memory locations be part of the context switch process |
| .CRAW [1] | 375 | 36 | Creates a window in virtual memory |
| .CRRG [1] | 375 | 36 | Creates a region in extended memory |
| .ELAW [1] | 375 | 36 | Eliminates an address window in virtual memory |
| .ELRG [1] | 375 | 36 | Eliminates an allocated region in extended memory |
| .GCMAP[2] | 375 | 46 | Returns CMAP status |
| .GMCX [1] | 375 | 36 | Returns mapping status of a specified window |
| .MAP [1] | 375 | 36 | Maps a virtual address window to extended memory |
| .MSDS[2] | 375 | 46 | Controls lockstep of User data space and Supervisor data space |
| .MWAIT[3] | 374 | 11 | Waits for messages to be processed |
| { .RCVD[3] .RCVDC[3] .RCVDW[3] } | 375 | 26 | Receives data—enables a job to read messages or data sent by FG or BG job. The three modes correspond to the .READ, .READC, and .READW requests |
| .RDBBK [1] | – | – | Reserves space in a program for a region definition block and sets up the region size and region status word |
| .RDBDF [1] | – | – | Defines the offsets and bit names associated with a region definition block |
| { .SDAT .SDATC .SDATW } | 375 | 25 | Sends messages or data to the FG or BG job. The three modes correspond to the WRITE, .WRITC, and .WRITW requests |
| .UNMAP [1] | 375 | 36 | Unmaps a virtual address memory window |
| .WDBBK [1] | – | – | Reserves space in a program for a window definition block and sets up the associated data |
| .WDBDF [1] | – | – | Defines the offsets and bit names associated with a window definition block |

[1]Single-mapped
[2]Fully-mapped
[3]Multijob

## Chapter 2

# Programmed Request Description and Examples

This chapter presents the programmed requests alphabetically, describing each one in detail and providing an example of its use in a program. Also described are macros and subroutines that are used to implement device handlers and interrupt service routines. 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** Pointer to the EMT argument block for those requests that require a block.

**blk** Block number specifying the relative block in a file or device where an I/O transfer is to begin.

**buf** Buffer address specifying a memory location into which or from which an I/O transfer will be performed. This address has to be word-aligned; that is, located at an even address.

**cblk** Address of the five-word block where channel status information is stored.

**chan** Channel number in the range 0-$376_8$.

**chrcnt** Character count in the range 1-$255_{10}$.

**code** Flag used to indicate whether the code is to be set in an EMT 375 programmed request.

**crtn** Entry point of a completion routine.

**dblk** Four-word Radix–50 descriptor block that specifies the physical device, file name, and file type to be operated upon.

**dnam** One-word RAD–50 device name; can be first of four-word *dblk*

**func** Numerical code indicating the function to be performed.

**jobblk** Pointer to a three-word ASCII system job name.

**jobdev** Pointer to a four-word system job descriptor where the first word is a Radix–50 device name and the next three words contain an ASCII system-job name (For keyword argument use, refer to this as a *dblk*).

**num** Number, whose value depends on the request.

**seqnum** File number.

For magtape operation, this argument describes a file sequence number. The values that the argument can have are described under the applicable programmed requests.

**unit** Logical unit number of a particular terminal in a multiterminal system.

**wcnt** Word count specifying the number of words to be transferred to or from the buffer during an I/O operation.

Many programmed requests require support only available if you have selected them during the SYSGEN process. Therefore, at SYSGEN, you should anticipate the special support you will need in addition to those normally provided in a distributed monitor.

# .ABTIO

**EMT 374, Code 13**

The .ABTIO programmed request allows a job to abort all outstanding I/O operations on a channel without terminating the program.

When .ABTIO is issued, the handler for the device opened on the specified channel is entered at its abort entry point. After the handler abort code is executed, control returns to the user program.

**NOTE**

.ABTIO does not necessarily abort I/O for certain devices. It will not abort another program's I/O.

Macro Call:

**.ABTIO chan**

where:

**chan** is the channel number on which to abort I/O

Request Format:

| R0 = | 13 | chan |
|---|---|---|

Errors:

None

Example:

```
        .TITLE  EABTIO.MAC

;This is an example of the .ABTIO request.  The .ABTIO request
;is useful for terminating .READC/.WRITC or .READ/.WRITE I/O on
;a particular channel without issuing a .EXIT or .HRESET, which
;would terminate the program or stop I/O on all channels.

        .MCALL  .ABTIO  .ENTER  .EXIT
        .MCALL  .READ   .SCCA   .WAIT

START:  .SCCA   #AREA,#CTCWRD           ;Inhibit ^C
        .ENTER  #AREA,#1,#FILNAM        ;Open chan 1 as input file
IOLOOP: .WAIT   #1                      ;Wait for last I/O
        .READ   #AREA,#1,#BUF,#256.,#0  ;Read a block
        ...                             ;Process
        TST     CTCWRD                  ;^C^C done?
        BPL     IOLOOP                  ;No
        .ABTIO  #1                      ;Abort all I/O on the channel
        ...
        .EXIT

AREA:   .BLKW   5.                      ;Request block area
FILNAM: .RAD50  "BINEABTIOSAV"          ;File to read
CTCWRD: .BLKW   1                       ;Terminal status word
BUF:    .BLKW   256.                    ;Buffer area
        .END    START
```

# .ADDR

**Macro Expansion**

The .ADDR macro computes the specified address in a position-independent manner. The address computed is a run-time location stored in a register or on the stack.

Macro Call:

**.ADDR addr,reg,push**

where:

**addr** is the label of the address to compute, expressed as an immediate value with a number sign (#) before the label.

**reg** is the register in which to store the computed address, expressed as a register reference Rn or @Rn.

To store the address on the stack, use @SP or -(SP). A @SP stores the address in the stack's current top, while -(SP) pushes the address onto a new location which becomes the top of the stack. The following register references are valid:

| | | |
|---|---|---|
| R0 | @R0 | |
| R1 | @R1 | @SP |
| R2 | @R2 | -(SP) |
| R3 | @R3 | |
| R4 | @R4 | |
| R5 | @R5 | |

**push** determines what to do with the original contents of the register. If you omit push, the computed address overwrites the register contents. If you use ADD for the push argument, the computed address is added to the original contents of the register. If you use PUSH for the push argument, the register's previous contents are pushed onto the stack before the computed address is stored in the register.

If you use -(SP) for the argument *reg*, you may omit the push argument, since PUSH is automatically used.

The following sample lines from a program show all three uses of the .ADDR macro:

```
.TITLE  EXADDR.MAC

.ADDR   #ABC,R0          ;Load address of ABC in R0
.ADDR   #ABC,R1,ADD      ;Add address of ABC into R1
.ADDR   #ABC,R2,PUSH     ;Push contents of R2 onto stack
                         ;then load address of ABC into R2
```

# .ASSUME

### Macro Expansion

The .ASSUME macro tests the validity for a condition you specify. If the test is false, MACRO generates an assembly error and prints a descriptive message. At assembly time, both .ASSUME and .BR check assertions that you make; they do no checking at run time.

Macro Call:

**.ASSUME a rel c [message = <text>]**

where:

| | |
|---|---|
| **a** | is an expression. |
| **rel** | is the relationship between a and c you want to test. There can be six values for *rel*: *eq, ne, gt, et, ge,* and *le*. |
| **c** | is an expression. |
| **text** | is the message you want MACRO to print if the condition you specified in the relationship between *a* and *c* is false. To specify your own error message, start the message with a semicolon (;), or start with a valid assembly expression followed by a semicolon (;) and the message. If you omit the message argument, the error message *a rel c is not true* displays; the expressions you used appear in the message in place of *a* and *c*. |

In the following example, if the location counter (.) is less than 1000, MACRO generates an assembly error and prints the message 1000 - .; location too high.

```
.TITLE EXASSU.MAC
        .ASSUME . LT 1000 Message=^/1000-.;location too high/
```

# .BR

**Macro Expansion**

The .BR macro warns you during assembly time if code that belongs together is separated. When you invoke the .BR macro, you specify an address as an argument. .BR checks that the next address equals the address you specified in the .BR macro. If it does not, MACRO prints the error message: *ERROR; ?SYSMAC-E-Not at location addr*. The location you specified in the .BR macro appears in place of *addr* in the message. If you specify a symbol as an argument and the symbol is not defined in the current assembly, you will get an error message: *ERROR; ?SYSMAC-E-addr is not defined*. At assembly time, both .ASSUME and .BR check assertions that you make; they do no checking at run time.

Macro Call:

**.BR addr**

where

**addr** is the address you want to test.

In the following example, MACRO tests the location that follows the .BR macro. Since the address does not match the address ABC, MACRO prints an error message.

```
        .TITLE  EXBR1.MAC

        .BR     ABC     ;Test next addr for ABC
.PAGE
FOO:
        TST     R0
ABC:
```

In the next example, no error occurs:

```
        .TITLE  EXBR2.MAC

        .BR     DEF             ;test next addr for DEF
.PAGE
DEF:
```

In the next example, because UNDEF is not defined, an error is reported:

```
        .TITLE  EXBR3.MAC

        .BR     UNDEF           ;undefined label
```

# .CALLK

**EMT 373**

The .CALLK request transfers control (and alters mapping) from the current mode to the specified virtual address in Kernel mode. The .CALLK request is especially useful when a program is running in User mode and needs to execute a monitor routine that can be called only from Kernel mode. Although .CALLK is supported under all monitors, it has meaning only under the mapped monitors, as these support multiple address spaces.

Macro Call:

**.CALLK [dest][,pic]**

where:

**dest** is a virtual address in Kernel mode; the address of the entry point to the routine. If *dest* is not specified, .CALLK assumes the address is on the stack

**pic** is an optional parameter that should be non-blank if the program that invokes .CALLK is written in position-independent (PIC) code. Device handlers, for example, are written in PIC code and therefore a .CALLK request in a device handler must specify this parameter. If *pic* is specified, the virtual address specified for the *dest* parameter must be specified in the form #address, or an assembly error is generated

The following monitor routines can be called from User mode by using .CALLK:

**$BLKMV, FINDGR, $JBLOC, $MPMEM, $P1EXT, $USRPH, XALLOC, XDEALC.**

The environment upon entry into Kernel mapping is as follows:

- Registers 0-5 are preserved across the change to Kernel mode.
- The condition code bits, trace trap bit, and previous mode bit in the PS are not preserved.
- The contents of the User and Kernel stacks during the mode change are not defined.
- The User mapped system communications area (SYSCOM) is not mapped to Kernel.

If a routine called in Kernel mode causes the SYSCOM area to change, that change must be processed before the return to User mapping. For example, if a routine places a value in $ERRBY, code must process that value before the routine returns.

The routine called in Kernel mode must, once it executes, return using the standard RETURN (RTS PC) instruction. After execution of the RETURN instruction, the environment upon return to User mapping is:

- Registers 0-5 are preserved across the change to User mode.

- The condition codes in the PS are preserved.
- The trace trap bit and previous mode bit in the PS are not preserved.
- The stack pointer (SP) and stack contents are the same as before the call to .CALLK, except that the destination address has been popped off the stack.

If .CALLK is invoked by a program running in Kernel mode, .CALLK performs as if it was invoked from within User mode. When the routine called after invoking .CALLK returns, the environment upon return to Kernel mode is the same as the return environment documented for User mode.

Errors:
None reported by .CALLK; however, the called routine may report errors.

**WARNING**

Calling unsupported addresses in Kernel mode may crash the system.

Example:

The following program fragment illustrates using .CALLK. The program is running in virtual User mapping and the code illustrates transferring control to Kernel mapping to use the monitor routine $BLKMV to perform a block move operation:

```
.TITLE  ECALLK.MAC

.GVAL   #AREA,#P1$EXT   ;Get RMON’s P1$EXT offset
MOV     R0,-(SP)        ;save it
ADD     #$BLMPT,@SP     ;Point to the block move routine
MOV     INPAR,R1        ;Input PAR value
MOV     INOFST,R2       ;Input PAR address (normalized)
MOV     OUPAR,R3        ;Output PAR value
MOV     OUOFST,R4       ;Output PAR address (normalized)
MOV     #WCOUNT,R5      ;Count of words to be moved
.CALLK                  ;Call the address on stack
                        ;.CALLK will pop the address from stack
```

# .CALLS

The .CALLS request supports transferring control to Supervisor mode. It is designed to work with the SHANDL Supervisor handler code.

Macro Call:

**.CALLS dest,return**

where:

**dest** Supervisor virtual address (#xxxxxx) to which you are transferring control.

**return** Character string designating condition code values that should be preserved for return transition from Supervisor mode to User mode. Default value is NZVC, meaning return all condition codes.

If no condition codes need to be returned, specify RETURN=<>. If only carry needs to be returned, specify RETURN=C. Any combination of condition codes may be specified.

### Notes

The .CALLS macro generates the following:

```
.TITLE  ECALS1

        MOV     #ccodes*2,-(SP)
        CSM     dest
```

The *ccodes*2* value is a mask based on the condition codes specified to the second argument.

.CALLS is used as a transfer vector, not as an inline call. For example, to transfer control to a supervisor routine called FRANK, use the following:

```
.TITLE  ECALS2

        ...
HEY:    CALL        FRANKS    ;Call FRANK in supervisor mode
        ...

FRANKS: .CALLS   #FRANK,RETURN=^// ;Transfer to FRANK in supy,
                                  ;return no condition codes
```

Note that control is passed to the instruction following HEY when FRANK returns, not to the instruction following FRANK2.

.CALLS also has a special form, .CALLS #0, by which you can transfer control to Supervisor mode and perform an RTI instruction (with Supervisor mode as the "previous" mode). For example:

```
.TITLE  ECALS3

        MOV     #NewPS,-(SP)    ;New PS to use in Supy
        MOV     #NewPC,-(SP)    ;New PC ...
        .CALLS  #0              ;Go to Supervisor mode
```

## .CDFN

**EMT 375, Code 15**

The .CDFN request redefines the number of I/O channels. Each job, whether foreground or background, is initially provided with $16_{10}$ I/O channels numbered 0-15 (0-17 octal). .CDFN allows the number to be expanded to as many as $255_{10}$ channels (0-254 decimal, or 0-376 octal). Channel 377 is reserved for use by the monitor.

The space for the new channels must be allocated by the User program. Each I/O channel requires five words of memory. Therefore, you must allocate 5*n words of memory, where n is the number of channels to be defined.

If the program is run under VBGEXE, the space for the new channels is allocated from memory controlled by VBGEXE and the address passed by the user program is not used.

It is recommended that you use the .CDFN request 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 channels to revert to the original 16 channels defined at program initiation. Hence, you must reissue any .CDFNs after using .SRESET or .HRESET. The keyboard monitor command CLOSE does not work if your program defines new channels with the .CDFN request.

The .CDFN request defines new channels so that the space for the previously defined channels cannot be used. Thus, a CDFN for $24_{10}$ channels (while 16 original channels are defined) creates 24 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.

If a program is overlaid, the overlay handler uses channel $17_8$ and this channel should not be modified. (Other channels can be defined and used as usual.)

In a mapped monitor environment, the area supplied for additional channels specified by the .CDFN request must lie in the lower 28K words of memory. In addition, it must not be in the virtual address space mapped by Kernel PAR1, specifically the area from 20000 to $37776_8$. If you supply an invalid area, the system generates an error message.

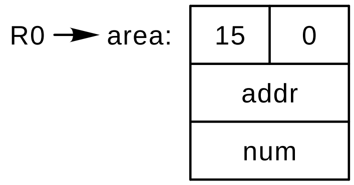
Macro Call:

**.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:




Errors:

| Code | Explanation |
|---|---|
| 0 | An attempt was made to define fewer than or the same number of channels that already exist. In an mapped environment, an attempt to violate the PAR1 restriction sets the carry bit and returns error code 0 in byte 52. |

Example:

```
        .TITLE  EXCDFN.MAC

;+
; .CDFN - This is an example in the use of the .CDFN request. The
; example defines 32 new channels to reside in the body of the
; program.
;-

        .MCALL  .CDFN,.PRINT,.EXIT

$USRRB  =:      53                      ;(.SYCDF) user error byte
 SUCCS$ =:      001                     ;(.UEBDF) success "error" bit
 FATAL$ =:      010                     ;(.UEBDF) fatal error bit

C.SIZ   =:      12                      ;(.CHNDF) size of a channel in bytes

START:  .CDFN   #AREA,#CHANL,#32.       ;Use .CDFN to define 32. new channels
        BCC     1$                      ;Branch if successful
        .PRINT  #BADCD                  ;Print failure message on console
        BISB    #FATAL$,@#$USRRB        ;Indicate error
        .EXIT                           ;Exit program
1$:     .PRINT  #GOODCD                 ;Print success message
        BISB    #SUCCS$,@#$USRRB        ;Indicate success
        .EXIT                           ;Then exit

AREA:   .BLKW   3                       ;EMT Argument Block
CHANL:  .BLKW   C.SIZ/2*32.             ;Space for new channels

BADCD:  .ASCIZ  /?ECDFN-F-.CDFN Failed/ ;Failure message
GOODCD: .ASCIZ  /!ECDFN-I-.CDFN Successful/ ;Success message

        .END    START
```

# .CHAIN

**EMT 374, Code 10**

The .CHAIN request lets a background program pass control directly to another background program without operator intervention. Since this process can be repeated, a long "chain" of programs can be strung together.

The chain area consists of locations 500-777 of the running job's virtual address space, which may or may not be at the low memory locations 500-777. For that reason, you should not use .PEEK or .POKE requests when referencing the chain area. Instead, use standard PDP–11 instructions, such as MOV, that access memory directly.

Macro Call:

**.CHAIN**

Request Format:

| | | |
|---|---|---|
| R0 = | 10 | 0 |

**Notes**

- Make no assumptions about which areas of memory remain intact across a .CHAIN. In general, only the resident monitor and locations 500-777 are preserved across a .CHAIN. In many programs stack begins at 1000 and expands downward. The .CHAIN operation does not protect from stack expansion; therefore, some locations between 500-777 may be corrupted by the stack.
- 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, nonresident device handlers are released during a .CHAIN request and must be fetched again by the new program. Note that FORTRAN logical units do not stay open across a .CHAIN.
- An executing program determines whether it was chained to or RUN from the keyboard by examining bit 8 of the Job Status Word. The monitor sets this bit if the program was invoked with .CHAIN request. 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 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, .CHAIN is abandoned and the keyboard monitor is entered.

When using .CHAIN, be careful with initial stack placement. The linker normally defaults the initial stack to $1000_8$; if caution is not observed, the stack can destroy chain data before it can be used.

Example:

```
        .TITLE  ECHAIN.MAC
;+
; .CHAIN - This example demonstrates the use of the .CHAIN
; program request. It chains to program 'CTEST.SAV' and passes it
; a command line typed in at the console terminal. As an exercise
; write the program 'CTEST' - in it, check to see if it was chained
; to, and if so, echo the data passed to it, otherwise print the
; message "Was not chained to".
;-
        .MCALL  .CHAIN,.GTLIN

NOCRLF  =:      200                     ;String terminator for no CRLF

CH.PGM  =:      500                     ;(.CHADF) Program DBLK in chain area

START:  MOV     #CH.PGM,R1              ;R1 => Chain area
        MOV     #CHPTR,R2               ;R2 => RAD50 Program Filespec
        .REPT   4                       ;Move the Program Filespec
        MOV     (R2)+,(R1)+             ;into the Chain area...
        .ENDR                           ;
        .GTLIN  R1,#PROMT               ;Now get a "command" line
        .CHAIN                          ;Chain to the next program.

CHPTR:  .RAD50  "BIN"                    ;RAD50 File spec...
        .RAD50  "ECTEST"
        .RAD50  "SAV"
PROMT:  .ASCII  "Enter data to be passed to ECTEST > "
        .BYTE   NOCRLF                  ;treat as prompt (no CRLF)
        .END    START
```

```
;
;* IN CASE YOU DON'T HAVE TIME HERE'S AN EXAMPLE *
;* 'ECTEST.MAC' PROGRAM...
;

        .TITLE  ECTEST.MAC

        .MCALL  .PRINT,.EXIT

$JSW    =:      44                      ;(.SYCDF) Location of JSW
 CHAIN$ =:     400                      ;(.JSWDF) CHAIN bit in JSW

CH.ARG  =:     510                      ;(.CHADF) CHAIN argument

START:  BIT     #CHAIN$,@#$JSW          ;Were we chained to?
        BEQ     1$                      ;Branch if not
        .PRINT  #CHAIND                 ;Say we were...
        MOV     #CH.ARG,R0              ;Get addr of start of data
        .PRINT                          ;Print it out
        .EXIT                           ;Exit program
1$:     .PRINT  #NOCHN                  ;Say we weren't chained to
        .EXIT                           ;Then exit
CHAIND: .ASCIZ  /!ECTEST-I-was chained to - and here's the data passed.../
NOCHN:  .ASCIZ  /!ECTEST-I-was not chained to/
        .END    START
```

# .CHCOPY

**Multijob**
**EMT 375, Code 13**

The .CHCOPY request opens a channel for input, logically connecting it to a file that is currently open by another job. This request can be used by a foreground, background, or system job and must be issued before the first .READ or .WRITE request on that channel.

.CHCOPY is valid only on files on disk. 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 [,jobblk]**

where:

| | |
|---|---|
| **area** | is the address of a three-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 |
| **jobblk** | is a pointer to a three-word ASCII logical job name that represents a system job |

Request Format:

| R0 → area: | 13 | chan |
|---|---|---|
| | ochan | |
| | jobblk | |

**Notes**

- 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.
- A channel open on a non-file-structured device should not be copied, because I/O from separate jobs will most likely become confused.
- 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.

- Foreground and background jobs optionally may leave the *jobblk* argument blank or set it to zero. This causes the job name to default to *F* if the background job issued the request, or to *B* if the foreground job issued the request.

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  ECHCOF;2
;+
; This is the Foreground program ...
;-
        .MCALL  .ENTER,.PRINT,.SDATW,.EXIT,.RCVDW,.CLOSE,.WRITW
.MACRO  ...
.ENDM

STARTF: MOV     #AREA,R5                ;R5 => EMT argument block
        .ENTER  R5,#0,#FILE,#5          ;Create a 5 block file
        .WRITW  R5,#0,#RECRD,#256.,#4   ;Write a record BG is interested in
        BCS     ENTERR                  ;Branch on error
        .SDATW  R5,#BUFR,#2             ;Send message with info to BG
        ...                             ;Do some other processing
        .RCVDW  R5,#BUFR,#1             ;When it's time to exit, make sure
        .CLOSE  #0                      ;BG is done with the file
        .PRINT  #FEXIT                  ;Tell user we're exiting
        .EXIT                           ;Exit the program
ENTERR: .PRINT  #ERMSG                  ;Print error message
        .EXIT                           ;then exit
FILE:   .RAD50  /DK ECHCOF/             ;File spec for .ENTER
        .RAD50  /TMP/
AREA:   .BLKW   5                       ;EMT argument block
BUFR:   .WORD   0                       ;Channel #
        .WORD   4                       ;Block #
RECRD:  .BLKW   256.                    ;File record
ERMSG:  .ASCIZ  /?ECHCOF-F-Enter Error/ ;Error message text
FEXIT:  .ASCIZ  /!ECHCOF-I-FG Job exiting/ ;Exit message
        .END    STARTF
```

```
        .TITLE  ECHCOB.MAC
;+
; This is the Background program ...
;-
        .MCALL  .CHCOPY,.RCVDW,.READW,.EXIT,.PRINT,.SDATW
.MACRO  ...
.ENDM
$USRRB  =:      53                      ;(.SYCDF) user error byte
 SUCCS$ =:      001                     ;(.UEBDF) success "error" bit
 FATAL$ =:      010                     ;(.UEBDF) fatal error bit
STARTB: MOV     #AREA,R5                ;R5 => EMT arg block
        .RCVDW  R5,#MSG,#2              ;Wait for message from FG
        BCS     1$                      ;Branch if no FG
        .CHCOPY R5,#0,MSG+2             ;Channel # is 1st word of message
        BCS     2$                      ;Branch if FG channel not open
        .READW  R5,#0,#BUFF,#256.,MSG+4 ;Read block which is 2nd word of msg
        BCS     3$                      ;Branch if read error
        ...                             ;Continue processing...
        .SDATW  R5,#MSG,#1              ;Tell FG we're thru with file
        .PRINT  #BEXIT                  ;Tell user we're thru
        BISB    #SUCCS$,$USRRB          ;Indicate success
        .EXIT                           ;then exit program
1$:     MOV     #NOJOB,R0               ;R0 => No FG error msg
        BR      4$                      ;Branch to print msg
2$:     MOV     #NOCH,R0                ;R0 => FG ch not open msg
        BR      4$                      ;Branch...
3$:     MOV     #RDERR,R0               ;R0 => Read err msg
4$:     .PRINT                          ;Print proper error msg
        BISB    #FATAL$,$USRRB          ;Indicate failure
        .EXIT                           ;then exit.
AREA:   .BLKW   5                       ;EMT argument blk
MSG:    .BLKW   3                       ;Message buffer
BUFF:   .BLKW   256.                    ;File buffer
BEXIT:  .ASCIZ  /!ECHCOB-I-Channel-Record copy successful/
NOJOB:  .ASCIZ  /?ECHCOB-F-No FG Job/   ;Error messages...
NOCH:   .ASCIZ  /?ECHCOB-F-FG channel not open/
RDERR:  .ASCIZ  /?ECHCOB-F-Read Error/
        .END    STARTB
```

# .CKXX

The .CKXX macro generates CK.Rn register checking macros. When you specify a register as an argument to .CKXX, .CKXX creates the CK.Rn checking macro for that register. When you specify more than one register for .CKXX, .CKXX creates a CK.Rn checking macro for each register. Similarly, more than one CK.Rn checking macro can be created for a register.

Using CK.Rn macro simplifies the checking of assumptions about registers that are used in autoincrement and autodecrement mode instructions. You can also assign symbols to CK.Rn that can be used to store register contents during autoincrement and autodecrement operations. Those symbols can later be used to verify the position of the stored values.

Macro Call:

**.CKXX <reg[,alph][,reg[,alph]...]>**

where:

**reg** is the register or registers you want .CKXX to define as check registers.

Calling .CKXX generates a CK.Rn check macro for each register you include in the *reg* argument. For example,

Macro Calls:

**.CKXX <R0>** Generates the check macro CK.R0.

**.CKXX <R0,R1>** Generates the check macros CK.R0 and CK.R1. To generate more than one check macro for a single register (for example, R1), append a different letter to the register number for each check macro you want to create. For example,

**.CKXX <R1A,R1B>** Generates the check macros CK.R1A and CK.R1B.

The check macro CK.Rn, generated by .CKXX, has the following form:

Form:

**CK.Rn[alph] [label][,change][,result]**

where:

**n** is the register number that .CKXX assigned to this check macro

**alph** is an alphabetic character that .CKXX assigned to this check macro

**label** is the value or label you assume equates to the check register.

If the value or label does not equate to the check register, a P error is returned at assembly time. See the *PDP–11 MACRO–11 Language Reference Manual*, Appendix D, for a description of the P assembly error

**change** indicates the check macro increment or decrement.

The change value must be preceded by a plus sign (+) to indicate increment or a minus sign (-) to indicate decrement. When the change is an increment, any assumption is checked first and the check macro is then incremented. When the change is a decrement, the check macro is first decremented and then any assumption is checked.

**result** is a new location assigned to the check macro.

Use the *result* argument to assign a symbol to the check macro. When you want to verify later that the check macro equates to that symbol, specify that symbol in the *label* argument

The check macro for a register must track exactly the operations done on that register; that is, the register's check macro must be explicitly incremented or decremented when the register is incremented or decremented. For example,

- Assigning an initial value to the check macro
- Transferring a value from one check macro to another
- Checking the current value of a register pointer and tracking for autoincrement
- Tracking for auto-decrement and then checking the current value of a register pointer (decrement performed first)
- Tracking for auto-increment and auto-decrement without checking for values

For example, assume the following data block:

```
        .TITLE  ECKXX.MAC

DBLK:   .BLKW   4
F.DEV   =:      0          ;(.DBKDF) device name in DBLK
F.NAME  =:      2          ;(.DBKDF) file name in DBLK
F.TYPE  =:      6          ;(.DBKDF) file type in DBLK

.MCALL  .CKXX              ; Call .CKXX
        .CKXX   R3         ; Create CK.R3
        .CKXX   R4         ; Create CK.R4
        ...
MOV     #DBLK,R3           ; point to data block DBLK
        CK.R3=F.DEV        ; assign initial value to check macro
MOV     R3,R4              ; copy the pointer
        CK.R4=CK.R3        ; copy the check macro to new one
        CK.R3 F.DEV,+2     ; check R3 equates to DEV
                           ;  and increment
MOV     (R3)+,R0           ; load device name into R0
        CK.R3 F.NAME,+2    ; check R3 equates to NAME
                           ;  and increment
MOV     (R3)+,NAME+0       ; get first part of file name
        CK.R3 ,+2          ; increment but no check (no label)
MOV     (R3)+,NAME+2       ; get last part of file name
        CK.R3 F.TYPE       ; check R3 equates to TYPE
                           ;  but no increment
MOV     @R3,R2             ; filespec extension into R2
        CK.R3 F.NAME+2,-2  ; decrement and check R3 equates
                           ;  to NAME+2
TST     -(R3)              ; test last 3 chars of filespec
        CK.R3 F.DEV,-2-2   ;  are they blank (0 in RAD50)?
CMP     -(R3),-(R3)        ; skip back to device
        ...
```

# .CLOSE

**EMT 374, Code 6**

The .CLOSE request terminates activity on the specified channel and frees it for use in another operation.

Macro Call:

**.CLOSE chan**

where:

**chan** is a channel number in the range 0 to $376_8$

Request Format:

| R0 = | 6 | chan |
|---|---|---|

Under certain conditions, a handler for the associated device and USR must be available when issuing a .CLOSE for files opened with either .ENTER or .LOOKUP:

- .CLOSE requires a handler and USR, if it is:
  - A special directory device (magtape).
  - An RT–11 standard directory device, and the file was opened with an .ENTER.
- All other RT–11 operations do not require either handler or USR.

USR is always in memory when a mapped monitor is selected. The handler for an associated device must be in memory if a a channel was established by the .ENTER. A .CLOSE is required on any channel opened with .ENTER if the associated file is to become permanent.

When issuing a .CLOSE, files opened with .LOOKUP do not require any directory operations and the USR does not have to be in memory. However, USR is required if, while the channel is open, a request was issued that required directory operations. The USR is always required for special structured devices such as magtape.

**NOTE**

Do not close channel $17_8$ if your program is overlaid, because overlays are read on that channel.

A .CLOSE performed on a file opened with .ENTER causes the device directory to be updated to make that file permanent. The first permanent file in the directory with the same name, if one exists, is deleted, provided that it is not protected. When a file that is opened with an .ENTER request 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.

For information about closing a file with a size other than with the default just described, see the .CLOSZ program request.

In magtape operations, the .CLOSE request causes the handler to write an ANSI EOF1 label in software mode (using MM.SYS, MT.SYS, MU.SYS or MS.SYS) and to close the channel in hardware mode (using MMHD.SYS, MUHD.SYS, MTHD.SYS or MSHD.SYS).

Errors:

| Code | Explanation |
|---|---|
| 3 | A protected file with the same name already exists on the device. The .CLOSE is performed anyway, resulting in two files with the same name on the device. |

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:

Refer to the example for the .READW, which shows typical uses for .CLOSE.

# .CLOSZ

**EMT 375, Code 45**

The .CLOSZ programmed request terminates activity on a channel that was opened by a .ENTER, frees it for use in another operation, and sets the file size. The device handler for the associated channel must be loaded in memory if the file was opened with a .ENTER request or if the .DRDEF macro used to build the handler was marked SPECL$. The .CLOSZ request has no effect on file size when a file was opened by a .LOOKUP request.

Macro Call:

**.CLOSZ area,chan,size**

where:

**area** is the address of a 2-word EMT argument block

**chan** is a channel number in the range of 0 to 376(octal). If the channel is not open, the request is ignored.

**size** is the specified size of the file at closing. Valid values for *size* are determined by whether the handler is RT–11 directory structured or special directory structured:

- If the handler is RT–11 directory structured, *size* must be less than or equal to the allocated file size; the file can only remain unchanged or become smaller.
- If the handler is special directory structured, *size* can be any value. RT–11 imposes no limits on size. The handler may independently impose rules on the closed file size. Magtape handlers treat a .CLOSZ request as a .CLOSE request.

Request Format:

R0 → area:

| 45 | chan |
|---|---|
| size | |

Errors:

If channel was opened to an RT–11 *directory device*,

| Code | Explanation |
|---|---|
| 1 | Size argument is greater than allocated size; file closed at size indicated by highest block written (equivalent to .CLOSE) |
| 2 | Channel not opened with .ENTER; channel purged |

| | |
|---|---|
| 3 | A protected file with the same name already exists on the device. File is closed with size as indicated by size argument. (If error 1 and error 3 conditions exist at same time, error 1 takes precedence) |

Errors:

If channel was opened on a special directory device,

| Code | Explanation |
|---|---|
| 1 | Meaning controlled by handler |
| 2 | Channel not opened by .ENTER; channel purged |
| 3 | Meaning controlled by handler |

Example:

```
        .TITLE  ECLOSZ.MAC

.MCALL        .ENTER  .WRITW  .CLOSZ  .EXIT

.MACRO       ...
.ENDM        ...

START::
;+
;            Use a new output file, which has had extra space allocated
;            for it as a temp work area, then truncate the file to its
;            desired size.
;-
             ...
             .ENTER  #AREA,#0,#FILE,#SIZE+10.     ;create file w/extra space
             BCS     ERROR
             ...
             .WRITW  #AREA,#0,#BUF,#400,#SIZE+10.-1  ;use temp space
             BCS     ERROR
             ...
             .CLOSZ  #AREA,#0,#SIZE                 ;close file at final
             ...                                     ; size
ERROR:
             ...
             .EXIT
AREA:        .BLKW   5.
FILE:        .RAD50  "DK TEST  TMP"
BUF:         .BLKW   400

SIZE         =:      20.

             .END    START
```

# .CMAP/.CMPDF/.GCMAP

**EMT 375, Code 46, Subcode 1**

**.GCMAP**

Issue the .GCMAP request to return the CMAP status. The value is returned in R0. This value is not implemented in unmapped monitors.

Macro Call:

**.GCMAP area,CODE=strg**

where:

| | |
|---|---|
| **area** | is the address of a two-word EMT request block area |
| **CODE=strg** | specifies *strg* as either "SET" (default), "NOSET", "SP" or "STACK" |

In Supervisor mode when you want to establish your own data space, distinct from User data space, you may not own any data space memory. Therefore, you can't use use standard request code. .CMAP, .GCMAP and .MSDS introduce a concept that allows you to specify CODE = "SP" or "STACK". In this way, you use "STACK" to:

- Build a request block on the stack
- Issue the request
- Clear the stack of the request

Errors:
None.

**.CMAP**

The .CMAP request sets the "CMAP" status and returns the old value in R0. This request is not implemented in unmapped monitors.

Macro Call:

**.CMAP area,value,CODE=strg**

where:

| | |
|---|---|
| **value** | is the setting desired |
| **CODE=strg** | Specify *strg* as either "SET" (default), "NOSET", "SP" or "STACK" |

In Supervisor mode when you want to establish your own data space, distinct from User data space, you may not own any data space memory. Therefore, you can't use use standard request code. .CMAP, .GCMAP and .MSDS introduce a concept so that you can specify CODE = "SP" or "STACK". In this way, you use to "STACK" to:

- Build a request block on the stack
- Issue the request

- Clear the stack of the request

### .CMPDF

Definition macro .CMPDF defines the bit pattern for .CMAP, .GCMAP, and .MSDS requests and for a field in the impure area.

The .CMAP programmed request writes the I.CMAP word that controls a job's mapping context. The system uses the job's mapping context to determine which RCBs, PARs and WCBs are supported for User, Supervisor and Kernel processor modes. This information is used when context switching those structures in and out of the processor's memory management unit.

The word is divided into 1 flag and 4 fields. The flag (CM.DUS) indicates if individual PARs are to be separately mapped. The fields each track a particular aspect of a job's mapping context. Refer to Table 2–1. The following bits are defined; any undefined bits are reserved by Digital.

**Table 2–1: Change Mapping Context (I.CMAP) Word Bits**

| Bit Mask | Symbol | Meaning |
|---|---|---|
| 000377 | CM.PAR | PAR selection byte.<br>CM.DUS determines if this field is active. If CM.DUS is set, the field is active and the bit mask in CM.PAR indicates which PARs are to be separately mapped. |
| 000001 | CM.PR0 | Separate PAR0. |
| 000002 | CM.PR1 | Separate PAR1. |
| 000004 | CM.PR2 | Separate PAR2. |
| 000010 | CM.PR3 | Separate PAR3. |
| 000020 | CM.PR4 | Separate PAR4. |
| 000040 | CM.PR5 | Separate PAR5. |
| 000100 | CM.PR6 | Separate PAR6. |
| 000200 | CM.PR7 | Separate PAR7. |
| 001400 | CM.S | Supervisor mode I & D Separation Field.<br>High bit indicates if this field is active. If active, low bit indicates the action to be taken. |
| 001000 | CM.SXX | Change current Supervisor mode support |
| 001000 | CM.SII | Non-separate Supervisor I & D space. |
| 001400 | CM.SID | Separate Supervisor I & D space. |
| 002000 | | Reserved. |
| 004000 | CM.DUS | Separate data space by PAR. |

**Table 2–1 (Cont.): Change Mapping Context (I.CMAP) Word Bits**

| Bit Mask | Symbol | Meaning |
|---|---|---|
| 030000 | CM.SUP | Supervisor mode support (context switching) field.<br>High bit indicates if this field is active. If active, low bit indicates the action to be taken. |
| 020000 | CM.XXS | Change current Supervisor mode support: |
| 020000 | CM.NOS | No Supervisor mode context switching. |
| 030000 | CM.JAS | Supervisor mode context switching. |
| 140000 | CM.U | User mode I & D separation field.<br>High bit indicates if this field is active. If active, low bit indicates the action to be taken. |
| 100000 | CM.UXX | Change User mode I & D separation |
| 100000 | CM.UII | Non-separate User I & D space. |
| 140000 | CM.UID | Separate User I & D space. |

Example:

```
 .TITLE ECMAP
;+
; This program demonstrates uses of the .CMAP request
;-
        .MCALL  .CMAP     .PRINT      .EXIT  .CRRG     .CRAW
        .MCALL  .CMPDF    .RDBBK      .WDBBK
        .CMPDF

        .PSECT  CODE,I
        .PSECT  DATA,D

        .PSECT  CODE
.ENABL  LSB
START::
        .PRINT  #IEQD                ;Start out with I=D
        CALL    A20000               ;Call the subr in PAR1
        .CMAP   #AREA,#CM.UID        ;Separate U I-D spaces
        .CRRG   #AREA,#RDB           ;Create region
        BCS     CRRERR               ;failure
        MOV     RDB+R.GID,WDB+W.NRID ;move region ID to window block
        .CRAW   #AREA,#WDB           ;create a window into it
        BCS     CRAERR               ;failure
        .PRINT  #INED                ;I not equal to D now
        CALL    A20000               ;Call the subr again
        MOV     #RETURN,A20000       ;Modify D (not I space)
        .PRINT  #THIRD               ;Tell to expect message
        CALL    A20000               ;Call the subr again
        .EXIT

CRRERR: .PRINT   #CRRMSG             ;CRRG failed
        .EXIT
```

```
CRAERR: .PRINT   #CRAMSG              ;CRAW failed
        .EXIT
.=START+20000-1000                    ;move to PAR1
A20000:
        .PRINT  #HLOPR1               ;Hello from PAR1
        RETURN
.=START+40000-1000                    ;move to PAR2
        .PSECT  DATA
AREA:   .BLKW      10.
RDB:    .RDBBK      1
WDB:    .WDBBK      1,1,0,0,0,WS.MAP!WS.D!WS.U
IEQD:   .ASCIZ     "!ECMAP-I-Running with I=D"
INED:   .ASCIZ     "!ECMAP-I-Running with I.ne.D"
THIRD:  .ASCIZ     "!ECMAP-I-We've crushed PAR1 D space"
CRRMSG: .ASCIZ     "?ECMAP-I-.CRRG failed"
CRAMSG: .ASCIZ     "?ECMAP-I-.CRAW failed"
HLOPR1: .ASCIZ     "!ECMAP-I-Hello from PAR1"
        .END    START
```

```
        .TITLE  EGCMAP
;+
; This program demonstrates the use of .GCMAP to display
; the current mapping status of a program.
;-
        .MCALL  .GCMAP  .PRINT  .EXIT   .CMPDF
        .MCALL  .GVAL   .DEBUG  .DPRINT
        .CMPDF
.LIBRARY "SRC:SYSTEM"
        .MCALL  .FIXDF  .CF3DF
        .FIXDF
        .CF3DF
.MACRO  ...
.ENDM   ...
        .PSECT  CODE,I
        .PSECT  DATA,D
.ENABL  LSB
;       set SWITCH=OFF to suppress debugging messages
        .DEBUG  SWITCH=ON,VALUE=YES,ID=YES
        .PSECT  CODE,I
.ENABL  LSB
DEBUG::
        BPT
START:
        ...                     ;setup mapping as desired
        CALL    SHOMAP          ;display the current mapping
        ...                     ;whatever
        .EXIT
```

```
SHOMAP:
        .GVAL   #AREA,#$CNFG3   ;Get 3rd config word
        MOV     R0,R1           ;save contents
        BIC     #^cCF3.SI&^cCF3.HI,R1 ;clear all but mapping bits
        BIT     #CF3.SI,R1      ;is there software support for full mapping?
        BNE     10$             ;yes
        .PRINT  #NOSOFT         ;no
10$:
        BIT     #CF3.HI,R1      ;is there hardware support for full mapping?
        BNE     20$             ;yes
        .PRINT  #NOHARD         ;no
20$:
        CMP     #CF3.SI!CF3.HI,R1 ;Is there support for both?
        BNE     90$             ;no, then GCMAP is useless
        .GCMAP  #AREA           ;Get mapping information
        MOV     R0,R1           ;save contents
        .DPRINT <GCMAP returns >,R1,OCT
        BIT     #CM.UID&^cCM.UII,R1 ;Separate U I-D?
        BEQ     30$             ;no
        .PRINT  #SEPUID         ;separated User I-D spaces
30$:
        BIT     #CM.JAS&^cCM.NOS,R1 ;Context switch Supy spaces
        BEQ     40$             ;no
        .PRINT  #SWPSPY         ;context switching Supy
40$:
        BIT     #CM.SID&^cCM.SII,R1 ;Separate S I-D?
        BEQ     50$             ;no
        .PRINT  #SEPSID         ;separated Supy I-D spaces
50$:
        TSTB    R1              ;any separated D pars?
        BEQ     90$             ;no, done
        MOVB    #'0,R2          ;par number in ascii
        MOV     #CM.PAR0,R3     ;par bit mask
        MOV     #BUFFER,R4      ;output buffer
60$:
        MOVB    #' ,@R4         ;assume locked
        BIT     R3,R1           ;unlocked?
        BEQ     70$             ;no
        MOVB    R2,@R4          ;yes, punch in number
70$:
        ASLB    R3              ;next par mask bit
        BEQ     80$             ;done
        INC     R2              ;next number
        INC     R4              ;next buffer slot
        BR      60$
80$:
        .PRINT  #SEPPAR         ;list separated pars
90$:    RETURN

.PSECT  DATA,D

AREA:   .BLKW   10.
```

```
NOSOFT: .ASCIZ  "?EGCMAP-F-Monitor does not support full mapping"
NOHARD: .ASCIZ  "?EGCMAP-F-Processor does not support full mapping"
SEPUID: .ASCIZ  "!EGCMAP-I-Separated User I and D spaces"
SWPSPY: .ASCIZ  "!EGCMAP-I-Supy enabled"
SEPSID: .ASCIZ  "!EGCMAP-I-Separated Supy I and D spaces"
SEPPAR: .ASCII  "!EGCMAP-I-Following User / Supy D pars unlocked:"
BUFFER: .BLKB   8.
        .ASCIZ  ""
        .END    START
```

# .CMKT

**EMT 375, Code 23**

The .CMKT request causes one or more outstanding mark time requests to be canceled (See the .MRKT programmed request). The .CMKT request is a SYSGEN option in the single-job monitor. Timer support is a selectable during the system generation process.

Macro Call:

**.CMKT area,id[,time]**

where:

**area** is the address of a three-word EMT argument block

**id** is a number that identifies the mark time request to be canceled. If more than one mark time request has the same *id*, the request with the earliest expiration time is canceled. If *id* = 0, all non-system mark time requests (those in the range 1 to 176777) for the issuing job are canceled

**time** is the address of a two-word area in which the monitor returns the amount of time (clock ticks) remaining in the canceled 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

Canceling a mark time request frees the associated queue element.

A mark time request can be converted into a timed wait by issuing a .CMKT followed by a .TWAIT, and by specifying the same time area.

If the mark time request to be canceled 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.

Because the *time* argument is address-checked by KMON, the macro definition always clears it to zero if it is not specified.

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | The *id* was not zero and 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:

Refer to the example for the .MRKT request.

# .CNTXSW

**Multijob**
**EMT 375, Code 33**

A context switch is an operation performed when a transition is made from running one job to running another. The .CNTXSW request is used to specify locations to be included in a list of locations saved and stored when a context switch occurs. Refer to the *RT–11 System Internals Manual* for further details.

The system always saves the parameters it needs uniquely to 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 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 before loading these locations accomplishes this. This procedure is not necessary for jobs running under the XM monitor. In the XM monitor, both IOT and BPT vectors are automatically context switched.

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. For virtual jobs, the IOT, BPT, and TRAP vectors are simulated 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-Inv SST (invalid synchronous system trap)
```

Macro Call:

**.CNTXSW area,addr**

where:

**area** is the address of a two-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 be one of the following:

- in the range 2-476
- in the user job area
- in the I/O page (addresses 160000-177776)

Request Format:

R0 → area:

| | |
|---|---|
| 33 | 0 |
| addr | |

Errors:

| Code | Explanation |
|---|---|
| 0 | One or more of the conditions specified by *addr* was violated. |

Example:

```
        .TITLE  ECNTXS.MAC
;+
; .CNTXSW - This is an example in the use of the .CNTXSW request.
; In this example, a .CNTXSW request is used to specify that location 250
; and 252 (MMU vector) and certain Memory management registers be context
; switched. This allows both jobs to use some MMU facilities simultaneously
; yet independently under a non-mapped monitor.
;-
        .MCALL  .CNTXSW,.PRINT,.EXIT

$USRRB  =:      53                      ;(.SYCDF) user error byte
 SUCCS$ =:      001                     ;(.UEBDF) success "error" bit
 FATAL$ =:      010                     ;(.UEBDF) fatal error bit

START:  .CNTXSW #AREA,#SWLIST           ;Issue the .CNTXSW request
        BCC     1$                      ;Branch if successful
        .PRINT  #ADDERR                 ;Address error (should not occur)
        BISB    #FATAL$,@#$USRRB        ;indicate error
        .EXIT                           ;Exit the program
1$:     .PRINT  #CNTOK                  ;Acknowledge success with a message
        BISB    #SUCCS$,@#$USRRB        ;indicate no error
        .EXIT                           ;then exit the program

SWLIST: .WORD   250                     ;Addresses to include in context switch
        .WORD   252                     ;MMU vector
        .WORD   172220                  ;SIPDR0
        .WORD   172240                  ;SIPAR0
        .WORD   177572                  ;MMR0
        .WORD   0                       ;List terminator !!!

AREA:   .BLKW   2                       ;EMT argument block

ADDERR: .ASCIZ  /?ECNTXS-F-.CNTXSW Addressing Error/
CNTOK:  .ASCIZ  /!ECNTXS-I-.CNTXSW Successful/

        .END    START
```

# .CRAW

**Mapped**

**EMT 375, Code 36, Subcode 2**

The .CRAW request, only available under mapped monitors, defines a virtual address window and optionally maps it into a physical memory region. Mapping occurs if you set the WS.MAP bit in the last word of the window definition block before you issue .CRAW. Since the window must start on a 4K word boundary, the program only has to specify which page address register to use and the window size in 32-word increments. If the new window overlaps previously defined windows, those windows are eliminated before the new window is created (except the static window reserved for a virtual program's base segment).

Macro Call:

**.CRAW area,addr**

where:

**area** is the address of a two-word EMT argument block

**addr** is the address of the window definition block. The .WDBBK macros generate static definitions pointed to by *addr*. See the *RT–11 System Internals Manual* for more information on mapping

The window status word (W.NSTS) of the window definition block may have one or more of the following bits set on return from the request:

- WS.CRW set if address window was successfully created
- WS.UNM set if one or more windows were unmapped to create and map this window
- WS.ELW set if one or more windows were eliminated

See program requests .WDBBK, .CRRG.

Request Format:

| | | |
|---|---|---|
| R0 → area: | 36 | 2 |
| | addr | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Window alignment error: the new window overlaps the static window for a virtual job. The window is too large or W.NAPR is greater than 7. |

1 An attempt was made to define more than seven windows in your program. Eliminate a window (.ELAW) or redefine your virtual address space into fewer windows.

If the WS.MAP bit were set in the window definition block status word, the following errors can also occur:

| Code | Explanation |
|---|---|
| 2 | An invalid region identifier was specified. |
| 4 | The combination of the offset into the region and the size of the window to be mapped into the region is invalid. |
| 17 | Inactive mode or space specified. |

Example:

```
        .TITLE  XMCOPY;2
;+
; This is an example in the use of the RT-11 Extended Memory requests.
; The program is a file copy with verify utility that uses extended
; memory to implement 4k transfer buffers. The example utilizes most of
; the Extended Memory requests and demonstrates other programming
; techniques useful in utilizing the requests.
;-
        .NLIST  BEX
        .MCALL  .UNMAP,.ELRG,.ELAW,.CRRG,.CRAW,.MAP,.PRINT,.EXIT,.CLOSE
        .MCALL  .RDBBK,.WDBBK,.TTYOUT,.WDBDF,.RDBDF,.CSIGEN,.READW,.WRITW

        .WDBDF                          ;Create Window Def Blk Symbols
        .RDBDF                          ;  "    Region  "   "    "

        $JSW    =:      44              ;(.SYCDF) JSW location
        VIRT$   =:      2000            ;(.JSWDF) Virtual Job bit in JSW
        $ERRBYT =:      52              ;(.SYCDF) Error byte location
        $USRRB  =:      53              ;(.SYCDF) User error byte
        SUCCS$  =:      001             ;(.UEBDF) Successful completion
        FATAL$  =:      010             ;(.UEBDF) Error completion
        APR     =:      2               ;PAR/PDR for 1st window
        APR1    =:      4               ;    "    "  2nd   "
        CORSIZ  =:      4096.           ;Size of buffer in words
        PAGSIZ  =:      CORSIZ/256.     ;Page size in blocks

        .ASECT                          ;Assemble in the Virt Job Bit
        .       = $JSW
        .WORD   VIRT$                   ;Make this a "virtual" job
        .PSECT                          ;Start code now

.ENABL  LSB

START:: MOV     SP,R5                   ;Save SP, .CSIGEN changes it
        .CSIGEN #ENDCRE,#DEFLT,#0       ;Get filespecs, handlers, open files
        MOV     R5,SP                   ;*C* Restore it (preserve carry)
        BCS     START                   ;Branch if error
        INCB    ERRNO                   ;ERR = 1x
        .CRRG   #CAREA,#RDB             ;Create a region
        BCC     10$                     ;Branch if successful
        JMP     ERROR                   ;Report error (JMP due to range!)
10$:    MOV     RDB,WRNID               ;Move region id to Window Def Blk
        INCB    ERRNO                   ;ERR = 2x
        .CRAW   #CAREA,#WDB             ;Create window...
        BCC     20$                     ;Branch if no error
```

```
        JMP     ERROR                   ;Report error...
20$:    INCB    ERRNO                   ;ERR = 3x
        .MAP    #CAREA,#WDB             ;Explicitly map window...
        BCC     30$                     ;Branch if no error
        JMP     ERROR                   ;Report error
30$:    CLR     R1                      ;R1 = RT11 Block # for I/O
        MOV     #CORSIZ,R2              ;R2 = # of words to read
        INCB    ERRNO                   ;ERR = 4x
READ:   .READW  #RAREA,#3,BUF,R2,R1     ;Try to read 4k worth of blocks
        BCC     WRITE                   ;Branch if no error
        TSTB    @#$ERRBYT               ;EOF?
        BEQ     PASS2                   ;Branch if yes
        JMP     ERROR                   ;Must be hard error, report it
WRITE:  MOV     R0,R2                   ;R2 = size of buffer just read
        .WRITW  #RAREA,#0,BUF,R2,R1     ;Write out the buffer
        BCC     ADDIT                   ;Branch if no error
        INCB    ERRNO                   ;ERR = 5x
        JMP     ERROR                   ;Report error
ADDIT:  ADD     #PAGSIZ,R1              ;Adjust block #
        BR      READ                    ;Then go get another buffer
PASS2:  INCB    ERRNO                   ;ERR = 6x
        .CRRG   #CAREA,#RDB1            ;Create a region
        BCC     40$                     ;Branch if no error
        JMP     ERROR                   ;Report error
40$:    MOV     RDB1,WRNID1             ;Get region id to window def blk

;* EXAMPLE USING THE .CRAW REQUEST DOING *
;* IMPLIED .MAP REQUEST.                 *

        INCB    ERRNO                   ;ERR = 7x
        .CRAW   #CAREA,#WDB1            ;Create window using implied .MAP
        BCC     VERIFY                  ;Branch if no error
        JMP     ERROR                   ;Report error
VERIFY::INCB    ERRNO                   ;ERR = 8x
        CLR     R1                      ;R1 = RT11 block # again
GETBLK: MOV     #CORSIZ,R2              ;R2 = 4k buffer size
        .READW  #RAREA,#3,BUF1,R2,R1    ;Try to get 4K worth of input file
        BCC     50$                     ;Branch if no error
        TSTB    @#$ERRBYT               ;EOF?
        BEQ     ENDIT                   ;Branch if yes
        JMP     ERROR                   ;Report hard error
50$:    MOV     R0,R2                   ;R2 = size of buffer read
        .READW  #RAREA,#0,BUF,R2,R1     ;Try to get same size from output file
        BCC     60$                     ;Branch if no error
        INCB    ERRNO                   ;ERR = 9x
        JMP     ERROR                   ;Report error
60$:    MOV     BUF,R4                  ;Get output buffer address
        MOV     BUF1,R3                 ;Get input buffer address
70$:    CMP     (R4)+,(R3)+             ;Verify that data is the same
        BNE     ERRDAT                  ;It's not, report error
        DEC     R2                      ;Are we finished?
        BNE     70$                     ;Branch if we aren't
        ADD     #PAGSIZ,R1              ;Adjust block # for page size
        BR      GETBLK                  ;Go get another buffer pair

ENDIT:  .PRINT  #ENDPRG                 ;Announce we're finished
        BISB    #SUCCS$,@#$USRRB        ;Indicate no error
XCLOS:  .CLOSE  #0                      ;Close output file
        .UNMAP  #CAREA,#WDB             ;Explicitly unmap 1st window
        .ELAW   #CAREA,#WDB             ;Explicitly eliminate 1st window
        .ELRG   #CAREA,#RDB             ;Eliminate 1st region
        .ELRG   #CAREA,#RDB1            ;Unmap,eliminate 2nd window & region
        .EXIT                           ;Exit program
```

```
ERROR:  MOVB    #'0,ERRNO2              ;Setup first digit
        MOVB    @#$ERRBYT,R0            ;Get error byte code
        CMPB    R0,#10                  ;If larger than 10(8)
        BLT     80$                     ;NOT
        INCB    ERRNO2                  ;make 0 into 1
        SUB     #10,R0                  ; and reduce 2nd "digit" by 10
80$:    ADD     #'0,R0                  ;of error code...
        MOVB    R0,ERRNO3               ;Put it in error message
        .PRINT  #ERR                    ;Print it...
        BR      ECLOS                   ;Go close output file
ERRDAT: .PRINT  #ERRBUF                 ;Report verify failed...
ECLOS:
        BISB    #FATAL$,@#$USRRB        ;Indicate error
        BR      XCLOS                   ;Go close output file

RDB:    .RDBBK  CORSIZ/32.              ;.RDDBK defines Region Def Blk
WDB:    .WDBBK  APR,CORSIZ/32.          ;.WDDBK defines Window Def Blk
RDB1:   .RDBBK  CORSIZ/32.              ;Define 2nd region same way
WDB1:   .WDBBK  APR1,CORSIZ/32.,0,0,CORSIZ/32.,WS.MAP ; and 2nd Window
                                        ;(but with mapping status set!)

        BUF     =:      WDB+W.NBAS      ;Virtual addr of 1st buffer
        BUF1    =:      WDB1+W.NBAS     ;   "    "    "  2nd   "
        WRNID   =:      WDB+W.NRID      ;Region ID addr of 1st region
        WRNID1  =:      WDB1+W.NRID     ;   "   "   "    " 2nd    "

CAREA:  .BLKW   2                       ;EMT argument blocks
RAREA:  .BLKW   6
DEFLT:  .WORD   0,0,0,0                 ;No default extensions
ENDPRG: .ASCIZ  /!XMCOPY-I-End of XM Example Program/
ERR:    .ASCII  /?XMCOPY-F-Request or I-O Error # /
ERRNO:  .ASCII  /0/
        .ASCII  /, CODE=/
ERRNO2: .ASCII  /0/
ERRNO3: .ASCIZ  /0/

ERRBUF: .ASCIZ  /?XMCOPY-F-Data Verification Error/
ENDCRE  =.                              ;For CSIGEN

        .END    START
```

# .CRRG

**Mapped**

**EMT 375, Code 36, Subcode 0**

The .CRRG request directs the monitor to allocate a dynamic region in physical memory for use by the current requesting program.

Macro Call:

**.CRRG area[,addr]**

where:

| | |
|---|---|
| **area** | is the address of a two-word EMT argument block |
| **addr** | is the address of the region definition block for the region to be created or attached by RT–11. The .RDBBK macro can be used to initialize the region definition block. For more information on mapping, see the *RT–11 System Internals Manual*. |

Request Format:

R0 → area:

| | |
|---|---|
| 36 | 0 |
| addr | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 6 | No region control blocks are available. You eliminate a region to obtain a region control block (.ELRG), or you can redefine your physical address space into fewer regions. |
| 7 | A region of the requested size cannot be created because not enough memory is available. The size of the largest available region is returned in R0. |
| 10 | An invalid region size was specified. A value of 0, or a value greater than the available amount of contiguous extended memory, is invalid. |
| 12 | Global region not found. |
| 13 | Too many global regions in use (none free). |
| 15 | Global region is privately owned. |
| 16 | Global region already exists at a different base address. |

Example:

Refer to example for the .CRAW request.

# .CSIGEN

**EMT 344**

The .CSIGEN request calls the Command String Interpreter (CSI) in general mode to process a standard RT–11 command string. In general mode, file .LOOKUP and .ENTER requests as well as handler .FETCH requests are performed.

**NOTE**

This request returns information on the stack.

When .CSIGEN accepts the command string:

**dev:output-filespec[size]=dev:input-filespec/options**

the following operations occur:

1. The handlers for 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. See the end of this section for a description of the way option information is passed. Note that this call always puts at least one word of information on the stack.

.CSIGEN purges channels 0 through $10_8$ before processing the command string. If errors occur during processing, it purges them again.

.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, your program can be destroyed. (The system, however, is protected.)

The three possible output files are assigned to channels 0, 1, and 2, and the six possible input files are assigned to channels 3 through $10_8$. A null specification causes the associated channel to remain inactive. For example, the string:

***,NL:=F1,F2**

causes:

- Channel 0 to be inactive since the first specification is null.
- Channel 1 to be associated with the null handler device
- Channel 2 to be inactive.
- Channels 3 and 4 to be associated with two files on DK:
- Channels 5 through 10 to be inactive.

Your 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.

## .CSIGEN

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 an explicit file type.

**cstrng** is the address of the ASCIZ command string or a 0 if input is to come from the console terminal. (In a multijob environment, if the input is from the console terminal, an .UNLOCK of the USR is automatically performed while the string is being read, even if the USR is locked at the time.) If the string is in memory, it must not contain a RETURN (octal 15 and 12), and must terminate with a zero byte. If the *cstrng* field is 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 storage address of the original command string. This is a user-supplied area, 81 decimal bytes in length. The command string is terminated with a zero byte. If this argument is omitted, the input command string is not copied to user memory.

On return, R0 points to the first available location above the handlers, the stack contains the option information, and all the specified files have been opened.

The four-word block pointed to by *defext* is arranged as:

**Word 1** Default file type for all input channels (3-10)

**Words 2,3,4** Default file types for output channels 0, 1, and 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 default extension block sets up default file types for a macro assembler:

```
        .TITLE  ECSIG1.MAC
DEFEXT:
        .RAD50  "MAC"   ;Input files default type
        .RAD50  "OBJ"   ;First output file default type
        .RAD50  "LST"   ;Second output file default type
        .WORD   0       ;Third output file default type (none)
```

In the command string:

***DU0:ALPHA,DU1:BETA=DU2:INPUT**

the default file type for input is MAC; for output, OBJ and LST. The following cases are valid:

***DU0:OUTPUT=**

***DU2:INPUT**

In other words, the equal sign is required after all output files but is not necessary if only input files are specified.

An optional argument *linbuf* is available in the .CSIGEN format that provides an area to receive the original input string. The input string, returned as an ASCIZ string, can be printed through a .PRINT request.

The .CSIGEN request automatically takes its input line from an indirect command file if console terminal input is specified *(cstrng = #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. In either case, the options and option-count are purged from the stack. These errors are:

| **Code** | **Explanation** |
|---|---|
| 0 | Invalid command (such as bad separators, invalid file names, and commands that are too long). |
| 1 | A device specified is not found in the system tables. |
| 2 | A protected file of the same name already exists. A new file was not opened. |
| 3 | Device full. |
| 4 | An input file was not found in a .LOOKUP. |

Example:

```
        .TITLE  ECSIGE;2
;+
; .CSIGEN - This is an example in the use of the .CSIGEN request.
; The example is a single file copy program. The file specs are
; input from the console terminal, and the input & output files opened
; via the general mode of the CSI. The file is copied using synchronous
; I/O, and the output file is made permanent via the .CLOSE request.
;-
        .MCALL  .CSIGEN,.READW,.EXIT,.WRITW,.CLOSE,.SRESET
        .MCALL  .PRINT

$ERRBYT =:      52                      ;(.SYCDF) Error Byte
$USRRB  =:      53                      ;(.SYCDF) User Error Byte
FATAL$  =:      010                     ;(.UEBDF) error indication
```

```
START:  MOV     SP,R5                      ;Save SP since .CSIGEN changes it
        .CSIGEN #DSPACE,#DEXT              ;Get string from terminal
        MOV     R5,SP                      ;Restore SP
        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 a block on Channel 3
        BCC     2$                         ;Branch if no errors
        TSTB    @#$ERRBYT                  ;EOF error ?
        BEQ     EOF                        ;Yes...
        MOV     #INERR,R0                  ;R0 => Read Error Message
1$:     .PRINT                             ;Print the message
        BISB    #FATAL$,@#$USRRB           ;Indicate error
        CLR     R0                         ;Clear R0 for hard exit
        .EXIT                              ;Exit the program

2$:     .WRITW  R5,#0,BUFF,#256.,INBLK     ;Write the block just read
        BCC     NOERR                      ;Branch if no error
        MOV     #WTERR,R0                  ;R0 => Write error message
        BR      1$                         ;Branch to output the message
NOERR:  INC     INBLK                      ;Otherwise, increment block #
        BR      READ                       ;and loop to read next block
EOF:    .CLOSE  #0                         ;End-of-File...CLose output channel
        .CLOSE  #3                         ;And input channel
        .EXIT                              ;Exit the program

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  /?ECSIGE-F-Input error/
WTERR:  .ASCIZ  /?ECSIGE-F-Output error/
        .EVEN
DSPACE:     ;area for handlers

        .END    START
```

#### Passing Option Information

Both .CSIGEN and .CSISPC parse options and their associated values in reverse order from that specified on the command line. That is, the last option and associated value (if present) placed last on the stack will be the first option retrieved.

In both general and special modes of the CSI, options and their associated values are returned on the stack. A CSI option is introduced by slash (/) followed by any character. The CSI does not restrict the option to display characters, although you should use printing characters to avoid confusion. The option can be followed by a value, which is indicated by a *: separator*. The : separator is followed by 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; for example, the following command string has two A options:

***DK:FOO/A,DU4:FILE.OBJ/A:100**

The first is associated with the input file DK:FOO. The second is associated with the input file DU4:FILE.OBJ and has a value of $100_8$. The format of the stack output of the CSI for options is as follows:

| Word | Value | Meaning |
|---|---|---|
| 1 | N | Number of options found in command string. If N=0, no options were found. |
| 2 | Option character and file number | Even byte = seven-bit ASCII option character |
| | | 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 character and file number, if any. |

For example, if the input line to the CSI is

*FILE/B:20.,FIL2/E=DU3:INPUT/X:SY:20

on return, the stack is:

```
                .TITLE  ECSIG2.MAC
;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   ;Value of option X = RAD50 for SY
                        505   ;Next option = E; with file 1; no value
                     100102   ;Next option = B; with file 0; has a value
                         24   ;Value of option B = 20(decimal)
```

Keyboard error messages that can occur when input is from the console keyboard include:

| Message | Meaning |
|---|---|
| *?CSI-F-Invalid command* | Syntax error. |
| *?CSI-F-File not found* | Input file was not found. |
| *?CSI-F-Device full* | Output file does not fit. |
| *?CSI-F-Invalid device* | Device specified does not exist. |
| *?CSI-F-Protected file* | Specified output file already exists and is protected. |

#### Notes

- In many cases, your program does not need to process options in CSI calls. However, because you could inadvertently enter options at the console you should save the value of the stack pointer before the call to the CSI, and restore it after the call, so that no extraneous values are left on the stack. Note that even a command string with no options causes a zero word to be pushed onto the stack. This word indicates the number of options to follow.
- Under a multijob monitor, calls to the CSI that require console terminal input always do an implicit .UNLOCK of the USR while the string is being gathered. This should be kept in mind when using .LOCK calls.

## .CSISPC

**EMT 345**

The .CSISPC request calls the Command String Interpreter to parse the command string and return file descriptors and options to the program. The CSI does not perform any .CLOSE, .ENTER, .LOOKUP, or handler .FETCH requests.

Options and their associated values are returned on the stack. The optional argument *linbuf* can provide your 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.

Note that in a multijob environment, calling the CSI performs a temporary and implicit .UNLOCK while the command line is being read.

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 *cstrng*, 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 RETURN (octal 15 and 12), and must terminate with a zero byte. If *cstrng* is blank, input is automatically taken from the console terminal or indirect file, if one is active

**linbuf** is the storage address of the original command string. This is a user-specified area, 81 bytes in length. The command string is terminated with a zero byte instead of RETURN (octal 15 and 12)

**Notes**

- The file description consists of 39 words, comprising 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 zeroes.
- The five-word blocks hold four words of Radix–50 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 ([ ]) that follows the output file descriptor.) For example:

*DU3:LIST.MAC[15]=TT:

Using special mode, the CSI returns in the first five-word slot:

```
        .TITLE  ECSIP1.MAC

         16151  ;RAD50 "DU3"
         46173  ;RAD50 "LIS"
         76400  ;RAD50 "T  "
         50553  ;RAD50 "MAC"
            17  ;WORD  15. (DECIMAL)
```

In the fourth slot (starting at an offset of 36 bytes [octal] into outspc), the CSI returns:

```
        .TITLE  ECSIP2.MAC

        100040  ;RAD50 "TT "
             0  ;no file name
             0  ;    "
             0  ;no file type
```

Since this is an input file, only four words are returned.

As an extended example, assume the following string was input for the CSI in general mode:

*FILE[8],LP:,SY:FILE2[20]=LD:,DU1:IN1/B,DU2:IN2.MLB/M:7

Assume also that the default file type block is:

```
        .TITLE  ECSIP3.MAC
DEFEXT:
        .RAD50  "MAC"   ;Input file type
        .RAD50  "OP1"   ;First output file type
        .RAD50  "OP2"   ;Second output file type
        .RAD50  "OP3"   ;Third output file type
```

This default extension block sets up default file types for a macro assembler, where:

- The first word is the default input file type.
- The second word is the first output file type.
- The third word is the second output file type.
- The fourth word is the third output file type.

The results of the above CSI call are as follows:

- 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 20-block file named FILE2.OP3 is entered on the system device on channel 2.
- Channel 3 is open for input from device LD:; channel 4 is open for input from a file IN1.MAC on device DU1:; channel 5 is open for input from IN2.MLB on device DU2:.
- The stack contains options and values as follows:

```
                .TITLE  ECSIP4.MAC
;Stack Pointer  ->
                   2  ;Two options found in string
              102515  ;Second option = M; with file 5; has a value
                   7  ;Value is 7(octal)
                2102  ;First option = B; with file 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:

```
        .TITLE  ECSIP5.MAC
OUTSPC:         15270  ;RAD50 "DK "
                23364  ;RAD50 "FIL"
                17500  ;RAD50 "E  "
                60137  ;RAD50 "OP1"
                   10  ;WORD   8. (decimal)
                46600  ;RAD50 "LP "
                    0  ;No name or size specified
                    0
                    0
                    0
                75250  ;RAD50 "SY "
                23364  ;RAD50 "FIL"
                22100  ;RAD50 "E2 "
                60141  ;RAD50 "OP3"
                   24  ;WORD  20. (decimal)
                45640  ;RAD50 "LD "
                    0  ;No name specified
                    0
                    0
                16147  ;RAD50 "DU1"
                35217  ;RAD50 "IN1"
                    0  ;RAD50 "   "
                50553  ;RAD50 "MAC"
                16150  ;RAD50 "DU2"
                35220  ;RAD50 "IN2"
                    0  ;RAD50 "   "
                51560  ;RAD50 "MLB"
                    0  ;12 words of zeros
                  ...
                    0
```

If you want to use default extensions, but you need to base the default on an option (for instance a .MAC without an option or a .MLB, if /M present), you can use the following trick: Define the default extension as *.word -1* which is an invalid RAD–50 value. Then, if the user specifies an extension, it appears as valid RAD50 in OUTSPC. However, if no extension is specified, -1 appears in OUTSPC and the program can, after consulting the options, substitute the required one.

Errors:

| Code | Explanation |
|---|---|
| 0 | Invalid command line. |
| 1 | Invalid device. |

These are the same errors as .CSIGEN returns, except that invalid device specifications are checked only for output file specifications with null file names.

Example:

```
        .TITLE  ECSISP;1
;+
; .CSISPC - This is an example in the use of the .CSISPC request.
; The example uses the "special" mode of CSI to get an input
; specification from the console terminal, then uses the .DSTATUS
; request to determine if the input device's handler is loaded;
; if not, a .FETCH request is issued to load the handler into
; memory. Finally a .DELETE request is issued to delete the specified
; file.
;-
;+
;       To use enter a single file name at the * prompt
;-
        .MCALL  .DSTATUS,.PRINT,.EXIT,.FETCH,.CSISPC,.DELETE

DS.ADR  =:      4                   ;(.DSTDF) address word in DSTAT return

$USRRB  =:      53                  ;(.SYCDF) user error byte
SUCCS$  =:      001                 ;(.UEBDF) success
ERROR$  =:      004                 ;(.UEBDF) error
FATAL$  =:      010                 ;(.UEBDF) fatal

CS.IN1  =:      36                  ;(.CSIDF) offset to first input dblk

START:  MOV SP, R5                  ;Save current stack pointer
        .CSISPC #OUTSP,#DEFEXT      ;Use .CSISPC to get output spec
        MOV R5, SP                  ;Restore SP to clear any CSI options
        .DSTAT  #STAT,#INSPEC       ;Check on the input device
                                    ;(CSISPC catches illegal devices!)
        TST     STAT+DS.ADR         ;See if the device is resident
        BNE     2$                  ;Branch if already loaded
        .FETCH  #HANLOD,#INSPEC     ;It's not loaded...bring it into memory
        BCC     2$                  ;Branch if successful
        .PRINT  #FEFAIL             ;FETCH failed...print error message
        BISB    #FATAL$,@#$USRRB    ;indicate a fatal error
        .EXIT                       ;then exit program
2$:     .DELETE #AREA,#0,#INSPEC    ;Now delete the file
        BCC     3$                  ;Branch if successful
        .PRINT  #NOFIL              ;Print error message
        BISB    #ERROR$,@#$USRRB    ;indicate an error
        BR      START               ;Then try again
3$:     .PRINT  #FILDEL             ;Acknowledge successful deletion
        BISB    #SUCCS$,@#$USRRB    ;indicate success
        .EXIT                       ;then exit program
AREA:   .BLKW   2                   ;EMT Argument block
.BLKW   4                           ;Block for status
DEFEXT: .WORD   0,0,0,0             ;No default extensions
FEFAIL: .ASCIZ  /?ECSIGE-F-.FETCH Failed/ ;Fetch failed message
NOFIL:  .ASCIZ  /?ECSIGE-E-File Not Found/ ;File not found
FILDEL: .ASCIZ  /!ECSIGE-I-File Deleted/ ;Delete acknowledgment
        .EVEN                       ;Fix boundary

OUTSP:  .BLKW   39.                 ;Output specs go here
INSPEC  =:      OUTSP+CS.IN1        ;Input specs go here
HANLOD: .BLKW 1                     ;Handlers begin loading here (if
                                    ;necessary)
        .END    START
```

# .CSTAT

**EMT 375, Code 27**

This request furnishes you with information about a channel.

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 consist of the following information:

- Channel status word (See the *RT–11 System Internals Manual* and the *RT–11 Device Handlers Manual* for details)
- Starting block number of file (0 if sequential-access device, or if channel was opened with a non-file-structured .LOOKUP or .ENTER)
- Length of file (0 if non-file-structured device, or if channel was opened with a non-file-structured .LOOKUP or .ENTER)
- Highest relative block written since file was opened (no information if non-file-structured device). This word is maintained by the .WRITE/.WRITC/.WRITW requests
- Unit number of device with which this channel is associated

  .CSTAT supports extended device unit handlers by returning the one-letter device name found in the $PNAM2 table, if the specified device unit is higher than 7. If the specified device unit is in the 0-7 range, .CSTAT continues to return the 2-letter device name found in the $PNAME table.
- Radix–50 of the device name with which the channel is associated (Note: This is a physical device name, unaffected by the use of a logical name when the channel was open.)

Errors:

| Code | Explanation |
|---|---|
| 0 | The channel is not open. |

Example:

```
        .TITLE  ECSTAT;2
;+
; .CSTAT - This is an example in the use of the .CSTAT request.
; In this example, .CSTAT is used to determine the .RAD50
; representation of the device with which the channel is associated.
; It also displays the starting block (in octal) and the length (in decimal).
;-
;       to use, supply 1 input file name
;-
        .MCALL  .CSTAT,.CSIGEN,.PRINT,.EXIT
        .MCALL  .DEBUG  .DPRINT
.ENABL  LSB
        .DEBUG  SWITCH=ON,VALUE=YES
.DSABL  LSB

CS.SBK  =:      02                      ;(.CSTDF) starting block returned
CS.LEN  =:      04                      ;(.CSTDF) length of file returned
CS.UNT  =:      10                      ;(.CSTDF) unit number returned
CS.NAM  =:      12                      ;(.CSTDF) device name returned

$USRRB  =:      53                      ;(.SYCDF) User Error Byte
SUCCS$  =:      001                     ;(.UEBDF) success indication
FATAL$  =:      010                     ;(.UEBDF) error indication
.ENABL  LSB
START:  MOV     SP, R5                  ;Save current stack pointer
        .CSIGEN #DEVSDC,#DEFEXT         ;Open files
        MOV     R5, SP                  ;Restore SP to clear any CSI options
        .CSTAT  #AREA,#3,#ADDR          ;Get the status
        BCS     NOCHAN                  ;Channel 3 not open
        MOV     #ADDR+CS.UNT,R5         ;Point to unit #
        MOV     (R5)+,R0                ;Unit # to R0
        ADD     #^r  0,R0               ;Make it RAD50
        CMP     #^r  7,R0               ;Was it 0--7?
        BGE     10$                     ;Yes, xxn form
        ADD     #^r 0 ,R0               ;Else must have been 10--77 (xnn form)
10$:    ADD     (R5),R0                 ;Get device name
        MOV     R0,DEVNAM               ;'DEVNAM' has RAD50 device name
        .DPRINT ^"!ECSTAT-I-First block - ",ADDR+CS.SBK ;Display first block
        .DPRINT ^"!ECSTAT-I-File length - ",ADDR+CS.LEN,DEC ;And size
        BISB    #SUCCS$,@#$USRRB        ;Indicate success
        .EXIT                           ;Exit the program
NOCHAN: .PRINT  #MSG                    ;Print error message
        BISB    #FATAL$,@#$USRRB        ;Indicate success
        .EXIT                           ;then exit program
.DSABL  LSB
MSG:    .ASCIZ  /?ECSTAT-F-No Input File/ ;Error message
        .EVEN                           ;Fix boundary
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                 ;No default extensions
DEVSDC: .BLKW   39.                     ;Start CSI tables here...
        .END    START
```

# .CTIMIO

**Timeout**

**Macro Expansion**

The .CTIMIO macro cancels the device time-out request in the handler interrupt service section when an interrupt occurs to disable the completion routine (See .TIMIO). The device time-out feature is only useable if it was selected during the system generation process.

If the time interval has already elapsed and the device has timed out, the .CTIMIO request fails and the completion routine has already been placed in the queue. The .CTIMIO call returns with the C bit set when it fails because the completion routine has already been queued.

Macro Call:

**.CTIMIO tbk**

where:

**tbk** is the address of the seven-word timer block shown in Table 2–2.

**Table 2–2: Timer Block Format**

| Offset | Filled in by | Contents |
|---|---|---|
| 0 | .TIMIO | High-order time word (expressed in ticks). |
| 2 | .TIMIO | Low-order time word (expressed in ticks). |
| 4 | Monitor | Link to next queue element; 0 indicates none. |
| 6 | Handler | Owner's job number; 0 for background job, MAXJOB for foreground job, and job priority *2 for system jobs. MAXJOB is equal to (the number of jobs in the system * 2)-2. The job number for the foreground job is 2 in a system without system jobs, and 16 for a system with system jobs. The job number is set from the queue element. |
| 10 | Handler | Sequence number of timer request. Use the xx$COD, plus the 177000. The valid range of sequence numbers is from 177000 to 177377. |
| 12 | Monitor | -1 |
| 14 | Handler | Address of the completion routine to execute if timeout occurs. The monitor zeroes this word when it calls the completion routine, indicating that the timer block is available for reuse. |

The .CTIMIO macro expands as follows:

```
.TITLE  ECTIMI

.CTIMIO tbk

JSR     R5,@$TIMIT
.WORD   tbk-.
.WORD   1
```

Errors:
None.

Example:
Refer to the example for the .TIMIO request.

# .DATE

**EMT 374, Code 12**

This request returns in R0 the current date information from the system date word. The date word returned is in the following format:

```
Bit:    15 14   13 ... 10   9 ... 5   4 ... 0
        \_____/\_________/\_______/\_______/
          Age     Month      Day      Year
```

where:

The *year* value in bits 4 to 0 is the actual year minus 1972. The *day* in bits 9-5 is a number from 1 to the length of the month. The *month* in bits 13 to 10 is a number from 1 to 12. *Age* in bits 14 and 15 is a number from 0 to 3. The age value multiplied by $32_{10}$ should be added to 1972 and to the year value in bits 4 through 0.

**NOTE**

RT–11 support of month- and year-rollover is a system generation option; if not selected, the keyboard monitor DATE command must be issued to change the month and year.

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:

```
        .TITLE  DATE.MAC

;+
; .DATE - This is an example in the use of the .DATE request.
; This example displays the date numerically
;
; INPUT:        none
;
; OUTPUT:       Day, Month, Year in Decimal
;
;
;
;ERRORS:        0 if no date entered
;-
```

```
        .MCALL  .DATE   .DEBUG  .DPRINT .EXIT
.ENABL  LSB
        .DEBUG  SWITCH=ON,VALUE=YES
.DSABL  LSB

$USRRB  =:      53                      ;(.SYCDF) User error byte
SUCCS$  =:      001                     ;(.UEBDF) Success code
ERROR$  =:      004                     ;(.UEBDF) Error code

.ENABL  LSB             ;54321098 76543210
DATE::  .DATE           ;AAMMMMDD DDDYYYYY Get date in R0 via .DATE request
        MOV     R0,R2                   ;Copy R0
        BEQ     1$                      ;If zero, no date was entered
        BIC     #^C37,R2 ;00000000 000YYYYY Clear all but year bits
        MOV     R0,R1    ;AAMMMMDD DDDYYYYY Copy R0 again
        SWAB    R1       ;DDDYYYYY AAMMMMDD Move age bits to low byte
        ASR     R1       ;?DDDYYYY YAAMMMMM And move them to 32.* position
        BIC     #^C140,R1;00000000 0AA00000 Save only age bits
        ADD     R1,R2                   ;add into year
        ADD     #1972.,R2               ;Make it current year
        MOV     R0,R1    ;AAMMMMDD DDDYYYYY Copy date word again
        ASL     R1       ;AMMMMDDD DDYYYYY0 Get day bits
        ASL     R1       ;MMMMDDDD DYYYYY00 on a byte boundary...
        ASL     R1       ;MMMDDDDD YYYYY000
        SWAB    R1       ;YYYYY000 MMMDDDDD Put day bits in low order byte
        BIC     #^C37,R1 ;00000000 000DDDDD Clear all but day bits
        SWAB    R0       ;DDDYYYYY AAMMMMDD Put month bits in low byte
        ASR     R0       ;?DDDYYYY YAAMMMMD Right adjust
        ASR     R0       ;??DDDYYY YYAAMMMM month bits...
        BIC     #^C17,R0 ;00000000 0000MMMM Clear all but month bits
        .DPRINT ^"!EDATE-I-Day   - ",R1,DEC
        .DPRINT ^"!EDATE-I-Month - ",R0,DEC
        .DPRINT ^"!EDATE-I-Year  - ",R2,DEC
        BISB    #SUCCS$,@#$USRRB        ;Indicate success
        .EXIT
1$:     .DPRINT ^"?EDATE-I-No date set"
        BISB    #ERROR$,@#$USRRB        ;Indicate minor error
        .EXIT

        .END    DATE
```

# .DEBUG/.DPRINT

Use these run-time debug message macros to insert debugging messages into programs. .DEBUG and .DPRINT enable simple printing of strings and optional printing of 16-bit octal or decimal values. Before .DEBUG and .DPRINT are issued, .ENABL LSB needs to be in effect. Alternatively, you can define local label arguments in each call, but this is not recommended.

### .DEBUG

.DEBUG sets up the environment for the .DPRINT macro and may generate routines to support octal/decimal displays used by .DPRINT. The symbol ...V23 controls the generation of .DPRINT macros:

- If ...V23 = 0, then .DPRINT macros will generate no code.
- If ...V23 = 177777, then all .DPRINT macros will be generated.

Other values for ...V23 may be used to select one or more classes of .DPRINT macros to generate. This macro also defines the .DPSec macro used by the .DPRINT macro to define the PSECT for the data strings.

Macro Call:

**.DEBUG switch,class,pic,id,value,psect,code,?L1,?L2,?L3,?L4**

where:

| | | |
|---|---|---|
| **switch** | (Default) | V23 to 0 |
| | ON/on | Set V23 to value specified for CLASS |
| | OFF/off | Set V23 to 0 |
| **class** | 177777 | (Default) Select all classes |
| | xxxxxx | Bit mask to select classes |
| **pic** | Default | Do not generate PIC code in .DPRINT macros |
| | YES | Generate PIC code in .DPRINT macros |
| **id** | Default | Do not generate code for separated I-D modes |
| | YES | Generate code for separated I-D modes |
| **value** | Default | Do not generate support subroutines for value printing |
| | YES | Generate support subroutines for value printing |
| **psect** | (Deb$ug) | PSECT used for text string generated by .DPRINT |
| **code** | (Deb.ug) | PSECT used for support subroutines |
| **L1–L4** | (xxxxx$) | Local labels used for support routines |

For example,

```
        .TITLE  EDEBUG.MAC

;+
; This example shows some uses of the .DEBUG and .DPRINT
; to debug a program
;-

.MCALL  .DEBUG  .DPRINT .EXIT

.ENABL  LSB
        .DEBUG  SWITCH=ON,VALUE=YES     ;enable all classes by default
.DSABL  LSB                             ;generate the octal and decimal
                                        ;display routines
.MACRO  ...
.ENDM

$ERRBY  =:      52                      ;(.SYCDF) error byte

.ENABL  LSB
MOWAT:
        .DPRINT ^"Entering MOWAT routine"
        ...
        SEC                             ;simulate error
        MOVB    #17,@#$ERRBY            ;...

        BCC     10$                     ;test for error
        MOVB    @#$ERRBY,R0             ;get error byte
        .DPRINT ^"Unexpected error - ",R0 ;display error "byte"
10$:

        .DEBUG  SWITCH=ON,CLASS=000001  ;display only class 1 .DPRINTs
        .DPRINT ^"This should not print",CLASS=2
        .DPRINT ^"This should print",CLASS=1
        ...

        .DEBUG  SWITCH=ON,CLASS=177777  ;display all classes of .DPRINTs
        MOV     #12345.,R0              ;load value for FARLEY
FARLEY:
        .DPRINT ^"On entry to FARLEY, R0 is - ",R0,DEC
        ...
        .EXIT

        .END    MOWAT
```

### .DPRINT

This macro conditionally generates code to print a string, thereby simplifying program debugging. The *class* arguments of .DEBUG and .DPRINT can be used to partition .DEBUG output into as many as 16 classes.

Macro Call:

**.DPRINT string,value,type,class,?L1**

where:

**string** String to print, enclosed in <>
.ASCII is generated with " " as delimiters

**value** Value to print if non-blank
Use R0 to print value in R0
Avoid stack references.

Note: .DPRINT issues a .PRINT that destroys (clears) location 52, the error byte, before .DPRINT picks up any value to display. If you want to display contents of the error byte, you must move it to a temporary location, and reference that location in .DPRINT.

**type** (OCT) Display value in octal format
(DEC) Display value in decimal format

**class** (177777) Generate code if any class enabled
xxxxxx Generate code if ...V23 and class is non-zero

**L1** (xxxxx$) Local symbol. If .ENABLE LSB is not in effect, you must provide non-local user symbol for L1.

# .DELETE

**EMT 375, Code 0**

The .DELETE request deletes a specified file from a specified device. This request is supported for distributed handlers that support direct access devices. .DELETE is invalid for magtapes; however, a special directory user written handler could support .DELETE.

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-$376_8$

**dblk** is the address of a four-word Radix–50 descriptor of the file to be deleted

**seqnum** is a file position number (Not supported or used by any handler supplied by Digital)

Request Format:

| | | |
|---|---|---|
| R0 → area: | 0 | chan |
| | addr | |
| | seqnum | |

**Notes**

The channel specified in the .DELETE request must be available when the request is made or an error will occur. For RT–11 file structured devices, the file is deleted from the device, and an empty 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 used be in memory when the request is made. When .DELETE is complete, the specified channel is free for reuse.

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel is not available. |
| 1 | File was not found in the device directory. |
| 2 | Invalid operation. |
| 3 | The file is protected and cannot be deleted. |

Example:
See the example for .CSISPC.

# .DEVICE

**EMT 375, Code 14, Subcodes 0, 1**

This request enables your program to load device registers with any necessary address values when the program is terminated. You set up the list of addresses with the specified values.

This request provides this list of *address-value* pairs to the system whenever your program terminates normally or abnormally. When you issue an .EXIT request or a CTRL/C from the terminal, the system loads these designated addresses with the corresponding values. In this way your program can turn off a device's interrupt enable bit whenever the program servicing the device terminates.

When you need to link requested tables, successive calls to .DEVICE are allowed. When a program terminates and the monitor has processed the device list, the monitor disables the feature until another .DEVICE call is executed. Therefore, reenterable background programs should include .DEVICE as a part of the reenter code.

The .DEVICE request is ignored when it is issued by a virtual job.

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 element of a list is composed of a one-word address and a one-word value to be put at that address. If *addr* is #0, any previous list is discarded; in this form, the argument *link* must be omitted.

**link** is an optional argument that, if present, specifies 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:

Linked

| R0 → area: | 14 | 0 |
|---|---|---|
| | addr | |

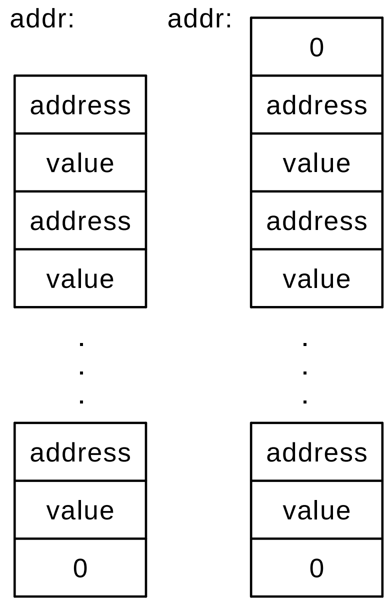
Unlinked

| R0 → area: | 14 | 1 |
|---|---|---|
| | addr | |

**NOTE**

The list referenced by *addr* must be in either linking 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.

**Nonlinking** **Linking**




Errors:
None.

Example:

```
        .TITLE  EDEVIC;2
;+
; .DEVICE - This is an example in the use of the .DEVICE request.
; The example shows how .DEVICE is used to disable interrupts from
; a device upon termination of the program. In this case the device
; is a DL11 Serial Line Interface.
;-
        .MCALL          .DEVICE,.EXIT,.PROTECT,.UNPROTECT,.PRINT
.MACRO  ...
.ENDM
        .GLOBL          DL11            ;Routine
        .GLOBL          DLVEC           ;Vector
        .GLOBL          DLCSR           ;CSR
```

```
START:  .DEVICE     #AREA,#LIST     ;Setup disables DL11 interrupts
                                    ;on .EXIT or ^C^C
        .PROTECT    #AREA,#DLVEC+4  ;Protect the DL11 output vector
        BCS         BUSY            ;Branch if already protected
        ...                         ;Set up data to transmit over DL11
        JSR         R5,DL11         ;Use DL11 xfer routine (see .INTEN
                                    ;example)
        .WORD       128.            ;Arguments...Word count
        .WORD       BUFFR           ;Data buffer addr
        ...                         ;Continue processing...
FINI:   .UNPROTECT  #AREA,#DLVEC+4  ;...eventually to exit program
        .EXIT

BUSY:   .PRINT      #NOVEC          ;Print error message...
        .EXIT                       ;then exit

AREA:   .BLKW       3               ;EMT Argument block
LIST:   .WORD       DLCSR+4         ;CSR of DL11
        .WORD       0               ;Fill it with '0'
        .WORD       0               ;List terminator
BUFFR:                              ;Data to send over DL11
        .REPT       8.              ;8 lines of 32 characters...
        .ASCII      /Hello DL11... Are You There ??/
        .BYTE       15,12
        .ENDR
NOVEC:  .ASCIZ      /?EDEVIC-F-Vector already protected/
                                    ;Error message text

        .END        START
```

# .DRAST

**Macro Expansion (Handlers only)**

The .DRAST macro sets up the interrupt and abort entry points, lowers the processor priority, and enters the $INTEN routine in the resident monitor, which it finds by using the pointer $INPTR. This pointer is filled in by the bootstrap (for a system device) or RMON at .FETCH or load time (for a data device).

Macro Call:

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

where:

**name** is the two-character device name

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

**abo** is an optional argument that represents the label of an abort entry point. If you omit this argument, the macro generates a RETURN instruction at the abort entry point, which is the word immediately preceding the interrupt entry point.

Example:

```
        .TITLE    XX.MAC

;+
; XX.MAC - This is an example of a simple, RT-11 device driver to illustrate
; the use of the .DRBEG, .DRAST, .DRFIN, .DREND, .FORK & .QELDF requests.
; This driver could be used to output to a serial ASCII printer-terminal
; over a DL11 Serial Line Interface. To use this driver as an RT-11 device
; handler, simply install it via the INSTALL command (eg. 'INSTALL XX').
;-

        .MCALL  .DRDEF
        .DRDEF  XX,0,0,0,176504,304
                                ;MACRO expansion
;       .MCALL .DRAST,.DRBEG,.DRBOT,.DREND,.DREST,.DRFIN,.DRFMS,.DRFMT
;       .MCALL .DRINS,.DRPTR,.DRSET,.DRSPF,.DRTAB,.DRUSE,.DRVTB
;       .MCALL .FORK,.QELDF
;       .IIF NDF RTE$M RTE$M=0
;       .IIF NE RTE$M RTE$M=1
;       .IIF NDF TIM$IT TIM$IT=0
;       .IIF NE TIM$IT TIM$IT=1
;       .IIF NDF MMG$T MMG$T=0
;       .IIF NE MMG$T MMG$T=1
;       .IIF NDF ERL$G ERL$G=0
;       .IIF NE ERL$G ERL$G=1
;       .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
```

```
;;      Q$LINK=:Q.LINK-^o4
;;      Q$CSW=:Q.CSW-^o4
;;      Q$BLKN=:Q.BLKN-^o4
;;      Q$FUNC=:Q.FUNC-^o4
;;      Q$JNUM=:Q.JNUM-^o4
;;      Q$UNIT=:Q.UNIT-^o4
;;      Q$BUFF=:Q.BUFF-^o4
;;      Q$WCNT=:Q.WCNT-^o4
;;      Q$COMP=:Q.COMP-^o4
;;      Q.ELGH=:^o16
;       HDERR$=:1
;       EOF$=:^o20000
;       VARSZ$=:^o400
;       ABTIO$=:^o1000
;       SPFUN$=:^o2000
;       HNDLR$=:^o4000
;       SPECL$=:^o10000
;       WONLY$=:^o20000
;       RONLY$=:^o40000
;       FILST$=:^o100000
;       XXDSIZ=:0
;       XX$COD=:0
;       XXSTS=:<0>!<0>
;.IIF NDF XX$VEC,XX$VEC=304
;       .ASECT
;.IIF NDF XX$NAM,XX$NAM=^rXX
;       .=^o100
;       .WORD  0
;       .=^o176
;.IIF NDF XX$CSR,XX$CSR=176504
;       .WORD  XX$CSR

XX$PRI  = 4                       ;Priority of device

        .DRBEG    XX,XX$VEC       ;Begin driver code with .DRBEG
                                  ;MACRO expansion
;       .ASECT
;       .=^o52
;       .WORD  <XXEND-XXSTRT>
;       .WORD  XXDSIZ
;       .WORD  XXSTS
;       .WORD  ^o<ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>>
;       .PSECT XXDVR
;XXSTRT::
;       .WORD  XX$VEC&^C3.
;       .WORD  XXINT-.,^o340
;XXSYS::
;XXLQE::.WORD   0
;XXCQE::.WORD   0
;       .WORD  240
         MOV       XXCQE,R4    ;R4 => Current Q-Element
         ASL       Q$WCNT(R4)  ;Make word count byte count
         BCC       XXERR       ;A read from a write/only device?
         BEQ       XXDUN       ;Zero word count...just exit
XXRET:   BIS           #100,@#XX$CSR ;Enable DL-11 interrupt
         RETURN                ;Return to monitor

; INTERRUPT SERVICE ROUTINE
```

```
         .DRAST  XX,XX$PRI      ;Use .DRAST to define Int Svc Sect.
                                ;MACRO expansion...
;       RETURN
;XXINT::JSR     R5,@$INPTR
;       .WORD  ^C<XX$PRI*^o40>&^o340
         MOV        XXCQE,R4    ;R4 => Q-Element
         TST        @#XX$CSR    ;Error?
         BMI        XXRET       ;Yes...'hang' until read,
         BIC        #100,@#XX$CSR       ;Disable interrupts
         .FORK    XXFORK        ;Continue at FORK level
XXNXT:   TSTB          @#XX$CSR ;Is device ready?
         BPL        XXRET       ;No...go wait 'till it is
         MOVB      @Q$BUFF(R4),@#XX$CSR+2;Xfer byte from buffer to DL-11
         INC        Q$BUFF(R4)  ;Bump the buffer pointer
         INC        Q$WCNT(R4)  ;and the word count (it's negative!)
         BEQ        XXDUN       ;Branch if done
         BR          XXNXT      ;Try to output another character

XXERR:   BIS            #HDERR$,@Q$CSW(R4);Set error bit in CSW
XXDUN:   .DRFIN      XX  ;Use .DRFIN to return to Monitor
                                ;MACRO expansion...
;       .GLOBL XXCQE
;       MOV    PC,R4
;       ADD    #XXCQE-.,R4
;       MOV    @#^o54,R5
;       JMP    @^o270(R5)

XXFORK:  .WORD      0,0,0,0     ;Fork Queue Element

         .DREND  XX             ;Use .DREND to end code
                                ;MACRO expansion...
;       .PSECT XXDVR
;$INPTR::.WORD  0
;$FKPTR::.WORD  0
;XXEND==.

         .END
```

# .DRBEG

**Macro Expansion (Handlers Only)**

.DRBEG sets up a variable number of words (at least six) as the first words of the handler. The number of words set up by .DRBEG is determined by options selected in this and other .DRxxx macros. This macro also generates the appropriate global symbols for your handler. Before you use .DRBEG, you must invoke .DRDEF to define xx$CSR, xx$VEC, xxDSIZ, and xxSTS (See macro .DRDEF).

Macro Call:

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

where:

| | |
|---|---|
| **name** | is the 2-character device name. |
| **spsym** | is the symbol name for the list of DMA standard special functions. |
| **nspsym** | is the symbol name for the list of DMA nonstandard special functions. |

The arguments, *spsym* and *nspsym*, point to lists of special functions within the memory resident portion of the handler. The special functions are listed in the same manner as that used by the .DRSPF macro *extension table* method for defining special functions. Standard DMA special functions are listed in a group (assigned a symbol name) and that symbol name is used by .DRBEG (*spsym*). Nonstandard DMA special functions are listed in a separate group with a different symbol name and that name is then used by .DRBEG (*nspsym*).

The size of the area in a handler that is set up by .DRBEG can vary according to the options used to build the handler.

An additional word *(word 6)* has been added to the handler information set up in block 1 by .DRBEG. The word has the value of a "NOP" instruction. The lowest 5 bits (0 through 4) of the word are used as bit flags to indicate the presence of entry points in block 0 for the fetch, release, load, and unload handler service routines. Those entry points are generated by the .DRPTR request. You will always get at least six words, depending on options you specify to other of the .DR series of macros.

The following list shows the lowest 5 bits of the sixth word set up by .DRBEG and their meaning when set:

| **Bits** | **Contents** |
|---|---|
| 0 | Fetch entry point exists in block 0 |
| 1 | Release entry point exists in block 0 |
| 2 | Load entry point exists in block 0 |
| 3 | Unload entry point exists in block 0 |
| 4 | Second flag word exists |

Handlers and programs that interact with .DRBEG in a nonstandard manner or use the size of the code generated by .DRBEG must account for that additional word or words.

Example:
Refer to example in the *RT–11 Device Handlers Manual*.

# .DRBOT

**Macro Expansion (Handlers Only)**

.DRBOT generates a routine that issues a read request to read blocks 3-5 from the device into memory. It generates an error message routine used to report errors during the booting process. It generates a structure in which you write the read routine that does the reading during the bootstrap. As such, the .DRBOT macro sets up the primary driver. A primary driver must be added to a standard handler for a data device to create a system device handler. The .DRBOT macro invokes the .DREND macro to mark the end of the handler so that the primary driver is not loaded into memory during normal operations.

Macro Call:

**.DRBOT name,entry,read[,CONTROL=arg...,arg][,SIDES=n][,FORCE=n][,PSECT=psect]**

where

| | |
|---|---|
| **name** | is the two-character device name |
| **entry** | is the entry point of the software bootstrap routine |
| **read** | is the entry point of the bootstrap read routine |
| **CONTROL** | defines the types of controllers supported by this handler. The values for *arg* can be UBUS or QBUS. If CONTROL is omitted, both Unibus and Q-bus are assumed. This is correct for all supported handlers |
| **SIDES** | specifies single- or double-sided diskettes. If omitted, single-sided diskettes are assumed. This is correct for all supported handlers |
| **FORCE**<br>**PSECT** | Both arguments are passed to a .DREND macro automatically generated by .DRBOT. See .DREND request for their use. |

.DRBOT macro puts a pointer to the start of the primary driver into location 62 of the handler file. It puts the length (in bytes) of the primary driver into location 64. Location 66 of the handler file contains the offset from the start of the primary driver to the start of the bootstrap read routine. The .DRBOT macro starts the bootstrap area in the handler, this area is ended by a .DREND macro which you must explicitly issue.

Example:

Refer to the *RT–11 System Subroutine Library Manual* for an example showing the use of .DRBOT.

# .DRDEF

**Macro Expansion (Handlers Only)**

The .DRDEF macro sets up handler parameters, calls the driver macros from the library, and defines useful symbols.

Four optional parameters, UNIT64, DMA, PERMUMR, and SERIAL, have been added to the .DRDEF macro.

Macro Call:

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

where:

**name** is the two-character device name. See Table 2–3.

**code** is the numeric code that is the device identifier value for the device. See Table 2–3.

**stat** is the device status bit pattern. The value for stat may use the following symbols:

| | | |
|---|---|---|
| FILST$ = 100000 | SPECL$ = 10000 | ABTIO$ = 1000 |
| RONLY$ = 40000 | HNDLR$ = 4000 | VARSZ$ = 400 |
| WONLY$ = 20000 | SPFUN$ = 2000 | |

**size** is the size of the device in 256-word blocks.

If the device is not random access, place the value 0 in the .DRDEF parameter *size*. The size of the RK device is $4800_{10}$ blocks ($11300_8$); the size for the MS (TS11 magtape) device is 0, since it is not random access.

**csr** is the default address for the device's control and status register

**NOTE**

If you specify CSR as *NO*, you prevent it from filling in INSCSR.

**vec** is the default address for the device's vector

**UNIT64** indicates whether this handler supports extended device units. For the UNIT64 parameter, valid arguments for *str* are:

- **no** The default. This handler does not support extended device units
- **yes** This handler supports extended device units

**DMA** indicates whether this handler supports direct memory access. For the DMA parameter, valid arguments for *str* are:

- **Yes** This handler supports direct memory access
- **No** This handler does not support direct memory access. There is no default.

**PERMUMR** indicates this handler should be assigned n permanent UNIBUS mapping registers. Valid values for *n* are 0 through 7. The default is 0. The PERMUMR parameter implies support for DMA for this handler; if you specify PERMUMR you need not specify DMA. If you specify DMA you must use the argument YES.

**SERIAL** indicates whether this handler requires serialized I/O request satisfaction. Magtape handlers, for example, should include this parameter. The UB pseudohandler checks for the inclusion of this parameter in determining when I/O request satisfaction must be serialized. For the SERIAL parameter, valid arguments for *str* are:

- **No** The default. This handler does not require serialized I/O request satisfaction
- **Yes** This handler requires serialized I/O request satisfaction

### Device-Identifier Byte

The low byte of the device status word, the device-identifier byte, identifies each device in the system. You specify the correct device identifier as the *code* argument to .DRDEF. The values are currently defined in octal as Table 2–3 shows.

**Table 2–3: Device-Identifier Byte Values**

| Name | Code | Device |
|---|---|---|
| RK | 0 | RK05 Disk |
| | 1 | Reserved |
| EL | 2 | Error Logger |
| LP | 3 | Parallel Interface Printer |
| TT | 4 | Console terminal |
| DL | 5 | RL01/RL02 Disk |
| DY | 6 | RX02 Diskette |
| | 7 | Reserved |
| VS | 10 | RTEM Virtual System VS(M) |
| MT | 11 | TM11/TMA11/TU10/TS03 Magtape |

**Table 2–3 (Cont.): Device-Identifier Byte Values**

| Name | Code | Device |
|---|---|---|
| | 12-17 | Reserved |
| MM | 20 | TJU16/TU45 Magtape |
| | 21 | Reserved |
| DX | 22 | RX11/RX01 Diskette |
| DM | 23 | RK06/RK07 Disk |
| | 24 | Reserved |
| NL | 25 | Null Device |
| | 26–30 | Reserved (DECnet) |
| | 31–33 | Reserved (CTS–300) |
| | 34 | Reserved |
| MS | 35 | TS11/TS04/TS05 Magtape |
| | 36–40 | Reserved |
| LS | 41 | Serial Interface Printer |
| MQ | 42 | Internal Message Handler |
| DR | 43 | DRV11J Interface (MRRT) |
| XT | 44 | Reserved (MRRT) |
| | 45 | Reserved |
| LD | 46 | Logical disk handler |
| VM | 47 | KT11 pseudodisk handler |
| DU | 50 | MSCP disk class handler |
| SL | 51 | Single-line Command Editor |
| DZ | 52 | RX50 diskette (CTI Bus-based processor) |
| DW | 53 | Hard Disk (CTI Bus-based processor) |

**Table 2–3 (Cont.): Device-Identifier Byte Values**

| Name | Code | Device |
|---|---|---|
| PI | 54 | Professional interface |
| SP | 55 | Transparent spooler |
| | 56 | Reserved |
| XC/XL | 57 | Communication port (Professional 325/350 or DL(V)–11) |
| MU | 60 | TMSCP magtape class handler |
| NC/NQ/NU | 61 | Ethernet class handler |
| SD | 62 | DBG–11 handler |
| ST | 63 | DBG–11 symbol table handler |
| | 64 | Reserved |
| UB | 65 | UMR pseudohandler |

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

The .DRDEF macro performs the following operations:

- A .MCALL is done for the following macros: .DRAST; .DRBEG; .DRBOT; .DREND; .DRFIN; .DRINS; .DRSET; .DRVTB; .FORK; .QELDF.
- If the system generation conditionals TIM$IT, MMG$T, or ERL$G are undefined in your program, they are defined as zero. If time-out support is selected, the .DRDEF macro does a .MCALL for the .TIMIO and .CTIMIO macros.
- The .QELDF macro is invoked to define symbolic offsets within a queue element.
- The symbols listed above are defined for the device status bits.
- The following symbols are defined:

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

- The symbol xxDSIZ is set to the value specified in size.
- The symbol xx$COD is set to the specified device identifier code.
- The symbol xxSTS is set to the value of the device identifier code plus the status bits.
- If the symbol xx$CSR is not defined, it is set to the default csr value.
- If the symbol xx$VEC is not defined, it is set to the default vector value.
- The symbols xx$CSR and xx$VEC are made global.

You should invoke the .DRDEF macro near the beginning of your handler, after all handler specific conditionals are defined.

Example:

See example shown for .DRAST.

# .DREND

**Macro Expansion (Handlers Only)**

The .DREND macro generates the table of addresses for service routines in RMON.

Macro Call:

**.DREND name[,FORCE=n][,PSECT=psect]**

where:

**name** is the two-character device name

**FORCE** Value specified in FORCE, combined with the settings of MMG$T, ERL$G, and TIM$IT, selects the entries to be generated in the vector table.

See Table 2–4 for the values in FORCE corresponding to the SYSGEN options. For example, specifying FORCE = 4 generates the device timeout vector in the table. Generating a vector in the handler vector table does not create support in that handler for a SYSGEN feature. The default value for FORCE is 0. Using a value of -1 for FORCE will generate all the possible entries of the System Service Table.

**PSECT** forces the .DREND request to be placed in the specified program section at link time. Use this argument when the handler is built from several PSECTs and you want to force location of .DREND code to properly determine the end of the memory-resident section of the handler.

The generation of the termination table is dependent upon certain conditions. See Table 2–4.

**Table 2–4: System Service**

| Label | Address | Condition | Source Value |
|---|---|---|---|
| $RLPTR: | .WORD 0 ($RELOC) | MMG$T=1 | 2 |
| $MPPTR: | .WORD 0 ($MPPHY) | MMG$T=1 | 2 |
| $GTBYT: | .WORD 0 ($GETBYT) | MMG$T=1 | 2 |
| $PTBYT: | .WORD 0 ($PUTBYT) | MMG$T=1 | 2 |
| $PTWRD: | .WORD 0 ($PUTWRD) | MMG$T=1 | 2 |
| $ELPTR: | .WORD 0 ($ERLOG) | ERL$G=1 | 1 |
| $TIMIT: | .WORD 0 ($TIMIO) | TIM$IT=1 | 4 |
| $INPTR: | .WORD 0 ($INTEN) | always | - |
| $FKPTR: | .WORD 0 ($FORK) | always | - |

The generation of the labels depends upon options chosen during the system generation process. All the pointers in the shown in Table 2–4 are initialized when the handler is loaded into memory with the .FETCH request, the LOAD command or as the system device at bootstrap time. The pointers are initialized with the address shown in the address column.

The addresses are located within the monitor. The first five addresses are locations of subroutines that are available to device handlers under mapped monitors. Device I/O time-out service is provided by $TIMIO and error logging is provided by $ERLOG. The $INPTR and $FKPTR labels are always filled in. For further information, see the *RT–11 Device Handlers Manual*.

Example:
Refer to the example for .DRAST.

#### Macro Expansion (Handlers Only)

The .DREST macro places device specific information in block 0 of a device handler. The device specific information includes:

- The device class
- The variants of a device class and additional information about some device classes
- Whether the device handler contains updatable internal data table(s) accessible by SPFUN 372
- The type (device class) of the updatable internal data table
- Whether the device handler has a table in block 0 that contains bad-block replacement information
- How the handler can be installed, loaded, and mounted

That information is used by RT–11 utilities to determine the characteristics of that device handler.

Macro Call:

```
.DREST  [CLASS=n][,MOD=n][,DATA=dptr][,TYPE=n][,SIZE=n][,REPLACE=rptr]
        [,MOD2=n][,STAT2=symb])
```

where:

**CLASS** is the device class. Specify the device class symbol (DVC.xx) for *n* in the *CLASS* argument. An octal device class value is stored in byte 20 of block 0 in the device handler. The following table lists valid device class symbols and stored values for the *CLASS* argument.

| **Symbol** | **Value** | **Meaning** |
|---|---|---|
| DVC.CT | 6 | Cassette tape (Obsolete; not supported) |
| DVC.DE | 10 | DECnet executive pseudohandler |
| DVC.DK | 4 | RT–11 file structured disk (DD, DL, DM, DP, DT, DU, DW, DX, DY, DZ, LD, RK, VM) |
| DVC.DL | 12 | DECnet port (line) handler |
| DVC.DP | 11 | DECnet protocol pseudohandler |
| DVC.LP | 7 | Printer (LP, LS, SP) |
| DVC.MT | 5 | Magtape (MM, MS, MT, MU) |
| DVC.NI | 13 | Ethernet port handler (NC, NQ, NU) |
| DVC.NL | 1 | NULL handler (AT and NL) |

| | | |
|---|---|---|
| DVC.PS | 14 | Pseudohandler (PI, SD, SL, ST, UB) |
| DVC.SB | 20 | Serialized input/output (PC and generic) |
| DVC.SI | 16 | Serialized input (generic) |
| DVC.SO | 17 | Serialized output (generic) |
| DVC.TP | 3 | Reserved |
| DVC.TT | 2 | Terminal class handler (BA and TT) |
| DVC.UK | 0 | Unknown device class |
| DVC.VT | 15 | Virtual terminal port handler (XL, XC) |

Values in the range of 200 through 377 are reserved for the user.

**MOD** indicates a variation or additional information about a device class. Specify the device modification symbol (DVM.xx) for *n* in the *MOD* argument. A device modification value is stored in byte 21 of block 0 in the device handler. Valid device modification symbols and stored values for the *MOD* argument are:

| Symbol | Value | Meaning |
|---|---|---|
| (None) | 0 | No variant or information (default) |
| DVM.DM | 2 | With CLASS=DVC.DK, indicates device has an extra error word prefixed to SPFUN 376 and SPFUN 377 buffers |
| DVM.DX | 1 | With CLASS=DVC.DK, indicates device is an RX01-compatible drive |
| DVM.NS | 1 | With CLASS=DVC.MT, indicates file handler has no file structure support |
| DVM.NF | 200 | With all class devices, indicates handler can only be loaded and cannot be fetched. This bit is read-only and cannot be set using the .DREST macro. (This bit is set by the .DRPTR macro with the FETCH=*NO* argument.) |

| | | |
|---|---|---|
| DVM.NL | 100 | With all class devices, indicates handler cannot be loaded. Bit DVM.NL is set by the .DRPTR macro *LOAD=*NO** argument. |

**DATA** specifies whether the handler has internal updatable data table(s) accessible by SPFUN 372. The *DATA* argument information is stored in word 72 of block 0 in the handler. You must include the *TYPE* argument if you specify the *DATA* argument. For the *DATA* argument, *dptr* can be:

0 The default; specifies that the handler does not have an internal data table

dptr is the file address of the internal data table(s). The file address is a symbol that is defined within the handler and associated by the linker with a file address.

**TYPE** specifies whether a device type classification exists for the internal data table(s). The device type classification is made up of one-to-three RAD50 characters and is normally the same as the RT–11 device name. The *TYPE* argument information is stored in word 70 of block 0 in the handler. You must include the *TYPE* argument *rad* if you specify the *DATA* argument *dptr*. For the *TYPE* argument, *n* can be:

omitted The default; the handler does not have a device type classified internal table

rad The handler has a device type classified internal table, and *rad* is the RAD50 device type classification

**SIZE** Provides size of table (pointed to by *DATA*), stored in word 74 of block 0 in the handler.

**REPLACE** specifies whether the handler has a table in block 0 that contains bad-block replacement geometry information. The distributed DL and DM handlers have a bad block replacement geometry table of this type. The replace argument information is stored in word 32 of block 0 in the handler. For the *REPLACE* argument, *rptr* can be:

0 The default; the handler does not contain a bad-block replacement geometry table

| | |
|---|---|
| rptr | Is the file address of a bad-block replacement geometry table. The file address is a symbol that is defined within the handler and associated by the linker with a 16-bit value. |

**MOD2** Currently only supports the LS handler:

| | |
|---|---|
| DV2.V2 | First .DRVTB table is followed by second display only. Sets the 40000 bit in XX'CQE in the file. |

**STAT2** specifies the conditions under which the handler can be installed, loaded, and mounted. The bit flag symbols can be OR'd to indicate more than one condition. The *STAT2* argument information is stored in the second handler status word (H.STS2) at location 36 of block 0 in the handler. For the *STAT2* argument, *symb* can be:

| Symbol | Meaning |
|---|---|
| 0 | The default; .DREST specifies no restrictions concerning installation, loading, or mounting, which does not imply that such restrictions do not exist elsewhere |
| HS2.BI | The handler cannot be installed by the monitor bootstrap (BSTRAP) |
| HS2.KI | The handler cannot be installed by the INSTALL command |
| HS2.KL | The handler cannot be loaded by the LOAD command. The HS2.KL bit flag can be set by the .DRPTR LOAD=*NO* parameter argument |
| HS2.KU | The handler cannot be unloaded by the UNLOAD command |
| HS2.MO | The handler can be mounted by the MOUNT command and dismounted by the DISMOUNT command |

Although all .DREST arguments are optional, some arguments are paired. For example, the *mod* argument has no meaning without the *class* argument. Also, the *data* argument requires the *type* argument.

Errors:
None

Example:

```
        .Title  SK -- Handler Skeleton
;+
; .DRPTR/.DREST/.DRSPF - This is an example skeleton handler
; that illustrates using the .DRPTR, .DREST, and .DRSPF requests.
;-
```

```
        .MCALL  .DRDEF          ; Get handler definitions
        .MCALL  .ASSUME         ; Checking macro
        .MCALL  .EXIT           ; To finish run

        .MACRO  ...             ; Define ellipsis (allow
                                ;  ellipsis to assemble)
        .ENDM

        ; Generate nonexecutable handler information tables
        ;  containing the following information:
        ;  Handler is SK
        ;  Handler ID is 350 (user-written handler)
        ;  Handler accepts neither .READ nor .WRITE
        ;  Handler accepts .SPFUN requests
        ;  Device is 1 block in size
        ;  Device has a CSR at 176544
        ;  Device has a vector at 20

        .DRDEF  SK,350,RONLY$!WONLY$!SPFUN$,1,176544,20

                ; Handler has .Fetch and $LOAD code to be executed:

        .DRPTR  FETCH=Fetch,LOAD=Load

                ; Handler is for a "Null" class device
                ; Handler has a data table called DATABL
                ; Data table is of the SKL format

        .DREST  CLASS=DVC.NL,DATA=DATABL,TYPE=SKL

                ; Handler accepts the following SPFUN codes:
                ;  372,376,377

        .DRSPF  372,TYPE=T
        .DRSPF  376,TYPE=W
        .DRSPF  377,TYPE=R

                ; Handler CSR is not to be checked at install,
                ;  but is to be displayed:

        .DRINS  -SK
        ...     ; Here is any installation check code
        RETURN

        .ASSUME . LE 400,MESSAGE=^";Installation area overflow"

                ; Handler accepts SET SK [NO]BONES command:

        .DRSET  BONES,123456,CORPUS,NO

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

        .ASSUME . LE 1000,MESSAGE=^";Set area overflow"
```

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

                ;Set up the Vector table:

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

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

START:                  ; Executable Queue code
        ...
        RETURN

        .DRAST  SK,4,ABORT      ; Interrupt entry point
        BCS     INT2            ; Interrupt from second vector
        ...
        RETURN
INT2:                           ; Second interrupt vector code
        ...
        RETURN

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

                                ; End of memory resident part of handler

        .DRBOT  SK,ENTRY        ; Boot code

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

READ:
        ...                     ; Read routine
        RETURN

        .DREND  SK              ; End of boot code

        .PSECT  SETOVR          ; Suggested block aligned PSect

FETCH:
        ...                     ; Code executed on FETCH
        RETURN
LOAD:
        ...                     ; Code executed on LOAD
        RETURN

RUN:
        ...                     ; Code executed on RUN
        .EXIT
        .END    RUN
```

# .DRFIN

### Macro Expansion (Handlers Only)

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

Macro Call:

**.DRFIN name**

where:

**name** is the two-character device name

Errors:
None.

Example:
Refer to the example for .DRAST.

# .DRINS

**Macro Expansion (Handlers Only)**

The .DRINS macro defines the following:

- Symbols for the locations that contain the CSR addresses listed by RESORC (display CSRs) and the CSR checked by the INSTALL keyboard command.
- Separate entry points for installing the handler as a system device or as a data device.
- List of CSR addresses in block 0.

Macro Call:

**.DRINS name,<csr,csr,...>**

where:

**name** The two-letter device mnemonic.

**– name** Specifying CSR = *NO* to the .DRDEF macro prevents it from filling in INSCSR.

**csr** Specifies a symbolic CSR address for that device. If more than one display CSR exists, separate them with commas and enclose the list in angle brackets <>. With multiple display CSRs (For example, first CSR is offset 176, second CSR is offset 174...), you do not have to list the first CSR.

When the .DRINS macro is processed, the following symbols are defined, based on the CSR addresses you provide:

**INSCSR** Installation check CSR

**DISCSR** First display CSR

**DISCSn** Subsequent display CSRs if any exist (n begins at 2 and is incremented by 1 for each subsequent display CSR)

In addition, the .DRINS macro sets the location counter to 200. It defines the symbols INSDAT =: 200) for the data device installation entry point, and defines the label INSSYS as 202 (INSSYS =: 202), the system device installation entry point.

The following example shows the installation code generated by a .DRINS macro used for a DX handler with two controllers.

```
.TITLE EDRIN1.MAC
       .DRINS  DX,DX$CS2              ;Generate installation code
                                      ;for two-controller RX01
```

The next example shows the installation code generated by a .DRINS macro used for a DU handler with three controllers.

```
 .TITLE EDRIN3.MAC

        .DRINS  -DU,^/DU$CS2,DU$CS3/    ;GENERATE INSTALLATION CODE
                                        ;FOR THREE-CONTROLLER
                                        ;MSCP DEVICE

        .TITLE  EDRIN4.MAC

.=166
        .WORD   0                       ;End of list
DISCS3: .WORD   DU$CS3                  ;Third display CSR
DISCS2: .WORD   DU$CS2                  ;Second display CSR
DISCSR: .WORD   DU$CSR                  ;First display CSR
INSCSR: .WORD   0                       ;Install CSR (none)
INSDAT:
.=202
INSSYS:
.=200
```

# .DRPTR

**Macro Expansion (Handlers Only)**

The .DRPTR macro places pointers in block 0 of a device handler that references handler service routines located at a file address in that handler. The file address is a symbol that is defined within the handler and associated by the linker with a 16-bit value.

Handler service routines, used by utilities, monitors, and the handler itself, help govern how the handler behaves during:

- Bootstrap operations (load argument)
- .FETCH and .RELEASE requests
- LOAD and UNLOAD commands
- Job abort (release argument)

Macro Call:

**.DRPTR [FETCH=n][,RELEASE=n][,LOAd=n] [,UNLOAD=n]**

where:

**fetch** specifies whether a handler service routine is called by the .FETCH programmed request. For the *FETCH* argument, *n* can be:

| | |
|---|---|
| 0 | The default; the handler does not have a service routine for the .FETCH programmed request. The handler can still be fetched |
| n | is the file address of the service routine to be called by .FETCH |
| *NO* | A literal string; the handler cannot be fetched. The handler can only be loaded.<br>Invalid with *load=*NO** parameter argument. |

**release** specifies whether a handler service routine is called by the .RELEASE programmed request. For the *release* argument, *n* can be:

| | |
|---|---|
| 0 | The default; the handler does not have a service routine for the .RELEASE programmed request |

n — is the file address of the service routine to be called by .RELEASE

**load** specifies whether a handler service routine is called when the handler is loaded by bootstrap routine or LOAD command. For the *load* argument, *n* can be:

0 — The default; the handler does not have a service routine to be called when it is loaded

n — is the file address of the service routine to be called when the handler is loaded

*NO* — A literal string; the handler cannot be loaded. The handler must be fetched. Invalid with *fetch=*NO** parameter argument

**unload** specifies whether a handler service routine is called when the handler is unloaded by the UNLOAD command. For the *unload* argument, *n* can be:

0 — The default; the handler does not have a service routine to be called when it is unloaded

n — is the file address of the service routine to be called when the handler is unloaded

.DRPTR arguments are often paired and argument values are often matched because routines they point to are used together or rely on each other. The *fetch* and *load* argument values, other than **NO**, are often paired. Similarly, the *release* and *unload* argument values are often the same.

Errors:
None.

Example:
See .DREST.

# .DRSET

**Macro Expansion (Handlers Only)**

The .DRSET macro sets up the option table for the SET command in block 0 of the device handler file. The option table consists of a series of four-word entries, one entry per option. Use this macro once for each SET option that is used. When used a number of times, the macro calls must be sequential.

Macro Call:

**.DRSET option,val,rtn[,mode]**

where:

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

**val** is a parameter that is passed to the routine in R3. It can be a numeric constant, such as minimum column width, or any one-word instruction that is substituted for an existing one in block 1 of the handler. It must not be a zero

**rtn** is the name of the routine that modifies the code in block 1 of the handler. The entry point must be between offsets 400 and 776 in block 0.

**mode** is an optional argument to indicate the type of SET parameter

A NO indicates that a NO prefix is valid for the option. NUM indicates that a decimal numeric value is required. OCT indicates that an octal numeric value is required. Omitting this argument indicates that the option takes neither a NO prefix nor a numeric argument. NO may be combined with NUM or OCT.

The .DRSET macro does an .ASECT and sets the location counter to 400 for the start of the table. The macro also generates a zero word for the end of the table and leaves the location counter there. In this way, routines to modify codes are placed immediately after the .DRSET calls in the handler, and their location in block 0 of the handler file is made certain.

Errors:
None.

Example:
See .DREST.

# .DRSPF

**Macro Expansion (Handlers Only)**

The .DRSPF macro defines the special function codes supported by a handler. .DRSPF builds a table or tables containing the supported special function codes. RT–11 utilities or user programs use the table(s) to determine which special function codes are supported by that handler. It is also used to generate tables of SPFUN code to be used by error logging and UMR support.

Macro Call:

**.DRSPF arg[,arg2][,TYPE=n]**

where:

**arg** can be specified in two ways: the *list* method and the *extension table* method which is discussed below.

**arg2** is a list of special function codes. Only use the *arg2* argument to specify special function codes in an extension table, that is, when the *arg* argument is a minus sign (-). See the discussion of the *arg* argument in the extension table method description.

**TYPE=n** is an optional parameter specifying the type of special function or functions that are coupled with this parameter. Include the *TYPE=n* parameter only when your handler uses special functions and you want to enable RT–11 error logging for the device controlled by the handler or for use with UMR support.

The RT–11 error logger does not recognize the type of operation performed by special function codes. Therefore, error logging for devices that use special functions to perform read, write, or motion operations requires the *TYPE=n* parameter to indicate which special function codes perform the type of operation logged.

Specifying the *TYPE=n* parameter causes a bit value representing the symbol *n* to be stored in bits 8, 9, and 10 of the word generated by that invocation of .DRSPF. Table 2–5 shows the valid symbols, bit pattern for bits 8, 9, and 10, and the meaning for the *n* argument:

**Table 2–5: Special Functions for the TYPE=n Parameter**

| Symbol | Bit Pattern | Meaning |
|---|---|---|
| O | 000 | The letter O (default) indicates unknown special function code type. |
| R | 001 | A read operation special function code. Any operation that obtains data from a device is defined here as a read operation. |
| W | 010 | A write operation special function code. Any operation that directs data to a device is defined here as a write operation. |
| M | 011 | A motion operation special function code. Any operation whose sole purpose is to cause the device to move is defined here as a motion operation. |
| T | 100 | A read/write operation special function code. The sign of the bit for the special function word count (wcnt) parameter (determined by a special function subcode) determines if the operation is a read or a write. |
| | 101-111 | Reserved for Digital. |

### List Method

*Arg* can be a list of one or more special function codes. That list is located in block 0 of the handler at locations 22 through 27.

The list method is the simpler of the two methods, but you must adhere to certain rules in specifying the list because of the way the special function codes are stored in the handler. If this restriction is a problem, you may find the extension table more useful.

Special function codes consist of three octal digits. When *arg* is used in this manner, you are allowed a total of only three unique, ordered combinations of the first two digits of a special function code in all lists or combination of lists. You can use any octal digit as the third digit of any entry in those lists. That restriction is not a problem for most handlers. You can define all supported special functions for most handlers in one list or series of lists. For more information, refer to the *RT–11 Device Handlers Manual*.

Each list must be enclosed in angle brackets (<>). Special function codes are separated by commas (,). The special function codes can be specified in any order.

Do not specify the *arg2* argument when using the .DRSPF macro in this manner.

An example list for the MU handler:

```
        .TITLE  EDRSP1.MAC
.DRSPF ^/360,370,371,372,373,374,375,376,377/
```

The same special functions for the MU handler could also be included in a series of lists:

```
        .TITLE  EDRSP2.MAC

.DRSPF 360
.DRSPF ^/370,371,372/
.DRSPF ^/373,374,375/
.DRSPF ^/376,377/
```

In both examples, only two unique ordered combinations of the first two digits (36 and 37) of the special function codes were used.

Each of the ordered combinations of the first two digits, together with the various third digits supported for that combination, is stored in a single word. Bits 8, 9, and 10 of that word are used to indicate the special function code type. See the *TYPE=n* parameter.

Using the *TYPE=n* parameter can reduce the number of special function codes you are allowed with the *list* method. Each type of special function code you specify with the *TYPE=n* parameter requires one of the three allowable words and you can specify only one combination of the first two code digits with each invocation.

For example, the following invocations of .DRSPF are valid for the *list* method for one handler:

```
.TITLE  EDRSP4.MAC

.DRSPF  ^/377,374/,TYPE=R
.DRSPF  376,TYPE=W
.DRSPF  373
```

#### Extension Table Method

*Arg* can be a pointer to an extension table address. Do not place the extension table in block 0 of the handler.

The pointer to the extension table address must be prefixed by a plus sign (+). The extension table address must have the high bit cleared.

The extension table contains one or more .DRSPF macros. The *arg* argument for each .DRSPF macro is a minus sign (-), and the *arg2* argument is a list of special function codes. Each of the special function codes in *arg2* must have the same first two octal digits. The list must be enclosed by angle brackets (<>). Special function codes are separated by commas (,). The special function codes can be specified in any order. The extension table is terminated by a word containing zero (0).

Each of the ordered combinations of the first two digits, together with the various third digits supported for that combination, are stored in a single word. Bits 8, 9, and 10 of that word indicate the special function code type. See the *TYPE=n* parameter.

Errors:
None.

In the following example, the pointer to the extension table is the symbol EXTTAB:

```
        .TITLE  EDRSP3.MAC

        .DRSPF +EXTTAB
        ...
EXTTAB:
        .DRSPF  -,^/340,341/
        .DRSPF  -,^/350,351,353/
        .DRSPF  -,^/200,202,203,204,205/
        .DRSPF  -,^/210,212/
        .WORD  0
```

The following invocations of .DRSPF are valid for the *extension table method*, since the third invocation, containing code 354, requires a fourth word. Because this fourth word would exceed the three-word limit of the *list* method, you must use the *extension table* method and include code 354 in a fourth invocation.

```
        .TITLE  EDRSP5.MAC

        .DRSPF  371,TYPE=W
        .DRSPF  370,TYPE=R
        .DRSPF  ^/360,354/
```

Example:

```
        .TITLE  EDRSP6.MAC

        .MCALL   .DRSPF  ; Get macro

        .MACRO  ...                     ; Elision macro
        .ENDM   ...                     ; Elision - "act of dropping
                                        ;  out or suppressing"
                                        ;  allows ellipsis to assemble

        ...                             ; Handler continues...

        .DRSPF  370,TYPE=R              ; A Read request
        .DRSPF  371,TYPE=W              ; A Write request
        .DRSPF  367                     ; That's all we can support
                                        ;  with list method -
                                        ; More are supported so
        .DRSPF  +XSPTAB                 ;  point to extension table

        ...                             ; Handler continues beyond
                                        ;  block 0
XSPTAB:                                 ; SPFUN extension table
        .DRSPF  -,^/200,202,207,203/    ; 20X group
        .DRSPF  -,^/222,224,227/,TYPE=M ; 22X group
        .WORD   000000                  ; End of list

        .END
```

See also the .DREST example in this manual.

# .DRTAB

**Macro Expansion (Handlers Only)**

The .DRTAB macro establishes the file address of a list of Digital-defined handler data tables that are part of the RT–11 distributed handlers and is for Digital use only. The file address is the number of bytes from the beginning of the file. A similar macro, .DRUSE, is available for user-defined handler data tables.

.DRTAB is included in a distributed handler when that handler contains more than one data table. Other distributed components can then reference the data tables. For example, the distributed SL and LET use .DRTAB to reference data tables and share data.

The relationship between the .DRTAB macro and .DREST macro is:

- When a distributed handler contains only one handler data table, .DREST is used to describe that table.
- When a distributed handler contains more than one handler data table, the .DRTAB macro is used to describe all those tables. The .DREST macro can be included in a handler that includes the .DRTAB macro because other information can be placed in the handler by .DREST. However, when the .DREST macro is included in a handler that also includes the .DRTAB macro, the .DREST macro does not contain the type and data arguments. (The information placed in the handler by those .DREST macro arguments would be destroyed (overwritten) by the .DRTAB macro type argument.)

Macro Call:

**.DRTAB type,addr,size**

where:

| | |
|---|---|
| **type** | is the handler data table format name in one to three RAD50 characters |
| **addr** | is the file address of the handler data table |
| **size** | is the size in bytes of the handler data table |

Request Format:

| | |
|---|---|
| R0= | type |
| | addr |
| | size |

.DRTAB is invoked once for each handler data table. Each invocation of .DRTAB creates a 3-word descriptor containing the values specified for the type, addr, and size arguments. A call to .DRTAB with no arguments specifies the end of that list of handler data table descriptors.

.DRTAB places the file address of the list of handler data table descriptors in block 0 of the handler. The list of descriptors and the data tables themselves are not located in block 0. When first invoked, .DRTAB sets up locations 70 through 74 in block 0 with the following contents:

| Location | Location After .DRTAB Is Invoked |
|---|---|
| 70 | -1 (indicates use of .DRTAB) |
| 72 | Pointer to list of handler data table descriptors |
| 74 | Size in bytes of total list of handler data table descriptors |

# .DRUSE

**Macro Expansion (Handlers Only)**

The .DRUSE macro establishes the file address of a list of user-defined handler data tables. The file address is the number of bytes from the beginning of the file. Use .DRUSE when you want to define your own handler data table(s).

Macro Call:

**.DRUSE type,addr,size**

where:

**type** is a handler data table format name in one to three RAD50 characters

**addr** is the file address of the handler data table

**size** is the size in bytes of the handler data table

Request Format:

| | |
|---|---|
| R0 = | type |
| | addr |
| | size |
| | 0 |

Invoke .DRUSE once for each user-defined handler data table in your handler. Each invocation of .DRUSE creates a 3-word descriptor containing the values you specified for the *type, addr*, and *size* arguments. Call .DRUSE with *no* arguments to indicate the end of the list of descriptors.

.DRUSE places the file address of the list of handler data table descriptors in block 0 of your handler. Do not place the list of descriptors or the data tables themselves in block 0 of your handler. When you first invoke .DRUSE, it sets up location 106 in block 0 with the following contents:

| Location | Contents After .DRUSE Is Invoked |
|---|---|
| 106 | Pointer to list of handler data table descriptors |

Errors:
None.

Example:

```
        .TITLE  EDRUSE.MAC

.PSECT  OUTMEM

        .DRUSE  JFW,JIMS,SZJIM
        .DRUSE  JBM,JIMS2,SZJIM2
        .DRUSE

JIMS:   .WORD   ^rJFW,1,2,3,4,5,6,7,8.
SZJIM=: .-JIMS
JIMS2:  .WORD   ^rJBM,10,1000,100000
SZJIM2=:.-JIMS2
```

# .DRVTB

### Macro Expansion (Handlers Only)

The .DRVTB macro sets up a table of three-word entries for each vector of a multivector device. The table entries contain the vector location, interrupt entry point, and processor status word. You must use this macro once for each device vector. The .DRVTB macros must be placed consecutively in the device handler between the .DRBEG macro and the .DREND macro. They must not interfere with the flow of control within the handler.

Macro Call:

**.DRVTB name,vec,int[,ps]**

where:

**name** is the two-character device name. This argument must be blank except for the first-time use of .DRVTB

**vec** is the location of the vector, and must be between 0 and 474

**int** is the symbolic name of the interrupt handling routine. It must appear elsewhere in the handler code. It generally takes the form ddINT, where dd represents the two-character device name

**ps** is an optional value that specifies the low-order four bits of the new Processor Status Word in the interrupt vector. This argument defaults to zero if omitted. The priority bits of the PS are set to 7, even if you omit this argument

Errors:
None.

Example:
Refer to the *RT–11 Device Handlers Manual* for an example of .DRVTB.

# .DSTAT

**EMT 342**

This .DSTAT request obtains information about a particular device. Refer to *RT–11 Device Handlers Manual* for details.

Macro Call:

**.DSTAT retspc,dnam**

where:

**retspc** is the address of a four-word block that stores the status information

**dnam** is the address of a word containing the Radix–50 device name.

.DSTAT looks for the device specified by *dnam* and, if successful, returns four words of status starting at the address specified by *retspc*. The four words returned are as follows:

**Word 1** **Status Word**

Bits 0-7: The low-order byte contains a numeric code (.DEVDF) that is the device identifier value for the device in the system.

Value is the numeric *code* parameter returned from the .DRDEF request. For more information, refer to Table 2–3 under the .DRDEF discussion.

Bits 8-15: The low-order byte contains value for stat (.DSTDF), a device status bit pattern returned from the .DRDEF request. Values may use the following symbols:

| | | |
|---|---|---|
| FILST$ = 100000 | SPECL$ = 10000 | ABTIO$ = 1000 |
| RONLY$ = 40000 | HNDLR$ = 4000 | VARSZ$ = 400 |
| WONLY$ = 20000 | SPFUN$ = 2000 | |

**Word 2:** **Handler Size**

The size of the device handler in bytes.

**Word 3:** **Load Address +6**

*Non-zero* implies the handler is now in memory. The address returned is the load address of the handler +6.

*Zero* implies that it must be fetched before it can be used.

**Word 4:** **Device Size**

The size of the device (in 256-word blocks) for block-replaceable devices; 0 for sequential-access devices, the smallest-sized volume for variable-sized devices. The last block on the device is the device size -1.

### Notes

The device name can be a user-assigned name. .DSTAT information is extracted from the device handler. Therefore, this request requires the handler for the device to be present on the system device and installed on the system. Refer to *RT–11 Device Handlers Manual*.

Errors:

| Code | Explanation |
|---|---|
| 0 | Device not found in tables. |

Example: See the example under .CSISPC.

# .ELAW

**EMT 375, Code 36, Subcode 3**

The .ELAW request eliminates a virtual address window.

Macro Call:

**.ELAW area,addr**

where:

| | |
|---|---|
| **area** | is the address of a two-word EMT argument block |
| **addr** | is the address of the window definition block for the window to be eliminated |

Request Format:

| | | |
|---|---|---|
| R0 → area: | 36 | 3 |
| | addr | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 3 | An invalid window identifier was specified. |
| 17 | Inactive mode or space was specified. |

Example:
See .CRAW.

## .ELRG

**Mapping**
**EMT 375, Code 36, Subcode 1**

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 other jobs.

When memory is freed after a region is eliminated, the .ELRG programmed request concatenates contiguous areas of memory segmented in the allocation table.

Macro Call:

**.ELRG area,addr**

where:

**area** is the address of a two-word EMT argument block

**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 cannot be eliminated

Request Format:

| | | |
|---|---|---|
| R0 → area: | 36 | 1 |
| | addr | |

Errors:

| Code | Explanation |
|---|---|
| 2 | An invalid region identifier was specified. |
| 14 | Global region in use. |
| 11 | Deallocation failure. |

Example:
See .CRAW.

# .ENTER

**EMT 375, Code 2**

The .ENTER request allocates space on the specified device and creates a tentative entry in the directory with the name of the specified file. The channel number specified is associated with the file.

Macro Call:

**.ENTER area,chan,dblk,len[,seqnum]**

where:

**area** is the address of a four-word EMT argument block

**chan** is a channel number in the range 0-$376_8$

**dblk** is the address of a Radix–50 descriptor of the file to be created.

If the file name is not provided in *dblk*, it is a non-file-structured .LOOKUP which connects the channel to the entire device, starting at block 0.

**len** is the file size specification. If you don't specify a value for *len*, the value in *area* is used to specify the value for *len*.

For RT–11 structured devices, the value of this argument determines the file length allocation:

0 is either half the largest empty entry or the entire second-largest empty entry. Whichever is larger is compared to MAXBLK (RMON fixed offset 314). The smaller value is selected. Value may be expressed as: MIN(MAXBLK, MAX(LEMPTY)/2, 2ND LEMPTY).

**NOTE**

MAXBLK is the maximum size for nonspecific .ENTER requests that are patched in the monitor by changing RMON offset 314. (See example for .PVAL.)

-1 is the smaller of the largest available empty entry compared to MAXBLK ($177777_8$ blocks.) Value may be expressed as MIN(MAXBLK, LEMPTY).

m is a file of *m* blocks. Use this argument to specify the number of blocks needed.

**seqnum** is a parameter for magtape. Programming for specific devices such as magtape is discussed in detail in *RT–11 Device Handlers Manual*.

*Seqnum* describes a file sequence number. The action taken depends on whether the file name is given or is null. The sequence number can have the following values:

| | |
|---|---|
| 0 | Rewind the magtape and space forward until the file name is found or until logical end-of-tape is detected. If the file name is found, an error is generated. If the file name is not found, then enter file. If the file name is a null, a non-file-structured lookup is done (tape is rewound) |
| -1 | Space to the logical end-of-tape and enter file |
| -2 | Rewind the magtape and space forward until the file name is found or until logical end-of tape is detected. A new logical end-of-tape is implied. |
| n | Position magtape at file sequence number *n* if *n* is greater than zero and the file name is not null |

Request Format:

R0 ⟶ area:

| | |
|---|---|
| 2 | chan |
| dblk | |
| len | |
| seqnum | |

On return from this call, R0 contains the size of the area actually allocated for use. Or zero (0) for a non-RT–11 device or non-file-structured .ENTER.

#### Notes

Because a file created with an .ENTER request is not permanent until a .CLOSE request is given on that channel, the newly created file is not available to .LOOKUP, and the channel cannot be used by .SAVESTATUS requests. However, it is possible to read data that has just been written into the file by reading the channel number on which the .ENTER was issued. When the .CLOSE to the channel is given, any existing permanent unprotected 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. The .CLOSZ request can be used to truncate the file wherever you wish.

Each program can have up to 255 files open on the system at any time. If required, all 255 can be opened for output with the .ENTER function.

When a file-structured .ENTER request is made, the device handler must be in memory. Thus, a .FETCH should normally be executed before an .ENTER can be done.

When using the zero-length feature of .ENTER, keep 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 diskettes that have a relatively small capacity. For example, transferring a 200-block file to a diskette, on which the largest available empty space is 300 blocks, does not work with a zero-length .ENTER. Since the .ENTER allocates half the largest space, only 150 blocks are really allocated and an output error occurs during the transfer. When transferring from A to B, with the length of A unknown, do a .LOOKUP first. This request returns the length so that value can be used to do a fixed-length .ENTER. The .ENTER request generates hard errors when problems are encountered during directory operations. These errors can be detected after the operation with the .SERR request. Hard errors are passed to the program when .SERR has been issued prior to .ENTER.

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel is not available. |
| 1 | In a fixed-length request, no space greater than or equal to *m* was found; or the device or the directory was found to be full. |
| 2 | Nonshareable device is already in use by another program. |
| 3 | A file by that name already exists and is protected. A new file was not opened. |
| 4 | File sequence number was not found. |
| 5 | File sequence number is invalid or file name is null. |
| 6 | Request is issued to a nonexistent or otherwise invalid special-directory device unit. The handler determines the validity of the device unit. |

Example:

```
        .TITLE  EENTER;2
;+
; .ENTER - This is an example in the use of the .ENTER request.
; The example makes a copy of the file 'PIP.SAV' on device DK:
;-
        .MCALL          .LOOKUP,.ENTER,.WRITW,.READW,.CLOSE
        .MCALL          .PRINT,.EXIT
$ERRBY  =:      52                              ;(.SYCDF) EMT error byte
$USRRB  =:      53                              ;(.SYCDF) user error byte
SUCCS$  =:      001                             ;(.UEBDF) success bit
FATAL$  =:      004                             ;(.UEBDF) error bit
```

```
.ENABL  LSB
START:  .LOOKUP         #AREA,#0,#PIP           ;Lookup file SY:PIP.SAV
        BCS             4$                      ;Branch if not there!
        MOV             R0,R3                   ;Copy size of file to R3
        .ENTER          #AREA,#1,#TFILE,R3      ;Enter a new file of same size
        BCS             5$                      ;Branch if failed
        CLR             BLK                     ;Initialize block # to zero
1$:     .READW          #AREA,#0,#BUFFR,#256.,BLK ;Read a block
        BCC             2$                      ;Branch if successful
        TSTB            @#$ERRBY                ;Was error EOF?
        BEQ             3$                      ;Branch if yes
        MOV             #RERR,R0                ;Hard read error message to R0
        BR              7$                      ;Branch to print message
2$:     .WRITW          #AREA,#1,#BUFFR,#256.,BLK ;Write a block
        INC             BLK                     ;*C*Bump block #
        BCC             1$                      ;Branch if write was ok
        MOV             #WERR,R0                ;R0 => Write error message
        BR              7$                      ;Branch to print message
3$:     .CLOSE          #1                      ;Make new file permanent
        MOV             #DONE,R0                ;R0 => Done message
        BR              6$                      ;Branch to print message
4$:     MOV             #NOFIL,R0               ;R0 => File not found message
        BR              7$                      ;Branch to print it
5$:     MOV             #NOENT,R0               ;R0 => Enter Failed message
        BR              7$                      ;Branch to print message
6$:     BISB            #SUCCS$,@#$USRRB        ;Indicate success
        BR              8$
7$:     BISB            #FATAL$,@#$USRRB        ;Indicate error
8$:     .PRINT                                  ;Print message on console
                                                ; terminal
        .EXIT                                   ;the exit program

AREA:   .WORD           0                       ;EMT Argument block
BLK:    .WORD           0,0,0,0                 ;
BUFFR:  .BLKW           256.                    ;I/O Buffer
PIP:    .RAD50          /SY/                    ;File descriptors...
        .RAD50          /PIP   /
        .RAD50          /SAV/
TFILE:  .RAD50          /DK/
        .RAD50          /PIP   /
        .RAD50          /TMP/
NOFIL:  .ASCIZ          /?EENTER-F-File not found/
NOENT:  .ASCIZ          /?EENTER-F-.ENTER Failed/
WERR:   .ASCIZ          /?EENTER-F-Write Error/
RERR:   .ASCIZ          /?EENTER-F-Read Error/
DONE:   .ASCIZ          /!EENTER-I-PIP Copy Complete/
        .END            START
```

# .EXIT

**EMT 350**

Macro Call:

**.EXIT**

The .EXIT request causes the user program to terminate. When used from a background job, .EXIT causes KMON to run in the background area, prior to running KMON; all outstanding mark-time requests are canceled; and 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 and SET EXIT SWAP is in effect, the user program 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 SET EXIT NOSWAP is in effect, the user program is overwritten when a .EXIT is done.

  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. See .HRESET.

The .EXIT request enables a user program to pass command lines to KMON in the chain information area (locations $500\text{-}777_8$) for execution after the job exits. This is performed under the following conditions:

- The word (not byte) location 510 must contain the total number of bytes of command lines to be passed to KMON.
- 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:

```
        .TITLE  EEXIT1.MAC

XIT.NU  =:      510             ;(.XITDF) char count
XIT.AS  =:      512             ;(.XITDF) .Asciz command(s)

.=XIT.NU
        .WORD B-A
.=XIT.AS
A:      .ASCIZ /COPY A.MAC B.MAC/
        .ASCIZ /DELETE A.MAC/
B:
```

  The user program must set SPXIT$ or CHNIF$ in the Job Status Word before doing an .EXIT, which must be issued with R0 = 0.

When the .EXIT request is used to pass command lines to KMON, the following restrictions are in effect:

- If CHNIF$ of the JSW is set and if the feature is used by a program that is invoked through an indirect file, the indirect file context is aborted before executing the supplied command lines. Any unexecuted lines in the indirect file are never executed.

- If SPXIT$ of the JSW is set and the feature is used by a program invoked through an indirect file, the indirect file context is preserved across the .EXIT request.
- An indirect file can be invoked, using the steps described above, 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. An indirect file must be the last item on a KMON command line.

The .EXIT request 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.

An attempt to use a .EXIT from a completion routine aborts the running job.

**NOTE**

You can prevent data passed to KMON from being destroyed during the .EXIT request by not allowing the User stack to overwrite this data area. If User passes command lines to KMON, reset the stack pointer to $1000_8$ or above before exiting.

Errors:
None.

Example:

```
        .TITLE  EEXIT2;2
;+
;.EXIT - This is an example in the use of the .EXIT request.
; The example demonstrates how a command line may be passed to
; Keyboard Monitor after job execution is stopped.
;-
        .MCALL .EXIT
$JSW    =: 44                           ;(.SYCDF)JSW location
SPXIT$  =: 000040                       ;(.JSWDF)Special command
CHNIF$  =: 004000                       ;(.JSWDF)Std command
XIT.NU  =: 510                          ;(.XITDF)command length word
XIT.AS  =: 512                          ;(.XITDF)command ASCIZ

START:  MOV             #XIT.NU,R0      ;R0 => Communication area
        MOV             #CMDSTR,R1      ;R1 => Command string
        MOV             #START,SP       ;Make sure that the stack is
                                        ;not in the communication area...
10$:    MOVB            (R1)+,(R0)+     ;Copy command string
        CMP             R1,#CMDEND      ;Done?
        BLO             10$             ;Branch if not
        BIS             #SPXIT$,@#$JSW  ;Set the "chain" bit to alert KMON
                                        ;that there’s a command in the
                                        ;communication area
        CLR             R0              ;R0 must be zero !
        .EXIT                           ;Exit the program

CMDSTR: .WORD           CMDEND-CMDSTR-2
        .ASCIZ          "$Directory SRC:EEXIT2.MAC"
CMDEND:
        .END            START
```

# .FETCH/.RELEAS

**EMT 343**

The .FETCH request loads device handlers into memory from the system device. FETCHing with mapped monitors is dependent upon a SYSGEN feature. FETCH support is the default.

Macro Call:

**.FETCH addr,dnam**

where:

| | |
|---|---|
| **addr** | is the starting address at which 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 contains the same value that was initially specified in the argument *addr*. If less than $400_8$, a handler .RELEAS is being done. .RELEAS does not remove a handler from memory that was LOADed. An UNLOAD must be done. After a .RELEAS, you must issue a .FETCH to use the device again. .FETCH issued from a foreground or system job will succeed, provided the specified handler is currently in memory.

If the program is run under VBGEXE, the space for the fetched handler is allocated from memory controlled by VBGEXE and the address passed by the user program is not used.

Several requests require a device handler to be in memory for successful operation. These include:

| | | |
|---|---|---|
| .CLOSE | .FPROT | .RENAME |
| .CLOSZ | .GF* | .SF* |
| .DELETE | .LOOKUP | .SPFUN |
| .ENTER | .READ* | .WRIT* |

**NOTE**

I/O operations cannot be executed on devices unless the handler for that device is in memory.

Errors:

| Code | Explanation |
|---|---|
| 0 | The device name specified is not installed and there is no logical name that matches the name and the *catch-all has not been assigned. |

Example:
See example for .CSISPC.

The .RELEAS request notifies the monitor that a fetched device handler is no longer needed. The .RELEAS is ignored if the handler is:

- Permanently resident (SY:, TT:, MQ:, UB:, PI:)
- Not in memory
- Loaded

.RELEAS of a valid device name from a foreground or system job is ignored.

Macro Call:

**.RELEAS dnam**

where:

**dnam** is the address of the Radix–50 device name

Errors:

| Code | Explanation |
|---|---|
| 0 | Device name is invalid. |

Example:

```
        .TITLE  RELEAS.MAC

;In this example, the Null handler (NL) is loaded into memory,
;used, then released. If NL is LOADed the handler is
;resident, and .FETCH will return HSPACE in R0.

        .MCALL  .FETCH,.RELEAS,.EXIT,.PRINT

START:  .FETCH  #HSPACE,#NLNAME         ;Load NL handler
        BCS     FERR                    ;Not available

                                        ; Use handler

        .RELEAS #NLNAME                 ;Mark NL no longer in
                                        ;memory
        .EXIT

FERR:   .PRINT  #NONL                   ;NL not available
        .EXIT

NLNAME: .RAD50  /NL /                   ;Name for NL handler
NONL:   .ASCIZ  /?ERELEA-F-NL handler not available/
        .EVEN
HSPACE:                                 ;Beginning of handler
                                        ;area
        .END    START
```

# .FORK

**Macro Expansion**

The .FORK call 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. .FORK issues a subroutine call to the monitor and does not use an EMT instruction request.

Macro Call:

**.FORK fkblk**

where:

**fkblk** is a four-word block of memory allocated within the driver

Errors:
None.

The .FORK macro expands as follows:

```
 .TITLE  EFORK1.MAC
        .FORK   fkblk

        JSR     R5,@$FKPTR
        .WORD   fkblk-.
```

The .FORK call must be preceded by an .INTEN call. Your program must not have left any information on the stack between the .INTEN and the .FORK calls. The contents of registers R4 and R5 are preserved through the call and, on return, registers R0 through R3 are available for use.

If you are using a .FORK call from a device handler, it is assumed that you used .DREND provided for handlers. The .DREND macro allocates a word labeled $FKPTR. This word is filled in when the handler is placed in memory with the address of the monitor fork routine.

If you want to use the .FORK macro in an in-line interrupt service routine rather than in a device handler, you must set up $FKPTR. The recommended way to do this is as follows:

```
        .TITLE  EFORK2.MAC

$FORK   =:      402             ;(.FIXDF)Monitor offset containing
                                ;offset to fork processor
        .GVAL   #AREA,#$FORK    ;Return value in R0
        MOV     R0,$FKPTR       ;Save address of the
                                ;fork processor
         ...

INTIN:                          ;Interrupt entry
        .INTEN  4               ;Declare interrupt and drop to PR4
        ...                     ;Process quick stuff
        .FORK   FORKBK          ;do a fork
        ...                     ;Process slow stuff
        RETURN                  ;and return from interrupt
```

```
AREA:   .BLKW   2               ;EMT request block
$FKPTR: .BLKW   1               ;Address of FORK routine
FORKBK: .BLKW   4               ;Fork block
```

Once the pointer is set up, use the macro in the usual way as follows:

```
        .TITLE  EFORK3.MAC

        .FORK   fkblk
```

This method permits you to preserve both R4 and R5 across the fork.

The fork request is linked into a fork queue and serviced on a first-in first-out basis. On return to the handler or interrupt service routine following the call, the interrupt has been dismissed and the processor 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. It also should not be used with devices that have continuous interrupts that cannot be disabled. The *RT–11 Device Handlers Manual* gives additional information on the .FORK request.

Example:
Refer to the example following the description of .DRAST.

# .FPROT

**EMT 375, Code 43**

The .FPROT programmed request sets or removes file protection on individual RT–11 files. A file marked as protected cannot be deleted by .CLOSE, .DELETE, .ENTER, or .RENAME requests. However, the contents of a protected file are not protected against modification. Use .SFSTAT to set E.READ for more protection. For example, a .LOOKUP of a protected file followed by a .WRITE to the file is permitted. To protect a file from being written to, set E.READ (bit 14) in the file's directory entry status word. See the example for the .SFSTAT request.

Protection is enabled by setting E.PROT (bit 15) of a file's directory entry status word.

Macro Call:

**.FPROT area, chan, dblk[,prot]**

where:

| | |
|---|---|
| **area** | is the address of a three-word EMT argument block |
| **chan** | is a channel number in the range 0-$376_8$ |
| **dblk** | is the address of a four-word block containing the filespec in Radix–50 of the file |
| **prot** | = #1—(or omitted) to protect the file from deletion |
| | = #0—to remove protection so that the file can be deleted |

Request Format:

| | | |
|---|---|---|
| R0 → area: | 43 | chan |
| | dblk | |
| | prot | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Channel is not available. |
| 1 | File not found or not a file-structured device.<br>To find out what condition returned the error code, issue a .DSTAT request to determine if a device is file structured. |
| 2 | Invalid operation. |
| 3 | Invalid value for PROT. |

.FPROT returns the previous file status word in R0:

- If the high bit in R0 is off, the file was not previously protected.

- If the high bit in R0 is on, the file was previously protected.

Example:

```
        .TITLE  EFPROT;2

;.FPROT, .SFDAT example.
;This is an example of the use of the .FPROT and .SFDAT
;programmed requests. It uses the "special" mode of the CSI to
;get an input filespec from the console terminal. .DSTATUS is
;used to determine if the device handler is loaded. If not, a
;.FETCH request is used to load the handler into memory. Finally,
;the file is marked as protected using the .FPROT request and
;the file date is changed to the current system date using the
;.SFDAT request.
;
        .MCALL .FPROT, .FETCH, .CSISPC, .DSTATUS, .SFDAT, .PRINT, .EXIT

$USRRB  =:      53                      ;(.SYCDF) user error byte
SUCCS$  =:      001                     ;(.UEBDF) success bit
ERROR$  =:      004                     ;(.UEBDF) error bit

START:  MOV     SP,R5                   ;Save SP, since CSI changes it
        .CSISPC #OUTSP,#DEFEXT          ;Use CSI to get input filespec
        MOV     R5,SP                   ;Restore SP
        .DSTAT  #STAT,#INSPEC           ;Check the device
        TST     STAT+4                  ;to see if the handler is
                                        ;resident
        BNE     10$                     ;Branch if it is
        .FETCH  #HANLOD,#INSPEC         ;Otherwise, load that handler
        BCC     10$                     ;ok
        .PRINT  #LOFAIL                 ;Otherwise, print load error
                                        ;message
        BR      30$                     ;and try again
10$:    .FPROT  #EMTBLK,#0,#INSPEC,#1   ;Mark file as protected
        BCC     20$                     ;and branch if okay
        .PRINT  #PRFAIL                 ;Otherwise, print protect
                                        ;error message
        BR      30$                     ;and try again
20$:    .SFDAT  #EMTBLK,#0,#INSPEC,#0   ;Finally, set current date
                                        ;A date of 0 means "use
                                        ;current system date"
        BCC     40$                     ;Branch if everything is okay
        .PRINT  #SDFAIL                 ;Otherwise, print date error
                                        ;message
30$:    BISB    #ERROR$,@#$USRRB        ;Indicate error
        BR      START                   ;and try again
40$:    .EXIT                           ;Everything okay - exit to KMON

EMTBLK: .BLKW   4                       ;The EMT argument block is
                                        ;built here
DEFEXT: .WORD   0,0,0,0                 ;No default extensions
STAT:   .BLKW   4                       ;Block for .DSTATUS to use
LOFAIL: .ASCIZ  /?EFPROT-F-.FETCH request failed/
PRFAIL: .ASCIZ  /?EFPROT-F-.FPROT request failed/
SDFAIL: .ASCIZ  /?EFPROT-E-.SFDAT request failed/
        .EVEN
OUTSP:  .BLKW   5*3                     ;Output specs go here
INSPEC: .BLKW   4*6                     ;Input specs go here
HANLOD: .BLKW   1                       ;Handlers begin loading here
                                        ;(if necessary)
        .END            START
```

# .GCMAP

See .CMAP/.CMPDF/.GCMAP.

# .GFDAT

**EMT 375, Code 44**

The .GFDAT programmed request returns in R0 the creation date from a file's directory entry (E.DATE word). .GFDAT is not supported for the distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.GFDAT area,chan,dblk**

where:

**area** is the address of a 4-word EMT argument block

**chan** is a channel number in the range of 0 to 376(octal)

**dblk** is the address of a 4-word block containing a device and file specification in Radix–50; the file specification for which you want to return the creation date.

Request Format:

| | | | |
|---|---|---|---|
| R0 → area: | 44 | chan | |
| | dblk | | |
| | 0 | | (Reserved) |
| | 14 | 0 | (High byte is E.DATE offset into directory entry) |

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel is not available |
| 1 | File not found, or not a file-structured device.<br>If it is necessary to determine what condition returned the error code, issue a .DSTAT request to determine if a device is file structured |
| 2 | Invalid operation |
| 3 | Invalid EMT request block |

Example:

```
.TITLE  EGFDAT.MAC

;       This program displays the creation date of SY:SWAP.SYS

.MCALL  .GFDAT  .PRINT  .EXIT

START::
        .GFDAT  #AREA,#CHAN,#DBLK ;Get the file date
        MOV     R0,DATE         ;*C* save result w/o changing carry
        BCS     ERROR           ;Failure
        MOV     #PDATE4,R5      ;Pass parameter list
        CALL    DATE4Y          ;Make the date displayable
        CLRB    DDATE+11.       ;Terminate the string
        .PRINT  #DDATE          ;And display it
        .EXIT

ERROR:
        .PRINT  #ERRMSG         ;GFDAT failed
        .EXIT

CHAN     =:     0               ;Available channel
DATE:   .BLKW   1               ;File date
DBLK:   .RAD50  "SY SWAP  SYS"  ;Dblock specifying file
AREA:   .BLKW   4.              ;EMT area

PDATE4: .WORD   2
        .WORD   DDATE           ;Addr of ASCII format date
        .WORD   DATE            ;Addr of RT-11 format binary date

DDATE:  .BLKB   12.             ;Displayable date
ERRMSG: .ASCIZ  "?EGFDAT-F-.GFDAT Failed"

             .END    START
```

# .GFINF

**EMT 375, Code 44**

The .GFINF programmed request returns in R0 the word contents of the directory entry offset you specify from a file's directory entry. .GFINF is not supported for the distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.GFINF area,chan,dblk,offse**

where:

- **area** is the address of a 4-word EMT argument block
- **chan** is a channel number in the range of 0 to 376(octal)
- **dblk** is the address of a 4-word block containing a device and file specification in Radix–50; the file specification for which you want to return directory entry information.
- **offse** is the octal byte offset for the directory entry word you want. The offset must be even. For example, specifying offset 12 returns the contents of E.USED in R0

Request Format:

| | | | |
|---|---|---|---|
| R0 → area: | 44 | chan | |
| | dblk | | |
| | 0 | | (Reserved) |
| | offse | 0 | (High byte is offset into directory entry) |

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel is not available |
| 1 | File not found, or not a file-structured device.<br>If it is necessary to determine what condition returned the error code, issue a .DSTAT request to determine if a device is file structured |
| 2 | Invalid operation |
| 3 | Invalid offset value |

Example:

```
        .TITLE  EGFINF;2
;
; This program displays the contents of offset 12(8)
; of the selected directory entry.  It is displayed
; as a time based on the number of seconds since
; midnight divided by 3.
;
.MCALL  .CSISPC .FETCH  .RELEAS .PRINT  .EXIT
.MCALL  .GFINF

.GLOBL  $DIVTK          ;divide by number of ticks in a second
.GLOBL  $DIVNN          ;divide by value in R4

START:: MOV     SP,R5           ;Save SP, since CSISPC changes it
        .CSISPC #OUTSPC,#DEFEXT ;Get a file name
        MOV     R5,SP           ;*C* Restore it (leave carry alone)
        BCS     CSIERR          ;Error
        .FETCH  LIMIT+2,#INSPC  ;Fetch the handler
        BCS     FETERR          ;Error
        .GFINF  #AREA,#0,#INSPC,#E.USED ;Get directory entry value
        BCS     GFIERR          ;Error
        MOV     R0,R1           ;build 32-bit value
        CLR     R0              ;out of 16 bit positive value
        MOV     #60./3.,R4      ;get number of seconds
        CALL    $DIVNN          ;R0..R1 quotient, R3 remainder
        MOV     R3,-(SP)        ;save number of seconds
        ASL     R3              ; *2
        ADD     R3,@SP          ; *3
        CALL    $DIV60          ;get number of minutes
        MOV     R3,-(SP)        ;R3 is number of minutes
        MOV     R1,-(SP)        ;R1 is number of hours
        MOV     #OUTPUT,R5      ;point to output area
        MOV     #3,R4           ;number of values to process
 10$:   MOV     (SP)+,R1        ;get the value
        ASL     R1              ;make into word index
        MOV     #CVT,R0         ;point to Ascii
        ADD     R1,R0           ; ...
        MOVB    (R0)+,(R5)+     ;copy into buffer
        MOVB    (R0)+,(R5)+     ; ...
        TSTB    (R5)+           ;skip ":"
        SOB     R4,10$          ;do hours, minutes, seconds

        .PRINT  #OUTPUT         ;display time
        .RELEASE #INSPC         ;Dismiss handler
        BR      START           ;And again

CSIERR: MOV     #CSIMSG,R0
        BR      DONE

FETERR: MOV     #FETMSG,R0
        BR      DONE

GFIERR: MOV     #GFIMSG,R0
DONE:   .PRINT  R0
        .EXIT
```

```
E.USED=:12                          ;Unused directory word
OUTSPC: .BLKW   3*5                 ;CSISPC return area
INSPC:  .BLKW   6*4                 ;  "      "     "
DEFEXT: .RAD50  "             "     ;Default extensions (none)
LIMIT:  .LIMIT                      ;Memory usage (macro directive)
AREA:   .BLKW   10.                 ;EMT request block area

CVT:
        .ASCII  "00010203040506070809"
        .ASCII  "10111213141516171819"
        .ASCII  "20212223242526272829"
        .ASCII  "30313233343536373839"
        .ASCII  "40414243444546474849"
        .ASCII  "50515253545556575859"

OUTPUT: .ASCIZ  "XX:XX:XX"
CSIMSG: .ASCIZ  "?EGFINF-E-CSI error"
FETMSG: .ASCIZ  "?EGFINF-E-Fetch error"
GFIMSG: .ASCIZ  "?EGFINF-E-GFINF error"

        .END    START
```

# .GFSTA

**EMT 375, Code 44**

The .GFSTA programmed request returns in R0 the status information from a file's directory entry (E.STAT word). .GFSTA is not supported for the distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.GFSTA area,chan,dblk**

where:

**area** is the address of a 4-word EMT argument block

**chan** is a channel number in the range of 0 to 376(octal)

**dblk** is the address of a 4-word block containing a device and file specification in Radix–50; the file specification for which you want to return the file status word.

Request Format:

| | | | |
|---|---|---|---|
| R0 → area: | 44 | chan | |
| | dblk | | |
| | 0 | | (Reserved) |
| | 0 | 0 | (High byte is E.STAT offset into directory entry) |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Channel is not available |
| 1 | File not found, or not a file-structured device.<br>If it is necessary to determine what condition returned the error code, issue a .DSTAT request to determine if a device is file structured |
| 2 | Invalid operation |
| 3 | Invalid offset value |

Example:

```
.TITLE  EGFSTA  -- sample program for .GFSTA
.MCALL  .CSISPC .SRESET .FETCH  .GFSTA  .PRINT  .EXIT
.ENABL  LSB
START::
10$:
        .SRESET                 ;Dismiss handlers
        .CSISPC #OUTSPC,#DEFEXT ;Get a file name
        BCS     CSIERR          ;error
        .FETCH  LIMIT+2,#DBLK   ;Fetch the handler
        MOV     R0,LIMIT+2      ;*C*new available pointer
        BCS     FETERR          ;error
        .GFSTAT #AREA,#0,#DBLK  ;Get the file's status word
        BCS     GFSERR          ;error
        MOV     R0,R5           ;Save the return value
        MOV     #1,R1           ;start sliding bit
        MOV     #BITNAM,R2      ;point to bit name array
20$:    BIT     R1,R5           ;is the bit set in the status?
        BEQ     30$             ;no
        .PRINT  R2              ;yes, display name
30$:
        ASL     R1              ;next bit
        BEQ     10$             ;tried all bits
        ADD     #NAMLEN,R2      ;next name
        BR      20$             ;test all bits

CSIERR: .PRINT  #CSIMSG         ;CSI error
        .EXIT

FETERR: .PRINT  #FETMSG         ;Fetch error
        .EXIT

GFSERR: .PRINT  #GFSMSG         ;GFSTA error
        .EXIT

OUTSPC: .BLKW   3*5             ;output file specs (unused)
DBLK:   .BLKW   4               ;first input file spec
        .BLKW   5*4             ;rest of file specs (unused)

DEFEXT: .RAD50  "            "  ;default extensions (none)

LIMIT:  .LIMIT

AREA:   .BLKW   4

BITNAM:
        .ASCIZ  "000001?"
NAMLEN  =:      .-BITNAM
        .ASCIZ  "000002?"
        .ASCIZ  "000004?"
        .ASCIZ  "000010?"
        .ASCIZ  "000020?"
        .ASCIZ  "000040?"
        .ASCIZ  "000100?"
        .ASCIZ  "000200?"
        .ASCIZ  "E.TENT?"
```

```
        .ASCIZ  "E.MPTY?"
        .ASCIZ  "E.PERM "
        .ASCIZ  "E.EOS? "
        .ASCIZ  "010000?"
        .ASCIZ  "020000?"
        .ASCIZ  "E.READ "
        .ASCIZ  "E.PROT "
CSIMSG: .ASCIZ  "?EGFSTA-E-CSISPC error"
FETMSG: .ASCIZ  "?EGFSTA-E-Fetch error"
GFSMSG: .ASCIZ  "?EGFSTA-E-GFSTAT error"
        .END    START
```

# .GMCX

**EMT 375, Code 36, Subcode 6**

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, entire windows can be changed by modifying certain fields of the window definition block.

Macro Call:

**.GMCX area[,addr]**

where:

**area** is the address of a two-word EMT argument block

**addr** is the address of the window definition block where the specified window's status is returned

Request Format:

| | | |
|---|---|---|
| R0 → area: | 36 | 6 |
| | addr | |

The .GMCX request modifies the following fields of the window definition block:

**W.NAPR** base page address register of the window

**W.NBAS** window virtual address

**W.NSIZ** window size in 32-word blocks

**W.RID** region identifier

If the window whose status is requested is mapped to a region, the .GMCX request loads the following additional fields in the window definition block; otherwise, these locations are zeroed:

**W.NOFF** offset value into the region

**W.NLEN** length of the mapped window

**W.NSTS** state of the WS.MAP bit is set to 1 in the window status word

Errors:

| Code | Explanation |
|---|---|
| 3 | An illegal window identifier was specified |
| 17 | Inactive space or mode was specified |

Example:
Refer to the *RT–11 System Internals Manual*.

# .GTIM

**EMT 375, Code 21**

.GTIM accesses the current time of day. The time is returned in two words and 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 the address of 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 is returned in the second word. Your program must perform 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 (offset $300_8$ relative to the start of the resident monitor). In the FB monitor, the time of day is automatically reset after 24:00, when a .GTIM request is done and the date is changed if necessary. In the SB monitor, the time of day is not reset unless you have selected the SB timer support option during system generation process. The month is not automatically updated in either monitor. (Proper month and year rollover is an option enabled during the system generation process.)

The default clock rate is 60 Hz, 60 ticks per second. Use SET CLOCK 50 or consult the *RT–11 System Generation Guide* if conversion to a 50-Hz rate is necessary.

**Note**

Several SYSLIB routines that perform time conversion are CVTTIM, TIMASC and TIME.

Errors:
None.

Example:

```
        .TITLE  EGTIM;2

;+
; .GTIM - This is an example in the use of the .GTIM request.
; This example includes a subroutine that can be assembled separately
; and linked with a user program.
;-
        .MCALL  .DEBUG  .DPRINT .EXIT
```

```
.ENABL  LSB
        .DEBUG  SWITCH=ON,VALUE=YES

START::
        CALL    TIME    ;get current time in binary
        MOVB    R4,R0   ;move hours to R0
        .DPRINT ^"!EGTIM-I-Hours   ",R0,DEC
        SWAB    R4      ;get minutes into low byte
        MOVB    R4,R0   ;move minutes to R0
        .DPRINT ^"!EGTIM-I-Minutes ",R0,DEC
        MOVB    R5,R0   ;move seconds to R0
        .DPRINT ^"!EGTIM-I-Seconds ",R0,DEC
        SWAB    R5      ;get ticks into low byte
        MOVB    R5,R0   ;move ticks to R0
        .DPRINT ^"!EGTIM-I-Ticks   ",R0,DEC
        .EXIT
.PAGE
;+
; CALLING SEQUENCE:  CALL TIME
;
; INPUT:                 none
;
; OUTPUT:                R4 = Minutes in hi byte / hours in lo byte
;                        R5 = Ticks in hi byte / seconds in lo byte
;                             (in that order for ease of removal !)
;
; ERRORS:                none possible
;
; NOTE: This example calls SYSLIB functions '$DIVTK' & '$DIV60'
;-

        .GLOBL  $DIVTK,$DIV60
        .MCALL  .GTIM

TIME::  MOV     #TICKS,R1       ;R1 points to where to put time
        .GTIM   #AREA,R1        ;Get ticks since midnight via .GTIM
        MOV     (R1)+,R0        ;R0 = lo order time
        MOV     @R1,R1          ;R1 = hi order time
        CALL    $DIVTK          ;Call SYSLIB 32 bit divide by clk freq
        MOV     R3,R5           ;Save ticks
        SWAB    R5              ;Put them in hi byte
        CALL    $DIV60          ;Call SYSLIB divide by 60. routine
        BISB    R3,R5           ;Put seconds in lo byte
        CALL    $DIV60          ;Divide by 60. once again
        MOV     R3,R4           ;Put minutes in R4
        SWAB    R4              ;Move them to hi byte
        BISB    R1,R4           ;Put hours in lo byte
        RETURN                  ;and return

AREA:   .BLKW   2               ;EMT argument area
TICKS:  .BLKW   2               ;Ticks since midnight returned here

        .END    START
```

# .GTJB

**EMT 375, Code 20, Subcode 1**

The .GTJB request returns information about a job in the system.

Macro Call:

**.GTJB area,addr[,jobblk]**

where:

**area** is the address of a three-word EMT argument block

**addr** is the address of an eight-word or twelve-word block into which the parameters are passed.

**jobblk** is a pointer to a three-word ASCII logical job name for which data are being requested. The values returned are:

| Word Offset | Contents |
|---|---|
| **Word 0** J.BNUM | Job Number:<br>• System Job Monitors:<br>Background job is 0<br>System jobs are 2, 4, 6, 10, 12, 14<br>Foreground job is 16<br>• Non-System Monitors:<br>Background job is 0<br>Foreground job is 2<br>• Single Job Monitor:<br>Job number is 0 |
| **Word 1** J.BHLM | High-memory limit of job partition (highest location available to a job in low memory if the job executes a privileged .SETTOP -2 request) |
| **Word 2** J.BLLM | Low-memory limit of job partition (first location) |
| **Word 3** J.BCHN | Pointer to I/O channel space |
| **Word 4** J.BIMP | Address of job's impure area in monitors |
| **Word 5** J.BLUN | Low byte: unit number of job's console terminal (used only with multiterminal option; 0 when multiterminal feature is not used)<br>High byte: reserved for future use |

| | |
|---|---|
| **Word 6**<br>J.BVHI | Virtual high limit for a job created with the linker /V option (XM only; 0 when not running under the mapped monitor or if /V option is not used) |
| **Word 7–8** | Reserved for future use |
| **Word 9–11**<br>J.BLNM | ASCII logical job name (system job monitors only; contains nulls for non-system job monitors.) |

Offset word 3 of *addr*, which describes where the I/O channel words begin, normally indicates an address within the job's impure area. However, when a .CDFN is executed, the start of the I/O channel area changes to the user-specified area.

If the *jobblk* argument to the .GTJB request is:

- Between 0 and 16, it is interpreted as a job number.
- 'ME' or equals -1, information about the current job is returned.
- Omitted or equals -3 (a V03B-compatible parameter block), only eight words of information (corresponding to words 1-8 of *addr*) are returned.

In an environment without the system job feature, you can get another job's status only by specifying its job number (0 or 2).

Request Format:

| | | |
|---|---|---|
| R0 → area: | 20 | 1 |
| | addr | |
| | jobblk | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | No such job currently running. |

Example:

```
.TITLE  EGTJB.MAC

;+
; .GTJB - This is an example of the .GTJB request. The
; example issues the request to determine if there is a loaded
; Foreground Job in the system. This program will execute properly
; with either a normal FB monitor or an FB monitor that includes
; System Job support.
;-
        .MCALL  .GVAL, .GTJB, .PRINT, .EXIT

        $SYSGEN=: 372                   ;(.FIXDF) Fixed offset to SYSGEN word
        STASK$=:  40000                 ;(.SGNDF)System Job option bit

START:  MOV     #2,R1                   ;Assume FG job number = 2
        .GVAL   #AREA,#$SYSGEN          ;Get SYSGEN option word
        BIT     #STASK$,R0              ;System job monitor?
        BEQ     1$                      ;Branch if not
        MOV     #16,R1                  ;If so, FG job number = 16
1$:     .GTJB   #AREA,#JOBARG,R1        ;Find out if FG loaded
        BCS     2$                      ;Branch if no active FG job
        .PRINT  #FGLOAD                 ;Announce that FG job is loaded
        .EXIT                           ;and exit from program.
2$:     .PRINT  #NOFG                   ;Announce that there's no FG job
        .EXIT                           ;and exit from program.

AREA:   .BLKW   3                       ;EMT Argument block
JOBARG: .BLKW   12.                     ;Job parameters passed back here

FGLOAD: .ASCIZ  "!EGTJB-I-FG Loaded"    ;FG loaded message
NOFG:   .ASCIZ  "?EGTJB-W-No FG job"    ;No FG message

        .END    START
```

## .GTLIN

**EMT 345**

The .GTLIN request collects a line of input from the terminal or an indirect command file, if one is active.

Macro Call:

**.GTLIN linbuf[,prompt][,type]**

where:

**linbuf** is the address of the buffer to receive the input line. This area must be at least 81 bytes in length. The input line is stored in this area and is terminated with a zero byte

**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. Usually, the prompt string ends with an octal 200 byte to suppress printing the carriage return/line feed at the end of the prompt

**type** is an optional argument which, if specified, forces .GTLIN to take its input from the terminal rather than from an indirect file.

**Notes**

In the following discussion, the term *indirect* refers to either command or control files. Otherwise, descriptions pertain specifically to command files or to control files, but not both.

IND control files let you provide a partial line of input to a program and to pass a command to the program on the same line in which you invoke the program. KMON command files, on the other hand, allow you to input multiple lines of input to a program .GTLIN, like .CSIGEN and .CSISPC, requires the USR, but does no format checking on the input line. Because the .GTLIN command is implemented in the USR, the CSI will generate an error message if you attempt to enter more than 80 characters to a .GTLIN request. Normally, .GTLIN collects a line of input from the terminal and returns it in the buffer you specify. However, if an indirect file is active, .GTLIN collects the line of input from the command file as though it were coming from the terminal.

When GTLIN$ of the Job Status Word is set and your program encounters a CTRL/C in a command file, the .GTLIN request collects subsequent lines from the terminal. If you then clear bit 3 of the Job Status Word, the next line collected by the .GTLIN request is the CTRL/C in the indirect command file, which causes the program to abort. Any additional input will come from the command file, if there are any more lines in it. When TTLC$ of the Job Status Word is set, the .GTLIN request passes lowercase letters.

An optional *prompt* string argument (similar to the CSI asterisk) allows your program to query for input at the terminal. 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 file and the SET TT 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 canceled from the terminal by CTRL/U or multiple DELETE characters, unless the single-line editor is running.

If your program requires a nonstandard command format, such as the user identification code (UIC) specification for FILEX, you can use the .GTLIN request to accept the command string input line. .GTLIN tracks indirect command files and your program can do a pre-pass of the input line to remove the nonstandard syntax before passing the edited line to .CSIGEN or .CSISPC.

**NOTE**

.GTLIN performs a temporary implicit unlock of the USR while the line is being read from the console.

The only requests that can take their input from an indirect file are .CSIGEN, .CSISPC, and .GTLIN. The .TTYIN and .TTINR requests cannot get characters from an indirect command file. They get their input from the console terminal (or from a BATCH file if BATCH is running). The .TTYIN and .TTINR requests and the .GTLIN request with the optional *type* argument are useful for information that is dynamic in nature—for example, when all files with a .MAC file type need to be deleted or when a disk needs to be initialized. In these circumstances, the response to a system query should be collected through a .TTYIN or a .GTLIN with the type argument so that confirmation can be done interactively, even though the process may have been invoked through an indirect 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. Also, if there is no active indirect command file, .GTLIN collects an input line from the terminal.

Errors:

| Code | Explanation |
| --- | --- |
| 0 | Invalid command line (if the line is too long) |

Example:

```
        .TITLE  EGTLIN.MAC

;+
; .GTLIN - This is an example in the use of the .GTLIN request.
; The example merely accepts input and  echoes it back.
;-

        .MCALL  .GTLIN,.PRINT,.EXIT

$JSW    =:      44              ;(.SYCDF) job status word
TTLC$   =:      040000          ;(.JSWDF) enable lowercase input

.ENABL  LSB
START:  BIS     #TTLC$,@#$JSW   ;Enable LC (no effect for CCL line)
10$:    .GTLIN  #BUFF,#PROMT    ;Get a line of input from keyboard
        TSTB    BUFF            ;Nothing entered?
        BEQ     20$             ;Branch if nothing entered
        .PRINT  #BUFF           ;Echo the input back
        BR      10$             ;Go back for more
20$:    .EXIT                   ;Exit program on null input
BUFF:   .BLKW   41.             ;80 character buffer (ASCIZ for .PRINT)
PROMT:  .ASCII  /Echo>/
        .BYTE   200             ;no CRLF
        .END    START
```

# .GVAL/.PVAL

**.GVAL: EMT 375, Code 34, Subcode 0**
**.PVAL: EMT 375, Code 34, Subcode 2, 4, 6**

.GVAL and .PVAL must be used in a mapped environment to read or change any fixed offset, and should be used with other RT–11 mapped monitors. The .GVAL request returns in R0 the contents of a monitor fixed offset; the .PVAL request changes the contents of a monitor offset. The .PVAL request also returns the old contents of an offset in R0 to simplify saving and restoring an offset value.

### .GVAL

Macro Call:

**.GVAL area, offset**

where:

**area** is the address of a two-word EMT argument block

**offset** is the displacement from the beginning of the resident monitor to the word to be returned in R0

Request Format for .GVAL:

R0 → area:

| | |
|---|---|
| 34 | 0 |
| offset | |

### .PVAL

Macro Call:

**.PVAL area, offset, value[,TYPE=strg]**

where:

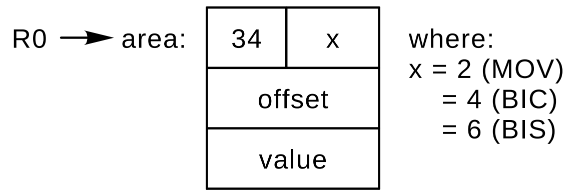
**area** is the address of a three-word EMT argument block

**offset** is the displacement from the beginning of the resident monitor to the word to be returned in R0

**value** is the new value to be placed in the fixed offset location

**TYPE=strg** Optional. Default argument is MOV or specify BIC or BIS:
MOV moves *value* to destination.
BIC uses *value* as a bit mask to clear *offset*.
BIS uses *value* to set *offset*.

Request format for .PVAL:




Errors:

| Code | Explanation |
|---|---|
| 0 | The offset requested is beyond the limits of the resident monitor. |
| 1 | Odd address. |

Example:

```
        .TITLE  EGVAL.MAC

;+
; .GVAL - This is an example of the .GVAL request. It finds out
; if the foreground job is active. Compare this example with the
; .GTJB example.
;-

        .MCALL  .GVAL, .PRINT, .EXIT

$CNFG1  =:      300     ;(.FIXDF)Offset in monitor of configuration word
FJOB$   =:      200     ;(.CF1DF)Bit in config word is on if FG active

START:  .GVAL   #AREA,#$CNFG1   ;Get monitor CONFIG word in R0
        BIT     #FJOB$,R0       ;See if FG Active bit is on
        BEQ     1$              ;Branch if not
        .PRINT  #FGLOAD         ;Announce FG is loaded
        .EXIT                   ;then exit program
1$:     .PRINT   #NOFG          ;Announce there’s no FG job
        .EXIT                   ;then exit program

AREA:   .BLKW      2                    ;EMT argument block

FGLOAD: .ASCIZ  "!EGVAL-I-FG Loaded"    ;FG loaded message
NOFG:   .ASCIZ  "?EGVAL-W-No FG job"    ;No FG message
        .EVEN

.END    START
```

```
        .TITLE  .PVAL.MAC

;+
; .PVAL - This is an example of the .PVAL request. The example
; illustrates a way of changing the default file size created
; by the .ENTER request. Compare this example with the .PEEK/.POKE
; example. .PVAL is used both to change the default file size and
; to read the old default file size, returning the old value in R0.
;-

        .MCALL  .PVAL   .ENTER  .PURGE  .EXIT
        .MCALL  .DEBUG  .DPRINT

.ENABL  LSB
        .DEBUG  SWITCH=ON

$MAXBLK =:      314             ;(.FIXDF)Monitor offset of default file size
$USRRB  =:       53             ;(.SYCDF) User Error Byte
SUCCS$  =:      001             ;(.UEBDF) success indication
FATAL$  =:      010             ;(.UEBDF) error indication

START:  MOV     #110,R1         ;Default size
        .PVAL   #EMTBLK,#$MAXBLK,R1 ;Change default file size
                                ; to 72. (110(8)) blocks
        MOV     R0,OLDSIZ       ;Save the old default
        .ENTER  #EMTBLK,#0,#DBLK,#0 ;Try non-specific size request
        BCS     ENTERR          ;Failure
        CMP     R0,R1           ;Got expected size?
        BNE     SZ1ERR          ;No
        .PURGE  #0              ;Get rid of temp file
```

```
        BIS     #1,R1           ;Adjust default size
        .PVAL   #EMTBLK,#$MAXBLK,#1,BIS ;Change default file size
                                ; to 73. (111(8)) blocks
        .ENTER  #EMTBLK,#0,#DBLK,#0 ;Try non-specific size request
        BCS     ENTERR          ;Failure
        CMP     R0,R1           ;Got expected size?
        BNE     SZ2ERR          ;No
        .PURGE  #0              ;Get rid of temp file

        BIC     #17,R1          ;Adjust default size
        .PVAL   #EMTBLK,#$MAXBLK,#17,BIC;Change default file size
                                ; to 64. (100(8)) blocks
        .ENTER  #EMTBLK,#0,#DBLK,#0 ;Try non-specific size request
        BCS     ENTERR          ;Failure
        CMP     R0,R1           ;Got expected size?
        BNE     SZ3ERR          ;No
        .PURGE  #0              ;Get rid of temp file

        .PVAL   #EMTBLK,#$MAXBLK,#OLDSIZ ;Restore old size
        .DPRINT ^"!EPVAL-I-Normal Successful Completion"
        BISB    #SUCCS$,@#$USRRB ;Indicate success
        .EXIT                   ;Done

ENTERR: .DPRINT ^"?EPVAL-F-.Enter failed"
        BR      10$
SZ1ERR: .DPRINT ^"?EPVAL-F-First file size wrong"
        BR      10$
SZ2ERR: .DPRINT ^"?EPVAL-F-Second file size wrong"
        BR      10$
SZ3ERR: .DPRINT ^"?EPVAL-F-Third file size wrong"
10$:    BISB    #FATAL$,@#$USRRB ;Indicate failure
        .EXIT

DBLK:   .RAD50  "DK XXXXXX.TMP"
EMTBLK: .BLKW   4               ;EMT argument block
OLDSIZ: .BLKW   1               ;Old default size is saved here

        .END    START
```

# .HERR/.SERR

**.HERR: EMT 374, Code 5**
**.SERR: EMT 374, Code 4**

.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 2–6).

Normally, these errors cause program termination with one of the *MON-F-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 and causing an error return to the offending EMT. 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. You must perform .PURGE or .CLOSE requests for these channels; otherwise, subsequent .LOOKUP/.ENTER requests will fail.

.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:

| | | |
|---|---|---|
| .HERR Request R0 = | 5 | 0 |
| .SERR Request R0 = | 4 | 0 |

Errors:

The .HERR and .SERR programmed requests return a code value in R0 that lets a subroutine implicitly control error condition handling:

| Code | Explanation |
|---|---|
| 0 | Previous state of error condition handling (.HERR/.SERR flag) was .HERR |
| 1 | Previous state of error condition handling (.HERR/.SERR flag) was .SERR |

A subroutine's error condition handling can be performed independently of the program that calls the subroutine. For example, the following subroutine code fragment sets the .SERR flag, executes some code, then restores the previous error condition handling status:

```
.TITLE  ESERR.MAC

.MCALL  .HERR   .SERR   .DEBUG  .DPRINT
.MCALL  .EXIT   .ENTER  .PURGE
.MACRO  ...
.ENDM

.ENABL  LSB
        .DEBUG  SWITCH=ON

START:
        .DPRINT ^"!ESERR-I-Expect message 2"
        .SERR                   ;Return errors to program
        CALL    SUBRTN
        .ENTER  #AREA,#255.     ;Try invalid chan
        .DPRINT ^"!ESERR-I-Message 2"
        .DPRINT ^"!ESERR-I-Expect MON-F message"
        .HERR                   ;Crash program on errors
        CALL    SUBRTN
        .ENTER  #AREA,#255.     ;Try invalid chan
        .DPRINT ^"?ESERR-F-Do not see this message"
        .EXIT

AREA:   .BLKW   10.

.PAGE
SUBRTN:                         ;Example subroutine
        .SERR                   ;Protect subroutine from
                                ; ?MON-F errors
        MOV    R0,SHERR         ;Save old state
        .ENTER  #AREA,#255.     ;Try invalid chan
        ...                     ;Execute some code
        TST    SHERR            ;What was the previous setting?
        BNE    10$              ;.SERR, so leave it
        .HERR                   ;Drop back to .HERR
10$:    RETURN

SHERR:  .BLKW   1               ;Saved .S/HERR status

        .END    START
```

Table 2–6 lists errors returned if soft error recovery is in effect. Traps to locations 4 and 10, floating-point exception traps, and CTRL/C aborts are not inhibited. These errors have their own recovery mechanism.

**Table 2–6: Soft Error Codes (.SERR)**

| Name | Code | Explanation |
|---|---|---|
| ER.USR | -1 | USR. Reserved. |
| ER.UNL | -2 | No device handler; this operation needs one. |
| ER.DIO | -3 | Error doing directory I/O. |
| ER.FET | -4 | .FETCH error. Either an I/O error occurred while the handler was being used, or an attempt was made to load the handler over USR or RMON. |
| ER.OVE | -5 | Reserved. Issued by overlay handler, but not passed to a program. |
| ER.DFU | -6 | No more room for files in the directory. |
| ER.IAD | -7 | Invalid address. Tried to perform a monitor operation outside the job partition. |
| ER.ICH | -10 | Invalid channel number; number is greater than actual number of channels that exist. |
| ER.EMT | -11 | Invalid EMT; an invalid function code has been decoded. |
| | 12-13 | Reserved for monitor internal use. |
| ER.DIR | -14 | Invalid directory. |
| ER.XFE | -15 | Unloaded handler under mapped system without fetch support. |
| | 16-22 | Reserved for monitor internal use. |

Example:

```
        .TITLE  EHERR1;2
;+
; .HERR / .SERR - This is an example in the use of the .HERR & .SERR
; requests. Under .SERR fatal errors will cause a return to the user
; program for processing and printing of an appropriate error message.
;-
        .MCALL  .HERR,.SERR,.LOOKUP,.PURGE
        .MCALL  .EXIT,.PRINT,.CSISPC

START:  .SERR                           ;Let program handle fatal errors
        MOV     SP,R5                   ;Save SP, since .CSISPC changes it
        .CSISPC #OUTSP,#DEFEXT          ;Use .CSISPC to get filespec
        MOV     R5,SP                   ;Restore it
        .PURGE  #0
        .LOOKUP #AREA,#0,#OUTSP+36
        BCS     ERROR                   ;Branch if there was an error
        .HERR                           ;Now permit '?MON-F-' errors.
        .PRINT  #LUPOK                  ;Announce successful LOOKUP
        .EXIT                           ;Exit program

ERROR:  MOVB    @#52,R0                 ;was the error fatal?
        BMI     FTLERR                  ;Branch if yes
        .PRINT  #NOFIL
        BR      START                   ;Try again ...

FTLERR: CALL    DOSERR                  ;Display a .SERR error message
        BR      START                   ;try again ...

NOFIL:  .ASCIZ  "?HERR1-F-File Not Found"
LUPOK:  .ASCIZ  "!HERR1-I-Lookup succeeded"
        .EVEN                           ;Fix boundary
AREA:   .BLKW   4                       ;EMT Argument block
DEFEXT: .WORD   0,0,0,0                 ;No default extensions
OUTSP:  .BLKW   5*3                     ;Output specs go here
INSPEC: .BLKW   4*6                     ;Input specs go here
HANLOD: .BLKW   1                       ;Handlers begin loading here (if
                                        ;necessary)
```

```
        .TITLE  EHERR2.MAC

;+
;DOSERR subroutine displays a message for each .SERR error code.
;It expects the error code in R0 and destroys R0's contents.
;-

.MCALL  .PRINT

DOSERR::
        TST     R0                      ;Negative code?
        BPL     10$                     ;no
        CMP     R0,#TBLEND-TBL/2        ;In range?
        BGE     20$                     ;yes
10$:    CLR     R0                      ;No, unexpected code
20$:    ASL     R0                      ;Multiply by 2 to make an index
        MOV     TBL(R0),R0              ;Put message address into R0
        .PRINT                          ;and print it.
        RETURN
```

```
;                                       ;(Overlay error)
;                                       ;(invalid software sync. trap)
;                                       ;(memory management fault)
;                                       ;(memory error)
;                                       ;(FPU trap)
TBLEND: .WORD   UNLHAN                  ;unloaded handler (mapped)
        .WORD   INVDIR                  ;invalid directory
        .WORD   UNEXPE                  ;(Trap 10)
        .WORD   UNEXPE                  ;(Trap 4)
        .WORD   INVEMT                  ;invalid EMT number or subcode
        .WORD   INVCHN                  ;invalid channel number
        .WORD   INVADR                  ;invalid address in request
        .WORD   DIROVF                  ;directory overflow
        .WORD   UNEXPE                  ;(error from overlay handler)
        .WORD   BADFET                  ;bad fetch
        .WORD   DIRIOE                  ;directory I/O error
        .WORD   HANMEM                  ;device handler not in memory
        .WORD   UNEXPE                  ;(invalid address for USR SWAP)
TBL:    .WORD   UNEXPE                  ;unexpected code

UNEXPE: .ASCIZ  "?DOSERR-F-Unexpected .SERR (negative) error code"
HANMEM: .ASCIZ  "?DOSERR-F-Handler not in memory when required"
DIRIOE: .ASCIZ  "?DOSERR-F-Directory I/O error"
BADFET: .ASCIZ  "?DOSERR-F-.FETCH failed"
DIROVF: .ASCIZ  "?DOSERR-F-Directory overflow"
INVADR: .ASCIZ  "?DOSERR-F-Invalid address in EMT request"
INVCHN: .ASCIZ  "?DOSERR-F-Invalid channel number in EMT request"
INVEMT: .ASCIZ  "?DOSERR-F-Invalid EMT or subcode in request"
INVDIR: .ASCIZ  "?DOSERR-F-Invalid directory"
UNLHAN: .ASCIZ  "?DOSERR-F-Unloaded handler (.FETCH in mapped monitor)"
```

# .HRESET

**EMT 357**

The .HRESET request stops all I/O transfers in progress for the issuing job, and then performs an .SRESET request. (.HRESET is not used to clear a hard-error condition.) Only the I/O associated with the job that issued the .HRESET is affected by entering active handlers at their abort entry points. All other transfers continue.

Macro Call:

```
.HRESET
```

Errors:
None.

# .INTEN

**Macro Expansion**

Interrupt service routines use .INTEN:

- To notify the monitor that an interrupt has occurred and to switch to system state.
- To set the processor priority to the correct value.
- To save the contents of R4 and R5 before returning to the Interrupt Service Routine. Any other registers must be saved by the routine.

Macro Call:

**.INTEN prio[,pic]**

where:

**prio** is the processor priority at which the interrupt routine is to run, 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 specify this argument

.INTEN issues a subroutine call to the monitor and does not use an EMT instruction request.

All external interrupts must cause the processor to go to priority level 7. .INTEN is used to lower the priority to the value, at which point the device should be run. On return from .INTEN, the device interrupt can be serviced, at which point the interrupt routine returns with a RETURN instruction.

**NOTE**

An RTI instruction should not be used to return from an interrupt routine that uses an .INTEN.

Errors:
None.

Example:

```
        .TITLE  EINTEN.MAC

;+
; DL11.MAC - This is an example in the use of the .INTEN request.
; The example is an in-line, interrupt service routine, which may
; be assembled separately and linked with a mainline program.
; The routine transfers data from a user specified buffer to a DL11
; Serial Line Interface.
;
; CALLING FORMAT:       JSR     R5,DL11         ;Initiate Output
;                       .WORD   wordcount       ;# words to transfer
;                       .WORD   BUFFER          ;Address of Data Buffer
;                                       .
;                                       .
;               BUFFER: .BLKW   wordcount
;-

        .MCALL  .INTEN

DLVEC   = 304                           ;DL11 Vector ***
DLCSR   = 176504                        ;DL11 OUTPUT CSR ***
DLPRI   = 4                             ;DL11 Priority for RT-11

DL11::  MOV     (R5)+,(PC)+             ;I/O Initiation - Get word count
WCNT:   .WORD   0
        MOV     (R5)+,(PC)+             ;Get address of Data Buffer
BUFAD:  .WORD   0
        ASL     WCNT                    ;Make word count byte count
        BEQ     1$                      ;Just leave if zero word count
        MOV     #DLINT,@#DLVEC          ;Initialize DL11 interrupt vector
        BIS     #100,@#DLCSR            ;Enable interrupts
1$:     RTS     R5                      ;Return to caller

DLINT:  .INTEN  DLPRI                   ;Interrupt service - Notify RT-11
                                        ;and drop priority to that of DL11
        MOVB    @BUFAD,@#DLCSR+2        ;Transfer a byte
        INC     BUFAD                   ;Bump buffer pointer
        DEC     WCNT                    ;All bytes transferred?
        BEQ     DLDUN                   ;Branch if yes
        RETURN                          ;No return from interrupt thru RT-11

DLDUN:  BIC     #100,@#DLCSR            ;All done - disable DL11 interrupts
        RETURN                          ;Return thru RT-11
        .END
```

# .LOCK/.UNLOCK

**.LOCK: EMT 346**
**.UNLOCK: EMT 347**

### .LOCK

The .LOCK request keeps the USR in memory to provide any of its services required by your program. A .LOCK inhibits another job from using the USR.

Macro Call:

**.LOCK**

The .LOCK request reduces time spent in file handling by eliminating the swapping of the USR in and out of memory. The .LOCK request keeps other jobs from using the USR while it is in use and loads USR into memory if it is not already in memory. Under mapped monitors, USR is always in memory.

.LOCK causes the USR to be read into memory or swapped into memory. 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. (Note that certain calls always require a fresh copy of the USR.)

The .LOCK/.UNLOCK requests are complementary and must be matched. That is, if you have issued three .LOCK requests, you must issue at least three .UNLOCK requests. You can issue more .UNLOCK requests than .LOCK requests without error.

Calling the CSI or using a .GTLIN request can also perform an implicit and temporary .UNLOCK.

#### Notes

- Do not put executable code or data in the area occupied by the USR while it is locked, because you can't access the area until an .UNLOCK is issued. When USR has swapped over, the return from the .LOCK request is to the USR itself, rather than to the user program, In this way, the .LOCK function inhibits the user program from being re-read.
- Once a .LOCK has been performed, it is not advisable for the program to destroy the area occupied by the USR, even if no further use of the USR is required, because this causes unpredictable results when an .UNLOCK is done.
- 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:
Refer to the example for the .UNLOCK request.

### .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 program does a .LOCK, the lock count is incremented. When the user does an .UNLOCK, the lock count is decremented. When the lock count goes to 0, the user program is swapped back in.

Macro Call:

**.UNLOCK**

#### Notes

- The number of .UNLOCK requests must at least match the number of .LOCK requests that were issued. If more .LOCK requests are done, the USR remains locked in memory. Extra .UNLOCK requests in your program do no harm since they are ignored.
- With two jobs running, use .LOCK/.UNLOCK pairs only where absolutely necessary. When a one job locks the USR, the other job cannot use it until it is unlocked, which can degrade performance in some cases.
- Calling the CSI, with input coming from the terminal, results in an implicit (though temporary) .UNLOCK.
- Make sure that the .UNLOCK request is not in the same area that the USR swaps into; otherwise, the request can never be executed.

Errors:
None.

Example:

```
        .TITLE  ELOCK.MAC

;+
; .LOCK / .UNLOCK - This is an example in the use of the .LOCK and .UNLOCK
; requests. This example tries to obtain as much memory as possible (using
; the .SETTOP request), which will force the USR into a swapping mode. The
; .LOCK request will bring the USR into memory (over the high 2k of our little
; program !) and force it to remain there until an .UNLOCK is issued.
;-

        .MCALL  .LOCK,.UNLOCK,.LOOKUP
        .MCALL  .SETTOP,.PRINT,.EXIT

$USRRB  =:      53              ;(.SYCDF) User error byte
SUCCS$  =:      001             ;(.UEBDF) Success code
ERROR$  =:      004             ;(.UEBDF) Error code
FATAL$  =:      010             ;(.UEBDF) Fatal code
$SYSPTR =:      54              ;(.SYCDF) Pointer to beginning of RMON
```

```
START:  .SETTOP @#$SYSPTR       ;Try to allocate all of memory (up to
                                ;RMON)
        .LOCK                   ;bring USR into memory
        .LOOKUP #AREA,#0,#FILE1 ;LOOKUP a file on channel 0
        BCC     1$              ;Branch if successful
2$:     .PRINT  #LMSG           ;Print Error Message
        BISB    #FATAL$,@#$USRRB ;Flag error
        .EXIT                   ;then exit program

1$:     .PRINT  #F1FND          ;Announce our success
        MOV     #AREA,R0        ;R0 => EMT Argument Block
        INC     @R0             ;Increment low byte of 1st arg (chan #)
        MOV     #FILE2,2(R0)    ;Fill in pointer to new filespec
        .LOOKUP                 ;Do the .LOOKUP from filled in arg block
                                ;pointed to by R0.
        BCS     2$              ;Branch on error
        .PRINT  #F2FND          ;Say we found it
        .UNLOCK                 ;now release the USR
        BISB    #SUCCS$,@#$USRRB ;Flag success
        .EXIT                   ;and exit program

AREA:   .BLKW   3               ;EMT Argument Block
FILE1:  .RAD50  /SY/            ;A File we're sure to find
        .RAD50  /SWAP  /
        .RAD50  /SYS/
FILE2:  .RAD50  /SY/            ;Another file we might find
        .RAD50  /PIP   /
        .RAD50  /SAV/
LMSG:   .ASCIZ  /?ELOCK-F-Error on .LOOKUP/ ;Error message
F1FND:  .ASCIZ  /!ELOCK-I-Found SWAP.SYS/
F2FND:  .ASCIZ  /!ELOCK-I-Found PIP.SAV/
        .EVEN
        .END    START
```

# .LOOKUP

**EMT 375, Code 1**

A .LOOKUP request can be used in two different ways:

- As a standard .LOOKUP file under all monitors.
- As an MQ job .LOOKUP under system job monitors.

### Standard 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
.CLOSZ
.SAVESTATUS
.SRESET
.HRESET
.PURGE
.CSIGEN (if the channel is in the range 0-10 $_8$)

### Notes

If the program is overlaid, channel $17_8$ is used by the overlay handler and should not be modified.

If the first word of the file name (the second word of *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 LP:FILE.TYP), the name is ignored.

The handler for the selected device must be in memory for a .LOOKUP.

Be careful doing a non-file-structured .LOOKUP on a file-structured device. If your program writes data, corruption of the device directory can occur and effectively destroy the disk's contents. The RT–11 directory starts in absolute block 6.

In particular, avoid doing a .LOOKUP or .ENTER with a file specification where the file value is missing. Unless you know you want to open an entire device, always supply a dummy file name when issuing a .LOOKUP or .ENTER.

Macro Call:

**.LOOKUP area,chan,dblk[,seqnum]**

where:

**area** is the address of a three-word EMT argument block

**chan** is a channel number in the range 0-$376_8$

**dblk** is the address of a four-word Radix–50 descriptor of the file or device to be operated upon

**seqnum** is a file number for magtape.

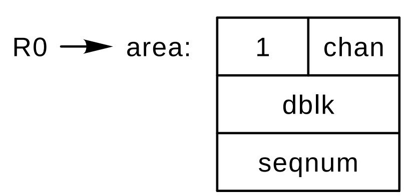
If this argument is blank, a value of 0 is assumed.

For magtape, it describes a file sequence number. The action taken depends on whether the file name is given or is null. The sequence number can have the following values:

| | |
|---|---|
| -1 | Means suppress rewind and search for a file name from the current tape position. If a file name is given, a file-structured lookup is performed (do not rewind). It is important that only -1 be specified and not any other negative number. If the file name is null, a non-file-structured lookup is done (tape is not moved). |
| 0 | Means rewind to the beginning of the tape and do a non-file-structured lookup. |
| n | Where n is any positive number. Position the tape at file sequence number *n* and check that the file names match. If the file names do not match, an error is generated. If the file name is null, a file-structured lookup is done on the file designated by *seqnum.* |

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 (typically magtape), R0 contains 0 to indicate the unknown length.

Request Format:




Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Channel already open. |
| 1 | File indicated was not found on the device. |

| | |
|---|---|
| 2 | File already open on a nonshareable device; for example, magtape. |
| 5 | Argument is invalid; for example, magtape file sequence number. |
| 6 | Error code is returned in $ERRBY if the request is issued to a nonexistent or otherwise invalid special-directory device unit. The handler determines the validity of the device unit. |

Example:

```
        .TITLE  ELOOKU.MAC

;+
; .LOOKUP - This is an example in the use of the .LOOKUP request.
; This example determines whether or not the RT-11 SWAP.SYS
; Workfile exists on device SY: and if so, prints its size in
; blocks on the console terminal.
;-

        .MCALL  .LOOKUP,.PRINT,.EXIT

$USRRB  =:      53              ;(.SYCDF) User error byte
SUCCS$  =:      001             ;(.UEBDF) Success code
ERROR$  =:      004             ;(.UEBDF) Error code
FATAL$  =:      010             ;(.UEBDF) Fatal code

START:  .LOOKUP #AREA,#0,#SPEC          ;See if there's a SY:SWAP.SYS
        BCC     1$                      ;Branch if there is
        .PRINT  #NOFIL                  ;Print 'File Not Found' message
        BISB    #FATAL$,@#$USRRB        ;indicate error
        .EXIT                           ;then exit program

1$:     MOV     #SIZE,R1                ;R1 => where to put ASCII size
        CALL    CNV10                   ;Convert size (in R0) to ASCII
        .PRINT  #BUFF                   ;Print size of QUFILE.TMP on console
        BISB    #SUCCS$,@#$USRRB        ;indicate success
        .EXIT                           ;then exit program

CNV10:  MOV     R0,-(SP)        ;Subroutine to convert Binary # in R0
        CLR     R0              ;to Decimal ASCII by repetitive
1$:     INC     R0              ;subtraction. The remainder for each
        SUB     #10.,@SP        ;radix is made into ASCII and pushed
        BGE     1$              ;on the stack, then the routine calls
        ADD     #72,@SP         ;itself. The code at 2$ pops the ASCII
        DEC     R0              ;digits off the stack and into the out-
        BEQ     2$              ;put buffer, eventually returning to
        CALL    CNV10           ;the calling program. This is a VERY
2$:     MOVB    (SP)+,(R1)+     ;useful routine, is short and is
        RETURN                  ;memory efficient.

AREA:   .BLKW   3                       ;EMT Argument Block
SPEC:   .RAD50  /SY SWAP  SYS/
BUFF:   .ASCII  /!ELOOKU-I-SY:SWAP.SYS = /
SIZE:   .ASCIZ  /    Blocks/
NOFIL:  .ASCIZ  /?ELOOKU-F-File Not Found SY:SWAP.SYS/
        .EVEN

        .END    START
```

### System Job Lookup

The foreground and background jobs can send messages to each other via the existing .SDAT/.RCVD/.MWAIT facility. A more general message facility is available to all jobs through the message queue (MQ) handler. By turning message handling into a formal "device" handler, and treating messages as I/O to jobs, the existing .READ /.WRITE/.WAIT mechanism can be used to transmit messages. A channel is opened to a job via a .LOOKUP request, after which standard I/O requests are issued to that channel.

Macro Call:

**.LOOKUP area,chan,jobdes**

where:

**area** is the address of a two-word EMT argument block

**chan** is the number of the channel to open

**jobdes** is the address of a four-word descriptor of the job to which messages will be sent or received:

```
jobdes: .RAD50 /MQ/             ;use MQ device
        .BYTE   0,0,0,0,0,0     ;insure null padding
10$:    .=.-6
        .ASCII /NAME/           ;Logical job name
.=10$
```

where *logical-job-name* can be from one to six characters long. It must be padded with nulls if less than six characters long. If *logical-job-name* is all nulls, the channel will be opened for .READ requests only and will accept messages from any job

Request Format:

R0 → area:

| | |
|---|---|
| 1 | chan |
| jobdes | |

The .LOOKUP request associates a channel with a specified job for the purposes of sending inter-task messages. R0 is undefined on return from the .LOOKUP.

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Channel not available. |
| 1 | No such job. |

Example:

```
        .TITLE  ELOOKB.MAC

;+
; .LOOKUP - This is an example in the use of the .LOOKUP request
; to open a message channel to a System Job, specifically, the
; companion ELOOKF program. ELOOKF changes A to B, B to C, etc.
; It must be run under a monitor generated with System Job
; Support and you must SRUN/FRUN ELOOKF first.
;-

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

$USRRB  =:      53                      ;(.SYCDF) User error byte
SUCCS$  =:      001                     ;(.UEBDF) Success code
ERROR$  =:      004                     ;(.UEBDF) Error code
FATAL$  =:      010                     ;(.UEBDF) Fatal code

START:  .LOOKUP #AREA,#0,#QMSG          ;Try to open a channel to ELOOKF
        BCC     1$                      ;Branch if successful
        .PRINT  #NOJOB                  ;Error...print error message
        BR      9$                      ;and done

1$:     .WRITW  #AREA,#0,#RMSG+2,#RMSGZ-RMSG-2/2 ;Send a message to ELOOKF
        BCS     2$                      ;Branch if error
        .READW  #AREA,#0,#RMSG,#RMSGZ-RMSG-2/2 ;Wait for an ack message
        BCS     2$                      ;Branch if error
        .PRINT  #RUN                    ;Announce ELOOKF alive and well
        .PRINT  #RMSG+2+6               ;Print returned value
        BISB    #SUCCS$,@#$USRRB        ;indicate it
        .EXIT                           ;Then exit

2$:     .PRINT  #MSGERR                 ;Print error message
9$:     BISB    #FATAL$,@#$USRRB        ;indicate it
        .EXIT                           ;Then exit

AREA:   .BLKW   5                       ;EMT Argument Block
QMSG:   .RAD50  /MQ/                    ;Job Descriptor Block for .LOOKUP
        .ASCII  /ELOOKF/
RMSG:   .WORD   0                       ;area for return count
        .ASCII  /ELOOKB/                ;our name (for reply)
        .ASCII  /ABCDEF/                ;data
        .ASCII  /GHIJKL/                ;...
RMSGZ:
        .BYTE   0
        .EVEN
MSGERR: .ASCIZ  /?ELOOKB-F-Message Error/ ;Error Messages, etc.
NOJOB:  .ASCIZ  /?ELOOKB-F-ELOOKF is not running/
RUN:    .ASCIZ  /!ELOOKB-I-ELOOKF is alive and running/
        .EVEN
        .END    START
        .TITLE  ELOOKF.MAC

;+
; .LOOKUP - This is the system job to be used with the previous
; example program.
;-

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

$USRRB  =:      53                      ;(.SYCDF) User error byte
SUCCS$  =:      001                     ;(.UEBDF) Success code
ERROR$  =:      004                     ;(.UEBDF) Error code
FATAL$  =:      010                     ;(.UEBDF) Fatal code
BUFLEN  =:      256.                    ;size of buffer
```

```
START:  .LOOKUP #AREA,#0,#QMSG          ;Try to open an MQ link to anyone
        BCS     10$                     ;Branch if error
10$:    .READW  #AREA,#0,#RMSG,#BUFLEN+1 ;Wait for a message
        BCS     50$                     ;Branch if error
        MOV     RMSG,R1                 ;load word count
        CMP     #BUFLEN,R1              ;is it all in the buffer?
        BGE     20$                     ;yes
        MOV     #BUFLEN,R1              ;else use truncated count
20$:    MOV     R1,R5                   ;save count
        SUB     #6/2,R1                 ;don't change caller's name
        ASL     R1                      ;make into byte count
        MOV     #RMSG+2+6,R0            ;point to data area
 30$:    INCB   (R0)+                   ;update a byte
         SOB    R1,30$                  ;through out the buffer

        MOV     #^RMQ,RMSG              ;Build DBLK for MQ
        .LOOKUP #AREA,#1,#RMSG          ;Try to open an MQ link to sender
        BCS     40$                     ;Branch if successful
        .WRITW  #AREA,#1,#RMSG+2,R5     ;Return data to other program
        BCS     50$                     ;Branch if error
        .PRINT  #RUN                    ;Announce communication worked
        BISB    #SUCCS$,@#$USRRB        ;indicate it
        .EXIT                           ;Then exit

40$:    .PRINT  #LOOKER                 ;Error...print error message
        BISB    #FATAL$,@#$USRRB        ;indicate it
        .EXIT                           ;then exit program

50$:    .PRINT  #MSGERR                 ;Print error message
        BISB    #FATAL$,@#$USRRB        ;indicate it
        .EXIT                           ;Then exit

AREA:   .BLKW   5                       ;EMT Argument Block
QMSG:   .RAD50  /MQ/                    ;Job Descriptor Block for .LOOKUP
        .BYTE   0,0,0,0,0,0             ;listen to any job
RMSG:   .BLKW   1                       ;Message buffer
        .BLKW   BUFLEN

MSGERR: .ASCIZ  /?ELOOKF-F-Message Error/ ;Error Messages, etc.
LOOKER: .ASCIZ  /?ELOOKF-F-LOOKUP on MQ failed/
RUN:    .ASCIZ  /!ELOOKF-I-Returned value to other program/
        .EVEN
        .END    START
```

# .MAP/.UNMAP

**.MAP: EMT 375, Code 36, Subcode 4**
**.UNMAP: EMT 375, Code 36, Subcode 5**

### 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 words of memory. The .MAP request checks to see if the specified window is already mapped. If it is, no unmapping and remapping operations are performed.

The .UNMAP request (See below) unmaps a window and flags that portion of the virtual address space as being inaccessible.

Macro Call:

**.MAP area[,addr]**

where:

**area** is the address of a two-word EMT argument block

**addr** is the address of the window definition block containing a description of the window to be mapped and the region to which it will map

Request Format:

R0 → area:

| | |
|---|---|
| 36 | 4 |
| addr | |

Errors:

| Code | Explanation |
|---|---|
| 2 | An invalid region identifier was specified. |
| 3 | An invalid window identifier was specified. |
| 4 | The specified window was not mapped because the offset is beyond the end of the region, the window is larger than the region or the window extends beyond the bounds of the region. |

Example:
See .CRAW.

### .UNMAP

The .UNMAP request unmaps a window and makes inaccessible that portion of the program's virtual address space. 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:

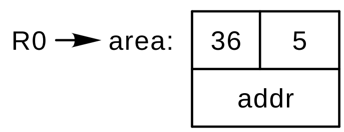
**.UNMAP area,addr**

where:

**area** is the address of a two-word argument block

**addr** is the address of the window control block that describes the window to be unmapped

Request Format:




Errors:

| Code | Explanation |
|---|---|
| 3 | An illegal window identifier was specified. |
| 5 | The specified window was not already mapped. |

Example:
See .CRAW.

# .MFPS/.MTPS

### Macro Expansion

The .MFPS and .MTPS macro calls allow processor-independent user access to the processor status word. The contents of the registers 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. (For another way to access the PS, see .PEEK/.POKE.)

In the mapped monitor, .MFPS and .MTPS can be used only by virtual jobs using .CALLK.

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 bits.

Macro Call:

**.MTPS value**

where:

**value** is either the value or the address of the value (depending on addressing mode) to be placed in the PSW. If value 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 value is present. If value is not present, you should set the stack to the appropriate value. In either case, the lower byte on the stack is put in the processor status word.

Perform MTPS and MFPS operations and access the condition codes by following this special technique:

1. To get the PSW or to set the PSW to a desired value, follow this sequence of instructions:

```
;+
; TO GET PS ...
;-
        CALL    MFPS          ;Get PS
                              ;Continue, PS in on stack ...

;+
; TO PUT PS ...
;-
```

```
        MOV     NEWPS,-(SP)  ;Put desired PS on stack ...
        CALL    MTPS         ;Call MTPS
                             ;Continue process w/ new PS ...
```

Errors:

None.

Example:

In the beginning of your program, set up the IOT trap vector as follows:

```
V.IOT   =:      20              ;IOT vector address
PR7     =:      340             ;PS priority 7

        .ASECT                  ;Set up IOT
. = V.IOT
        .WORD   GETPS           ;IOT service address in 'MFPS' subroutine
        .WORD   PR7             ; Priority 7
```

Elsewhere in your program place the following routines:

```
;+
; MFPS/MTPS ROUTINES ...
;-

MFPS:   IOT                     ;Execute IOT
                                ;Will return to caller w/ PS on stack

GETPS:  MOV     4(SP),@SP       ;Put user return on top
        MOV     2(SP),4(SP)     ;Move PS saved by IOT
MTPS:   RTI                     ;Will return to caller w/ new PS


        .TITLE  EMFPS4.MAC

;+
; .MFPS / .MTPS - This is an example in the use of the .MFPS and .MTPS
; requests. The example is a skeleton mainline program which calls a
; subroutine to get the next free element in an RT11-like linked queue.
;-

        .MCALL  .MFPS,.MTPS,.EXIT,.PRINT,.TTINR

$JSW    =:         44                   ;Job Status Word location
TTSPC$  =:      10000                   ;TTY Special bit
PR7     =:        340                   ;Priority 7 in PS

START:                                  ;Skeleton mainline program...
        BIS     #TTSPC$,@#$JSW          ;Set TTY Special bit
        ...
        CALL    GETQUE                  ;Call subroutine to return next free
                                        ;element - on return R5 => element
        BCC     1$                      ;Branch if no error
        .PRINT  #NOELEM                 ;No more elements available
        BIC     #TTSPC$,@#$JSW          ;Reset special bit
        .EXIT                           ;Exit program

1$:     ...                             ;Program continues
        .PRINT  #GOT1                   ;Announce success
2$:     .TTINR                          ;Wait for a key to be hit on console
        BCS     2$
        BR      START
```

```
GETQUE: MOV     #QHEAD,R4               ;Point to queue head
        TST     @R4                     ;Queue exhausted?
        BEQ     11$                     ;Yes...set error on leaving
        .MFPS                           ;Save status on stack
        .MTPS   #PR7                    ;Raise priority to 7
        MOV     @R4,R5                  ;R5 points to next element
        MOV     @R5,@R4                 ;Relink the queue
        .MTPS                           ;Restore previous status
        TST     (PC)+                   ;This clears carry & skips next
                                        ; instruction
11$:    SEC                             ;Set carry bit (to flag error)
        RETURN                          ;Return to caller

QHEAD:  .WORD Q1                        ;Queue head
Q1:     .WORD Q2,0,0                    ;3 linked queue elements
Q2:     .WORD Q3,0,0
Q3:     .WORD 0,0,0

NOELEM: .ASCIZ  /?EMFPS4-W-No more Queue Elements Available/
GOT1:   .ASCIZ  /Element acquired...press any key to continue/

        .END    START
```

# .MRKT

**EMT 375, Code 22**

The .MRKT request schedules a completion routine to be entered after a specified time interval (measured in clock ticks) has elapsed. Single-job monitor SB is distributed without timer support, but it is a selectable option at SYSGEN.

A .MRKT request requires 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 (See .CMKT request). You should allocate enough queue elements to handle at least as many mark time and I/O requests that you expect may be simultaneously pending.

Macro Call:

**.MRKT area,time,crtn,id**

where:

| | |
|---|---|
| **area** | is the address of a four-word EMT argument block |
| **time** | is the address of a two word-block containing the time interval (high order first, low order second), specified as a number of clock ticks |
| **crtn** | is the entry point of a completion routine |
| **id** | is a non-zero number or memory address assigned by the user to identify the particular request to the completion routine and to any cancel mark time requests. The number must be within the range 1–176777; the rest (177700–177777) 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  EMRKT.MAC

;+
; .MRKT/.CMKT - This is an example of the use of the .MRKT/.CMKT requests.
; The example illustrates a user implemented "Timed Read" to cancel an
; input request after a specified time interval.
;-

        .MCALL  .MRKT,.TTINR,.EXIT,.PRINT
        .MCALL  .TTYOUT,.CMKT,.TWAIT,.QSET

        LF =: 12                        ;Line Feed
        $JSW =: 44                      ;(.SYCDF) Job Status Word location
        TCBIT$ =: 100                   ;(.JSWDF) Return C-bit bit in JSW
        TTSPC$ =: 10000                 ;(.JSWDF) Special Mode bit in JSW

START:  .QSET   #XQUE,#1                ;Need an extra Q-Elem for this
10$:    MOV     #PROMT,R0               ;Mainline - R0 => Prompt
        MOV     #BUFFR,R1               ;R1 => Input buffer
        CALL    TREAD$                  ;Do a "timed read"
        BCS     20$                     ;C-bit set = Timed out
        .PRINT  #LINE                   ;"Process" data...
        BR      10$                     ;Go back for more
20$:    .PRINT  #TIMOUT                 ;Read timed out - could process
        .EXIT                           ;partial data but we'll just exit

        ;* TREAD$ - "Timed Read" Subroutine                             *
        ;* Input:     R0 => Prompt String / R0 = 0 if no prompt         *
        ;*            R1 => Input Buffer                                *
        ;* Output:    Buffer contains input chars, if any, terminated   *
        ;*            by a null char. C-Bit set if timed out            *
TREAD$: TST     R0                      ;See if we have to prompt
        BEQ     10$                     ;Branch if no...
        .PRINT                          ;Output prompt
10$:    CLR     TBYT                    ;Clear time-out flag
        .MRKT   #TAREA,#TIME,#TOUT,#1   ;Issue a .MRKT for 10 sec
        BIS     #TCBIT$,@#$JSW          ;Set NoWait bit in JSW
        CLRB    @R1                     ;Start with "empty" buffer
TTIN:   .TWAIT  #AREA                   ;Wait so we don't lock out BG
        .TTINR                          ;Look for a character
        BIT     #1,TBYT                 ;*C*Timed out?
        BNE     10$                     ;*C*Branch if yes
        BCS     TTIN                    ;Branch if input not complete
        MOVB    R0,(R1)+                ;Xfer 1st character
        .CMKT   #TAREA,#0               ;Cancel .MRKT
10$:    BIS     #TTSPC$,@#$JSW          ;Turn on TT: Special mode
20$:    .TTINR                          ;Flush TT: ring buffer
        MOVB    R0,(R1)+                ;*C*putting characters in user buffer
        BCC     20$                     ;If more char, go get 'em
        CLRB    -(R1)                   ;Terminate input with null byte
        BIC     #TCBIT$!TTSPC$,@#$JSW   ;Clear bits in JSW
        ROR     TBYT                    ;Set carry if timed out
        RETURN                          ;Return to caller

TOUT:   INC     TBYT                    ;Indicate time out happened
        RETURN                          ;Leave completion code
```

```
TBYT:   .WORD   0                       ;Time-out flag
XQUE:   .BLKW   10.                     ;Extra Q-Element
AREA:   .WORD   0,WAIT                  ;EMT Argument block for .TWAIT
TAREA:  .BLKW   4                       ;EMT Argument block for .MRKT
TIME:   .WORD   0,600.                  ;Time-out interval (10 sec)
WAIT:   .WORD   0,1                     ;1/60 sec wait between .TTINRs
LINE:   .ASCII  "!EMRKT-I-Not in stock - Part # " ;Dummy response
BUFFR:  .BLKB   81.                     ;User input buffer
PROMT:  .ASCII  "Enter Part # >"        ;Prompt
        .BYTE   200                     ;No CRLF
TIMOUT: .ASCIZ  "!EMRKT-I-Timed read expired" ;Too bad message
        .END    START
```

## .MSDS

**EMT 375, Code 46**

The .MSDS sets the lockstep of User data space and Supervisor data space and returns the old status in R0. This value is not implemented in unmapped monitors.

Macro Call:

**.MSDS area,value[,CODE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a two-word EMT request block area |
| **value** | is the setting value desired. |
| **CODE=strg** | specifies *strg* as either "SET" (default), "NOSET", "SP" or "STACK" |

**Notes**

In Supervisor mode when you want to establish your own data space, distinct from User data space, you may not own any data space memory. Therefore, you cannot use standard request code. .CMAP, .GCMAP and .MSDS introduce a concept that allows you to specify CODE = "SP" or "STACK". In this way, you use "STACK" to:

- Build a request block on the stack
- Issue the request
- Clear the stack of the request

Errors:
None.

Example:

```
        .TITLE  EMSDS;2

.MCALL  .CRRG   .MSDS   .CMPDF
        .CMPDF

;               Assume the following is running in Supy mode
        ...
        .MSDS   ,#CM.PR7,CODE=STACK     ;disconnect PAR7 Supy D

;               Here build a WDB and .CRRG request on the stack
;               which remaps PAR7 Supy D
        ...
        .CRRG   SP                      ;map to Supy data in PAR7

;               Here use the Supy data
        ...
        .MSDS   ,#0,CODE=STACK          ;reconnect PAR7 User & Supy
        ...
```

# .MTATCH

**EMT 375, Code 37, Subcode 5**

.MTATCH is a multiterminal feature which must be selected at SYSGEN. The .MTATCH request attaches a terminal for exclusive use by the requesting job. This operation must be performed before any job can use a terminal with multiterminal programmed requests, although a job can issue a .MTGET request before a .MTATCH.

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 system option you can select at SYSGEN.) In a fully-mapped monitor, if you set the low bit of *addr* on, it will be treated as a Supervisor/Data space; otherwise, it is treated as a User/Data space address. |
| **unit** | is the logical unit number of the terminal (The logical unit number is the number assigned by the system to a particular physical unit during the system generation process.) |

Request Format:

<table>
<tr><td>R0 → area:</td><td>37</td><td>5</td></tr>
<tr><td></td><td colspan="2">addr</td></tr>
<tr><td></td><td>0</td><td>unit</td></tr>
</table>

Errors:

| **Code** | **Explanation** |
|---|---|
| 2 | Nonexistent logical unit number. |
| 3 | Invalid request; function code out of range. |
| 4 | Unit attached by another job (job number returned in R0). |
| 5 | In mapped monitors, the optional status word address is not in valid user virtual address space. |
| 6 | Unit attached by a handler (Radix–50 handler name returned in R0) |

Example:

```
        .TITLE  EMTXXX;2
;+
; EMTXXX.MAC - The following is an example program that
; demonstrates the use of the multiterminal
; programmed requests. The program attaches all the
; terminals on a given system, then proceeds with an
; input/echo exercise on all attached terminals until
; CTRL/C is sent to it.
;-
.MCALL  .MTATCH,.MTPRNT,.MTGET,.MTIN,.MTOUT
.MCALL  .PRINT,.MTRCTO,.MTSET,.MTSTAT,.EXIT

HNGUP$  =: 4000                         ;Terminal off-line bit
TTSPC$  =: 10000                        ;Special mode bit
TTLC$   =: 40000                        ;Lower-case mode bit
AS.INP  =: 40000                        ;Input available bit
M.TSTS  =: 0                            ;Terminal status word
M.TSTW  =: 7                            ;Terminal state byte
M.NLUN  =: 4                            ;# of LUNs word

ESC     =:      033                     ;Escape char

.ENABL  LSB

EMTXXX:                                 ;Start of program
        .MTSTAT #MTA,#MSTAT             ;Get MTTY status
        MOV     MSTAT+M.NLUN,R4         ;R4 = # LUNs
        BEQ     MERR                    ;None? Not MTTY!!!
        CLR     R1                      ;Initial LUN = #0
        MOV     #AST,R2                 ;R2 -> AST word array
10$:    .MTATCH #MTA,R2,R1              ;Attach terminal
        BCC     20$                     ;Success!
        CLRB    TAI(R1)                 ;Set attach failed
        BR      30$                     ;Proceed with next LUN
20$:    MOVB    #1,TAI(R1)              ;Attach successful
        MOV     R1,R3                   ;Copy LUN
        ASL     R3                      ;Multiply by 8 for offset
        ASL     R3                      ;to the terminal status
        ASL     R3                      ;block...
        ADD     #TSB,R3                 ;R3 -> LUN's TSB
        .MTGET  #MTA,R3,R1              ;Get LUN's status
        BIS     #TTSPC$+TTLC$,M.TSTS(R3) ;Set special
                                        ;mode and lower case
        .MTSET  #MTA,R3,R1              ;Set LUN's status
        BITB    #HNGUP$/400,M.TSTW(R3)  ;On line?
        BNE     30$                     ;Nope!
        .MTRCTO #MTA,R1                 ;Reset CTRL/O
        .MTPRNT #MTA,#HELLO,R1          ;Say hello...
30$:    ADD     #2,R2                   ;R2 -> Next AST word
        INC     R1                      ;Get next LUN
        CMP     R1,R4                   ;Done?
        BLOS    10$                     ;Nope, go attach another

.DSABL  LSB
.ENABL  LSB
```

```
LOOP:                                           ;Input & echo forever
        CLR     R1                              ;Initial LUN = 0
        MOV     #AST,R2                         ;R2 -> AST words
10$:    TSTB    TAI(R1)                         ;Terminal attached?
        BEQ     20$                             ;Nope...
        BIT     #AS.INP,(R2)                    ;Any input?
        BEQ     20$                             ;Nope...
        .MTIN   #MTA,#MTCHAR,R1,#1              ;Input a character
        BCS     ERR                             ;Ooops! Error on input
        .MTOUT  #MTA,#MTCHAR,R1,#1              ;Echo the character
        BCS     ERR                             ;Ooops! Error on output
20$:    ADD     #2,R2                           ;Point to next AST word
        INC     R1                              ;Get next LUN
        CMP     R1,R4                           ;Done them all?
        BLOS    10$                             ;No, go check another
        BR      LOOP                            ;Yes, repeat (forever!)

ERR:    .PRINT  #UNEXP                          ;Unexpected error...
        .EXIT                                   ;Print message & exit!
MERR:   .PRINT  #NOMTTY                         ;Not multiterminal
        .EXIT                                   ;Print message & exit

AST:    .BLKW   16.                             ;Asynchronous Terminal
                                                ;Status Words (1/LUN)
TAI:    .BLKB   16.                             ;Terminal attached list
                                                ;1 Byte per LUN...
                                                ;0 = Not attached
MSTAT:  .BLKW   8.                              ;MTTY status block
TSB:    .BLKW   16.*4.                          ;Terminal status blocks
                                                ;16. blocks of 4 words
MTA:    .BLKW   4                               ;EMT argument block
MTCHAR: .BYTE   0                               ;Character stored here

HELLO:  .BYTE   ESC
        .ASCII  "[H"                            ;home the cursor
        .BYTE   ESC
        .ASCII  "[J"                            ;erase rest (all of screen)
        .ASCIZ  "Hello! Characters typed will be echoed"
NOMTTY: .ASCIZ  "?EMTXXX-F-Not multiterminal system"
UNEXP:  .ASCIZ  "?EMTXXX-F-Unexpected error"

        .END    EMTXXX                          ;End of program
```

# .MTDTCH

**EMT 375, Code 37, Subcode 6**

.MTDTCH is a multiterminal feature which must be selected during SYSGEN. The request detaches a terminal from one job and makes it available for other jobs. When a terminal is detached, it is deactivated, and unsolicited inputs are ignored. Input is disabled immediately, but any characters in the output buffer are printed. Attempts to detach a terminal not attached by the current job result in an error.

Macro Call:

**.MTDTCH area,unit**

where:

**area** is the address of a three-word EMT argument block

**unit** is the logical unit number (lun) of the terminal to be detached

Request Format:

| R0 → area: | 37 | 6 |
|---|---|---|
| | unused | |
| | -- | unit |

Errors:

| Code | Explanation |
|---|---|
| 1 | Unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |

Example:

```
        .TITLE  EMTDTC;2
;+
;       Attach to a multi-terminal unit, print a message
;       then detach from it.
;-
        .MCALL  .MTDTCH,.MTPRNT,.MTATCH,.EXIT,.PRINT

LUN     =:      3

START:
        .MTATCH #MTA,#0,#LUN    ;Attach to LUN
        BCS     10$             ;Attach error
        .MTPRNT #MTA,#MESS,#LUN ;Print message
        BCS     20$             ;Unexpected error
        .MTDTCH #MTA,#LUN       ;Detach LUN
        BCS     20$             ;Unexpected error
        .EXIT
```

```
10$:    .PRINT  #ATTERR         ;Attach error
                                ;(printed on console)
        .EXIT

20$:    .PRINT  #UNKERR         ;Unexpected error
                                ;(printed on console)
        .EXIT

ATTERR: .ASCIZ  "?EMTDTC-F-Attach error"
UNKERR: .ASCIZ  "?EMTDTC-F-Unexpected error"
MESS:   .ASCIZ  "!EMTDTC-I-Detaching terminal"
        .EVEN
MTA:    .BLKW   3
        .END    START
```

# .MTGET

**EMT 375, Code 37, Subcode 1**

.MTGET is a multiterminal feature which must be selected during SYSGEN. Issuing the .MTGET request returns the status of the specified terminal unit to the caller. If a .MTGET request fails because the terminal is owned by another job, the job number of the owner (or name of the handler) is returned in R0. You do not need to do an .MTATCH before using the .MTGET request. See .MTSET.

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

**unit** is the logical unit number (*lun*) of the terminal whose status is requested. A unit need not be attached to the job issuing a .MTGET request. If the unit is attached to another job (error code 4), the terminal status will be returned and the job number will be contained in R0. For all other error conditions, the contents of R0 are undefined.

R0 → area:

| | |
|---|---|
| 37 | 1 |
| addr | |
| −− | unit |

The status block has the following structure:

| | |
|---|---|
| M.TSTS | |
| M.TST2 | |
| M.FCNT | M.TFIL |
| M.TSTW | M.TWID |

The following information is contained in the status block:

| Byte Offset | Description |
|---|---|
| 0 (M.TSTS) | Terminal configuration word 1 |
| 2 (M.TST2) | Terminal configuration word 2 |
| 4 (M.TFIL) | Character requiring fillers |
| 5 (M.FCNT) | Number of fillers |
| 6 (M.TWID) | Carriage width |
| 7 (M.TSTW) | Terminal status byte (high byte of TSTDF) |

Note that if an error occurs, and the error code is not 1 or 4, the status block will not have been modified.

### Terminal Configuration Word 1 - M.TSTS

The bit definitions for terminal configuration word 1 (M.TSTS) are as follows:

| Name | Value | Bit | Meaning |
|---|---|---|---|
| HWTAB$ | 1 | 0 | Terminal has hardware tab |
| CRLF$ | 2 | 1 | Output RETURN when carriage width exceeded |
| FORM$ | 4 | 2 | Terminal has hardware form feed |
| FBTTY$ | 10 | 3 | Process CTRL/F and CTRL/B (and CTRL/X if system job) as special command characters (If clear, CTRL/F and CTRL/B are treated as ordinary characters.) |
| TCBIT$ | 100 | 6 | Inhibit TT wait (similar to bit 6 in the Job Status Word) |
| PAGE$ | 200 | 7 | Enable CTRL/S-CTRL/Q processing |
| | 7400 | 8-11 | Line speed (baud rate) mask. The terminal baud rate values for DZ11 /DZV11 and DH for bits 11–8 are as follows: |

| Mask (M.TSTS bits 11-8) | DZ Baud Rate | DH Baud Rate |
|---|---|---|
| 0000 | 50 | 50 |
| 0400 | 75 | 75 |
| 1000 | 110 | 110 |
| 1400 | 134.5 | 134.5 |
| 2000 | 150 | 150 |

| | | |
|---|---|---|
| 2400 | 300 | 300 |
| 3000 | 600 | 600 |
| 3400 | 1200 | 1200 |
| 4000 | 1800 | 1800 |
| 4400 | 2000 | 2000 |
| 5000 | 2400 | 2400 |
| 5400 | 3600 | 38400 |
| 6000 | 4800 | 4800 |
| 6400 | 7200 | 7200 |
| 7000 | 9600 | 9600 |
| 7400 | n/a | 19200 |

| Name | Value | Bit | Meaning |
|---|---|---|---|
| TTSPC$ | 10000 | 12 | Character mode input (same as bit 12 in Job Status Word) |
| REMOT$ | 20000 | 13 | Terminal is remote (Read-only bit) |
| TTLC$ | 40000 | 14 | Lowercase to uppercase conversion disabled |
| BKSP$ | 100000 | 15 | Use backspace for rubout (video type display) |

#### Terminal Configuration Word 2 - M.TST2

The bit definitions for terminal configuration word 2 (M.TST2) are as follows:

| Name | Value | Bit | Meaning |
|---|---|---|---|
| CHRLN$ | 3 | 0-1 | Character length, which can be 5(00), 6(01), 7(10), or 8(11) bits (DZ only) |
| USTOP$ | 4 | 2 | Unit stop, which sends one stop bit when clear, two stop bits when set (DZ only) |
| PAREN$ | 10 | 3 | Parity enable (DZ only) |
| ODDPR$ | 20 | 4 | Odd parity when set; even parity when clear |
| | 140 | 5-6 | Reserved |
| RPALL$ | 200 | 7 | Read pass all |
| | 77400 | 8-14 | Reserved |
| WPALL$ | 100000 | 15 | Write pass all |

### Terminal Status Byte - M.TST2

The bit definitions for terminal status byte (M.TST2) are as follows:

| Name | Value | Bit | Meaning |
|---|---|---|---|
| FILL$ | 1 | | Fill sequence in progress |
| CTRLU$ | 2 | | CTRL/U in progress |
| DTACH$ | 20 | | Detach in progress |
| WRWT$ | 40 | | TT: I/O flag |
| INEXP$ | 100 | | Output interrupt is expected |
| PAGE$ | 200 | | Output is suspended by XOFF |
| SHARE$ | 2000 | 10 | Terminal is shared console |
| HNGUP$ | 4000 | 11 | Terminal has hung up |
| DZ11$ | 10000 | 12 | Terminal interface is DZ11 |
| CTRLC$ | 40000 | 14 | Double CTRL/C was entered (The .MTGET request resets this bit in the terminal control block if it is on.) |
| CONSL$ | 100000 | 15 | Terminal is acting as console |

Errors:

| Code | Explanation |
|---|---|
| 1 | Invalid unit number, unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Invalid request; function code out of range. |
| 4 | Unit attached by another job (job number returned in R0). |
| 5 | In the XM monitor, the status block address is not in valid user virtual address space. |
| 6 | Unit attached by a handler (Radix–50 handler name returned in R0) |

Example:
Refer to the example for the .MTATCH request.

# .MTIN

**EMT 375, Code 37, Subcode 2**

.MTIN is a multiterminal feature that must be selected at SYSGEN. The .MTIN request reads characters from a terminal. It is the multiterminal form of the .TTYIN request. The .MTIN request moves one or more characters from the input ring buffer to a buffer you specify. The terminal must be attached. An updated user buffer address is returned in R0 if the request is successful. 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 input buffer. If the |
| **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 0 to $255_{10}$. A character count of zero means one character |

TCBIT$ and TTSPC$ in the M.TSTS word (See the .MTSET request) affect how the .MTIN request processes input. TCBIT$, the *inhibit terminal wait* bit, determines whether the .MTIN request waits or returns an error immediately if the appropriate input is not available at the time the request is issued:

- If TCBIT$ is clear, the .MTIN request waits and the job is suspended until the appropriate input is available.
- If TCBIT$ is set and the appropriate input is not available, .MTIN returns immediately with the carry bit set.

TTSPC$, the *special mode* bit, determines what type of input is needed—an entire line or a single character:

- If TTSPC$ is clear (normal mode I/O), input is available to the user program only after one of the following line terminators has been typed: carriage return, line-feed, CTRL/Z, or CTRL/C. Typing any of these passes all characters on that line, one by one, to the user program.
- When TTSPC$ is set, it selects special mode I/O, in which each character is immediately available to the user program as it is typed. See the .TTYIN request for more information on normal and special mode I/O operation.

If TCBIT$ is set and TTSPC$ is clear, the .MTIN request returns immediately with the carry bit set (code 0), if a line is not available.

If TCBIT$ is set and TTSPC$ is set (special mode I/O), the .MTIN request returns immediately with the carry bit set, if a character is not available.

Results are similar for the system console if TCBIT$ of the JSW is set. The relationship between TCBIT$ and TTSPC$ in the terminal configuration word (M.TSTS) for the .MTIN programmed request is as follows:

| TCBIT$ | TTSPC$ | Meaning |
|---|---|---|
| 0 | 0 | Normal mode of input (echo provided); wait for line |
| 1 | 0 | Carry bit set if no line is available |
| 1 | 1 | Carry bit set if no character is available |
| 0 | 1 | No echo provided; wait for character |

Request Format:

<table>
<tr><td>R0 → area:</td><td>37</td><td>2</td></tr>
<tr><td></td><td colspan="2">addr</td></tr>
<tr><td></td><td>chrcnt</td><td>unit</td></tr>
</table>

Errors:

| Code | Explanation |
|---|---|
| 0 | No input available. TCBIT$ is set in the Job Status Word (for the system console) or in M.TSTS by the .MTSET request. |
| 1 | Unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |
| 5 | In the mapped monitor, the user buffer address is not in valid user virtual address space. |

Example:
Refer to the example for the .MTATCH request.

## .MTOUT

**EMT 375, Code 37, Subcode 3**

This multiterminal feature, selected during SYSGEN, is the multiterminal form of the .TTYOUT request. The .MTOUT request places characters into the user buffer in both User and Supervisor modes:

- If the program is executing in Supervisor mode, characters go into Supervisor-mapped data space, if enabled; otherwise, characters go into Supervisor instruction space.
- If the program is executing in User mode, characters go into User-mapped data space, if enabled; otherwise, characters go into User instruction.

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 output buffer. If low order bit of *addr* word is zero, the data buffer is in User space. If low order bit is one, the data buffer is in Supervisor space. |
| **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 0 to $255_{10}$ |

**Notes**

The .MTOUT request moves one or more characters from the user's buffer to the output ring buffer of the attached terminal. An updated user buffer address is returned in R0 if the request is successful.

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. TCBIT$ in terminal configuration word M.TSTS determines the response to the request:

- If TCBIT$ in M.TSTS is clear, the request waits until it can complete the full transfer.
- If TCBIT$ is set, the request returns with a partial transfer. R0 contains the updated buffer address (pointing past the last character transferred), the C bit is set, and the error code is 0.

For the .MTOUT request, the meaning of TCBIT$ in M.TSTS is as follows:

| **TCBIT$** | **Meaning** |
|---|---|
| 0 | Normal mode for output; wait for room in buffer |
| 1 | Carry bit set: no room in output ring buffer |

Request Format:

| R0 → area: | 37 | 3 |
|---|---|---|
| | addr | |
| | chrcnt | unit |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | No room in output buffer. |
| 1 | Unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |
| 5 | In the mapped monitor, the user buffer address is not in valid user virtual address space. |

Example:
Refer to the example for the .MTATCH request.

# .MTPRNT

**EMT 375, Code 37, Subcode 4**

This multiterminal feature must be selected during SYSGEN. The .MTPRNT request causes one or more lines to be printed at the specified terminal in a multiterminal environment. This request is the multiterminal equivalent of the .PRINT request (See .MTSET request for more details). Like the string used with the .PRINT request, the string to be printed must be terminated with a null byte or a 200 byte:

**.ASCIZ /string/**

or

**.ASCII /string/<200>**

The null byte causes a RETURN/LINE FEED combination to be printed after the string. The 200 byte suppresses the RETURN/LINE FEED combination and leaves the carriage positioned after the last character of the string. 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 |
| **unit** | is the unit number associated with the terminal |

Request Format:

R0 → area:

| | |
|---|---|
| 37 | 4 |
| addr | |
| -- | unit |

Errors:

| **Code** | **Explanation** |
|---|---|
| 1 | Unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |
| 5 | In the mapped monitor, the character string address is not in valid user virtual address space. |

Example:
Refer to the example for the .MTATCH request.

# .MTPS

See .MFPS/.MTPS.

# .MTRCTO

**EMT 375, Code 37, Subcode 4**

This multiterminal feature must be selected during SYSGEN. The .MTRCTO request resets the CTRL/O switch of the specified terminal and enables terminal output in a multiterminal environment. It is the multiterminal equivalent of the .RCTRLO request.

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:

| **Code** | **Explanation** |
|---|---|
| 1 | Unit not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |

Example:
Refer to the example for the .MTATCH request.

# .MTSET

**EMT 375, Code 37, Subcode 0**

.MTSET is a multiterminal feature which must be selected during SYSGEN. This multiterminal request:

- Sets terminal and line characteristics.
- Determines the input/output mode of the terminal service requests for the specified terminal.

For more detail on line characteristics, such as baud rate, number of data bits, stop bits, and parity, refer to .MTGET.

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 set.

If low order bit of *addr* word is zero, the status block is in User space. If low order bit is one, the status block is in Supervisor space.

**unit** is the logical unit number associated with the line and terminal

Request Format:

| | | |
|---|---|---|
| R0 → area: | 37 | 0 |
| | addr | |
| | -- | unit |

The .MTSET request sets the parameters listed below. When the program returns from the request, the following information is returned to the status block:

| Byte Offset | Contents |
|---|---|
| 0 (M.TSTS) | Terminal configuration word 1 (The bit definitions are the same as those for the .MTGET request.) |
| 2 (M.TST2) | Terminal configuration word 2 (The bit definitions are the same as those for the .MTGET request.) |
| 4 (M.TFIL) | Character requiring fillers |
| 5 (M.FCNT) | Number of fillers |
| 6 (M.TWID) | Carriage width (byte) |

When issuing the .MTSET request, proceed as follows:

1. Issue an .MTGET request before using .MTSET.
2. Use BIS and BIC instructions to set or clear bit fields, modifying only the bits or bytes that you intend to change.
3. Issue the .MTSET request to replace the previous terminal status with the updated status.

Note that if an error occurs, and the error code is not 1, the status block remains unmodified.

Errors:

| Code | Explanation |
|---|---|
| 1 | *lun* not attached. |
| 2 | Nonexistent logical unit number. |
| 3 | Function code out of range. |
| 5 | In the mapped monitor, the status block address is not in valid user virtual address space. |

Example:
Refer to the example for the .MTATCH request.

# .MTSTAT

**EMT 375, Code 37**

.MTSTAT is a multiterminal feature that must be selected at SYSGEN. The request returns multiterminal system status information.

Macro Call:

**.MTSTAT area,addr**

where:

| | |
|---|---|
| **area** | is the address of a three-word EMT block |
| **addr** | is the address of an eight-word status block where multiterminal status information is returned. |
| **addr** | is the address of a four-word status block containing the line and terminal status being requested. |

| Byte Offset | Explanation |
|---|---|
| 0 (MST.1T) | Offset from the base of the resident monitor to the first terminal control block (TCB) |
| 2 (MST.CT) | Offset from the base of the resident monitor to the terminal control block of the console terminal for the program |
| 4 (MST.LU) | The value (0-16 decimal) of the highest logical unit number (LUN) built into the system |
| 6 (MST.ST) | The size of the terminal control block in bytes |
| 10-17 | Reserved |

Request Format:

| R0 → area: | 37 | 10 |
|---|---|---|
| | addr | |
| | 0 | |

Errors:

| Code | Explanation |
|---|---|
| 3 | Function code out of range |
| 5 | In mapped monitors, the status block address is not in valid user address space. |

Example:

Refer to the example for the .MTATCH request.

# .MWAIT

**EMT 374, Code 11, Subcode 0**

This request is similar to the .WAIT request, except that .MWAIT suspends execution of the job issuing the request until all messages sent to the other job or requested from the other job have been received. Using .MWAIT with .RCVD or .SDAT modes of message handling causes the program to wait until data is transferred. This request is available only under multijob monitors.

Macro Call:

**.MWAIT**

Request Format:

| | | |
|---|---|---|
| R0 = | 11 | 0 |

Errors:
None.

Example:

```
;+
; .MWAIT - This is an example in the use of the .MWAIT request.
; The example is actually two programs, a Background job
; which sends messages, and a Foreground job, which receives them.
; NOTE: Each program should be assembled and linked separately.
;-
        .TITLE  EMWAIF
;+
; Foreground Program...
;-
        .MCALL  .RCVD,.MWAIT,.PRINT,.EXIT

.MACRO  ...
.ENDM   ...

MWAITF: .RCVD   #AREA,#MBUFF,#40. ;Request a message up to 80 char.
        ...                     ;No error possible - always a BG
                                ;Do some other processing
        .PRINT  #FGJOB          ;like announcing FG active...
        ...
        .MWAIT                  ;Wait for message to arrive...
        TST     MBUFF+2         ;Null message?
        BEQ     FEXIT           ;Yes...exit the program
        .PRINT  #FMSG           ;Announce we got the message...
        .PRINT  #MBUFF+2        ;and echo it back
        BR      MWAITF          ;Loop to get another one

FEXIT:  .EXIT                   ;Exit program

AREA:   .BLKW   5               ;EMT Argument Block
MBUFF:  .BLKW   41.             ;Buffer - Msg length + 1
        .WORD   0               ;Make sure 80 char message ends ASCIZ
```

```
FGJOB:  .ASCIZ  "!EMWAIF-I-FG running"
        .EVEN
FMSG:   .ASCIZ  "!EMWAIF-I-Message from BG:"
        .END    MWAITF

        .TITLE  EMWAIB;2
;+
; Background Program - Send a message or a'null' message
; to stop both programs.
;-
        .MCALL  .SDAT,.MWAIT,.GTLIN,.EXIT,.PRINT

MWAITB: CLR     BUFF                ;Clear 1st word
        .GTLIN  #BUFF,#PROMT        ;Get something to send to FG from TTY
        .SDAT   #AREA,#BUFF,#40. ;Send input as message to FG
        BCS     10$                 ;Branch on error - No FG
        .MWAIT                      ;Wait for message to be sent
        TST     BUFF                ;Sent a null message?
        BNE     MWAITB              ;No...loop to send another message.
        .EXIT                       ;Yes...exit program

10$:    .PRINT   #NOFG              ;No FG !
        .EXIT                       ;Exit program

AREA:   .BLKW   5                   ;EMT Argument Block
BUFF:   .BLKW   40.                 ;Up to 80 char message
PROMT:  .ASCIZ  "Enter message to be sent to FG job"
NOFG:   .ASCIZ  "?EMWAIB-F-No FG"
        .END    MWAITB
```

# .PEEK

**EMT 375, Code 34, Subcode 1**

The .PEEK programmed request accesses processor status and returns in R0 the contents of a specified low memory location (below 28K words) or I/O page location. .POKE deposits the value you specify into that low memory location (below 28K words) or I/O page location.

Use both requests to access and alter some contents of the processor status (PS) word. .PEEK and .POKE must be used in a mapped environment to change memory locations not defined as monitor fixed offsets, and should be used with all RT–11 monitors for compatibility.

.PEEK is very similar to .GVAL, but references locations differently. Addresses used by .PEEK are memory addresses. .GVAL accesses only monitor fixed offsets calculated relative to the base of the resident monitor. Although you can use .PEEK to access monitor fixed offsets, you have to find the base address of RMON, add the offset value, and use the resulting address as an argument to .PEEK. For information on valid bits, see .POKE.

Macro Call:

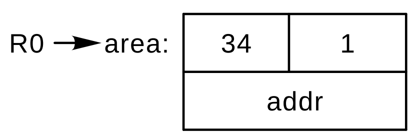
**.PEEK area,addr**

where:

**area** is the address of a two-word EMT argument block

**addr** is the address of the location to examine or change

Request Format:




Errors:

| **Code** | **Explanation** |
|---|---|
| 1 | Odd or nonexistent address<br>Error code is returned with carry bit set, when an attempt is made to access an odd or nonexistent address with .PEEK request. |

Example:

```
.TITLE  EPEEK

;Example of .PEEK and .POKE programmed requests.
;This example illustrates a way of reading and setting
;the default file size used by the .ENTER request.
;Normally, this would be done using the .GVAL and .PVAL programmed
;requests. (Refer to the example given for the .PVAL request.) This
;example computes the address of the word in RMON containing the
;default file size used by the .ENTER request and uses .POKE
;both to change the default file size to 100. blocks and to return
;the old default file size in R0.
;
        .MCALL  .PEEK,  .POKE,  .ENTER, .CLOSZ, .EXIT

        $RMON=: 54
        $MAXBL=: 314

START:  .PEEK   #EMTBLK,#$RMON  ;Pick up base of RMON from loc. 54
        ADD     #$MAXBL,R0      ;Add fixed offset of default file size,
        MOV     R0,R5
        .POKE   #EMTBLK,R5,NEWSIZ ;Set a new default file size, return old
        MOV     R0,OLDSIZ       ;default file size in R0, save it
        .ENTER  #EMTBLK,#0,#DBLK,#0 ;create a file of default size
        MOV     R0,R1           ;Save the size
        .CLOSZ  #EMTBLK,#0,R1   ;close the file
        .POKE   #EMTBLK,R5,OLDSIZ ;Restore previous default size
        .EXIT

EMTBLK: .BLKW   10.             ;EMT area
NEWSIZ: .WORD   100.
OLDSIZ: .WORD   0               ;The old default size is saved here.
DBLK:   .RAD50  "DK EPEEK TMP"

        .END    START
```

```
        .TITLE  EPEEK1.MAC

;+
;       The following is a subroutine that returns the current
;       PS contents (with undefined condition codes) in R0.
;
;       It can be used to determine the current MODE (K, S, or U)
;       the code is executing in.
;
;       If the top two bits are 000000 - K, 040000 - S, 140000 - U
;
;NOTE: This even works on processors w/o addressable PSs.
;-

        .MCALL  .PEEK

PS      =:      177776                  ;PS address

MYMODE::
        .PEEK   #AREA,#PS               ;Get PS Value into R0
        RETURN

AREA:   .BLKW   2                       ;Request area
```

# .POKE

**EMT 375, Code 34, Subcode 3, 5, 7**

.POKE deposits the value you specify into a low memory location (below 28K words) or I/O page location and returns the old contents of the memory location in R0. This simplifies the saving and restoring of a location. .POKE supports BIC (bit clear) and BIS (bit set) operations, as well as the previous MOV operation, using an optional parameter, *type*.

.POKE is very similar to .PVAL, but references locations differently. Addresses used by .POKE are memory addresses. .PVAL accesses only monitor fixed offsets calculated relative to the base of the resident monitor. Although you can use .POKE to access monitor fixed offsets, you have to find the base address of RMON, add the offset value, and use the resulting address as an argument to .POKE.

Macro Call:

**.POKE area,addr,value[,type]**

where:

**area** is the address of a three-word EMT argument block

**addr** is the address of the location to examine and change

**value** is the new contents to place in the location

**type** is the instruction used to modify the address to the specified value. The *type* parameter can be BIC, BIS, or MOV (default).

Request Format:

R0 → area:

| | |
|---|---|
| 34 | x |
| addr | |
| value | |

where:
x = 3 (MOV)
= 5 (BIC)
= 7 (BIS)

An attempt to access an odd or nonexistent address with a .POKE request returns the following error code with the carry bit set:

| Code | Explanation |
|---|---|
| 1 | Odd or nonexistent address |

The .POKE request can alter priority bits in the processor status (PS) word:

- A .POKE request returns the contents of the PS with undefined condition codes.
- .POKE can modify all bits in the PS except the 000020 (trace trap) and 140000 (current mode) bits. However, modifying the 000400 (instruction suspension) or 004000 (register set) bits can cause unexpected results. Although not prohibited, this is not recommended.

Modifying priority bits is supported. However, changing processor priority with a .POKE request automatically lowers processor priority to zero (PR0) during the time .POKE is executing. Therefore, a period of lowest processor priority exists between the time the processor is running at a given priority and the time the processor priority change takes effect.

Setting the carry bit is not recommended, as it causes .POKE to return an error.

- The priority portion of the modified PS is preserved at the completion of the .POKE request.

Example:
See .PEEK.

# .PRINT

**EMT 351**

The .PRINT request causes output to be printed at the console terminal.

Macro Call:

**.PRINT addr**

where:

**addr** is the address of the string to be 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 RETURN /LINE FEED combination. If a $200_8$ byte terminates the string, no RETURN/LINE FEED combination is generated.

Control returns to the user program after all characters have been placed in the output buffer.

When a foreground job is running and the job that is producing output changes, a B> or F> displays the location of the job. Any text display generated by a job is produced in the last indicated (background/foreground) until the next B> or F> display.

When a system job displays a message on the terminal, the message is preceded by logical-job-name.

If the foreground job issues a message using .PRINT, the message is printed immediately, no matter what the state of the background job. Therefore, for urgent messages, use the .PRINT request (rather than .TTYOUT 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 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. A foreground job can force its message through by doing a .PRINT instead of the .TTYOUT.

**NOTE**

Unlike other SYSMAC.SML definitions, .ENABL MCL will not work for .PRINT; therefore, to use .PRINT, you must do a .MCALL for it.

Errors:
None.

Example:
See .GTLIN.

# .PROTECT/.UNPROTECT

**.PROTECT: EMT 375, Code 31, Subcode 0**
**.UNPROTECT: EMT 375, Code 31, Subcode 1**

### .PROTECT

The .PROTECT request allows a job to obtain exclusive control of a vector (two words) in the area of 0 to 474. If the request is successful, it indicates that the locations are not currently in use by another job or by the monitor. The job then 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 two-word EMT argument block

**addr** is the address of the word pair to be protected

**NOTE**

The value of the *addr* argument must be a multiple of four, and must be less than or equal to $474_8$. (That is, the argument is the address of a word containing a value that is a multiple of four, but the address itself is not.) 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 *addr* is greater than $474_8$ or is not a multiple of 4. |

Example:

```
        .TITLE  EPROTE;2
;+
; .PROTECT / .UNPROTECT - This is an example in the use of the .PROTECT
; and .UNPROTECT requests. The example illustrates how to protect the
; vectors of a device while an inline interrupt service routine does
; a data transfer (in this case the device is a DL11 Serial Line
;Interface).
; When the program is finished, the vectors are unprotected for
;possible use by another job.
;-
        .MCALL  .DEVICE,.EXIT,.PROTECT,.UNPROTECT,.PRINT

.MACRO  ...
.ENDM   ...

        .GLOBL  DL11

START:  .DEVICE #AREA,#LIST     ;Setup to disable DL11 interrupts on
                                ;.EXIT or ^C^C
        .PROTECT #AREA,#300     ;Protect the DL11 vectors
        BCS     BUSY            ;Branch if already protected
        ...                     ;Set up data to transmit over DL11
        JSR     R5,DL11         ;Use DL11 xfer routine (see .INTEN
                                ;example)
        .WORD   128.            ;Arguments...Word count
        .WORD   BUFFR           ;Data buffer addr
        ...                     ;Continue processing...
FINI:   .UNPROTECT #AREA,#300   ;...eventually to exit program
        .EXIT

BUSY:   .PRINT  #NOVEC          ;Print error message...
        .EXIT                   ;then exit

AREA:   .BLKW   3               ;EMT Argument block

LIST:   .WORD   176500          ;CSR of DL11
        .WORD   0               ;Stuff it with '0'
        .WORD   0               ;List terminator

BUFFR:                          ;Data to send over DL11
        .REPT   8.              ;8 lines of 32 characters...
        .ASCII  "Hello DL11... Are You There ??"
        .BYTE   15,12
        .ENDR

NOVEC:  .ASCIZ  "?EPROTE-F-Vector already protected"

        .END    START
```

### .UNPROTECT

The .UNPROTECT request is the complement of the .PROTECT request. It cancels any protected vectors in the 0 to $474_8$ area. An attempt to unprotect a vector that a job has not protected is ignored.

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 canceled. The argument *addr* must be a multiple of four, and must be less than or equal to $474_8$ |

Request Format:

| | | |
|---|---|---|
| R0 → area: | 31 | 1 |
| | addr | |

Errors:

| Code | Explanation |
|---|---|
| 1 | Address (*addr*) is greater than $474_8$ or is not a multiple of four. |

Example:
See .PROTECT.

# .PURGE

**EMT 374, Code 3**

The .PURGE request makes a channel available, unlike .HRESET or .SRESET which affects all channels for a job, .SAVESTATUS, .CLOSE or .CLOSZ which make a .ENTERed channel permanent. .PURGE frees a channel without taking any other action. If a tentative file has been entered on the channel, the file is discarded. An attempt to purge a channel that is not open is ignored. When a program makes a channel available by issuing a .PURGE request, the handler for the device associated with that channel must now be in memory if the handler is marked SPECL$ (supports a special directory structure).

**NOTE**

Do not purge channel $17_8$ if your program is overlaid because overlays are read on that channel.

Macro Call:

**.PURGE chan**

where:

**chan** is the number of the channel to be made available

Request Format:

| | | |
|---|---|---|
| R0 = | 3 | chan |

Errors:
None.

Example:
See .SAVESTATUS.

# .PVAL

See .GVAL/.PVAL.

# .QELDF

**Macro Expansion**

The .QELDF macro symbolically defines queue element offsets.

Since the handler usually deals with queue element offsets relative to Q.BLKN, the .QELDF macro also defines these associated symbolic offsets, using a $ character substituted for the period (.).

Macro Call:

**.QDELF list, E**

In the following example, if memory address conditional (MMG$T) equals 1 (default), additional offsets are generated for use only for mapped monitors. The length of queue element (Q.ELGH) is controlled by the MMG$T setting.

```
Normal Offset   Handler Offset         Description
-------------   --------------         ------------
Q.LINK=0        Q$LINK=Q.LINK-^o4      Link to next queue element
Q.CSW=2.        Q$CSW=Q.CSW-^o4        Pointer to channel status word
Q.BLKN=4.       Q$BLKN=Q.BLKN-^o4      Physical block number
Q.FUNC=6.       Q$FUNC=Q.FUNC-^o4      Special function code
Q.2UNI=Q.FUNC   Q$2UNI=Q.2UNI-^o4      High 3 bits of unit number(3)
Q.TYPE=Q.FUNC   Q$TYPE=Q.TYPE-^o4      Normal I/O or special function flag(3)
Q.FMSK=^o17                            Special function bits mask(3)
Q.2MSK=^o160                           High 3 bits of unit number mask(3)
Q.TMSK=^o200                           Normal or special flag mask(3)
Q.JNUM=7.       Q$JNUM=Q.JNUM-^o4      Job number
Q.UNIT=Q.JNUM   Q$UNIT=Q.UNIT-^o4      Low (or all) bits of unit number
Q.UMSK=^o3400                          Unit number mask
Q.JMSK=^o74000                         Job number mask
Q.BUFF=^o10     Q$BUFF=Q.BUFF-^o4      Buffer physical address
Q.WCNT=^o12     Q$WCNT=Q.WCNT-^o4      Transfer count
Q.COMP=^o14     Q$COMP=Q.COMP-^o4      Completion routine or I/O type flag

Q.PAR=^o16(2)   Q$PAR=Q.PAR-^o4(2)     DMA PAR1 base address
Q.MEM=^o20(2)   Q$MEM=Q.MEM-^o4(2)     CPU access PAR1 base address

Q.ELGH=^o16(1)                         Length of queue element
Q.ELGH=^o24(2)                         Length of queue element

(1) unmapped system (MMG$T=0)
(2) mapped system (MMG$T=1)
(3) extended unit handlers and systems only
```

# .QSET

**EMT 353**

.QSET request enables additional entries to the RT–11 I/O free queue.

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 unmapped monitors, each queue entry is seven words long; hence the space set aside for the queue should be *len*7* words. In the mapped monitors, $10_{10}$ words per queue element are required. (For compatibility with all monitors, use $10_{10}$ words.)

Generally, each program should require 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 .MRKT, .TWAIT, .SDAT/C, and .RCVD/C also require queue elements and must be considered when allocating queue elements for a program. On completion, R0 contains the address of the first word beyond the allocated queue elements. (Note that if synchronous I/O is done, such as .READW /.WRITW, and no timing requests are done, no additional queue elements need be allocated.)

The following programmed requests require queue elements:

| | | | | |
|---|---|---|---|---|
| .MRKT | .READW | .RCVDW | .WRITW | .SDATW |
| .READ | .RCVD | .WRITC | .SDAT | |
| .READC | .RCVDC | .WRITE | .SDATC | |

Each time .QSET is called, a specified contiguous area of memory is divided into seven-word segments (10-$\text{word}_{10}$ for the mapped monitors) 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 your program space. Make sure the program in no way alters the elements after they are set up. The .SRESET and .HRESET requests discard all user-defined queue elements; therefore, any previous .QSET requests must be reissued. However, you must not specify the same space in two separate .QSET requests if there has been no intervening .SRESET or .HRESET request.

Be sure 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. The monitor returns the address of the first unused word beyond the queue elements. 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 mapped monitor, queue elements must be allocated in the lower 28K words of memory, since they must be accessible in Kernel mapping. In addition, the elements must not be in the virtual address space mapped by Kernel PAR1, specifically the area from 20000 to $37776_8$.

**NOTE**

Programs that are to run in mapped monitor as well as multijob environments should allocate $10_{10}$ words for each queue element. Alternatively, a program can specify the start of a large area and use the returned value in R0 as the top of the queue element.

Errors:

In an extended memory environment, an attempt to violate the PAR1 restriction results in a *?MON-F-addr* error, which can be intercepted with a .SERR programmed request.

Example:
See .MRKT.

# .RCTRLO

**EMT 355**

The .RCTRLO request resets the CTRL/O flag for the terminal. A CTRL/O typed while output is directed to the terminal causes output to be discarded until either another CTRL/O is typed or the program resets the CTRL/O flag. Therefore, a program with a message that must appear at the terminal should reset the CTRL/O switch.

A program must issue a .RCTRLO request whenever it changes the contents of the job status word (JSW). Issuing a .RCTRLO request updates the monitor's internal status information to reflect the current contents of the JSW.

Macro Call:

**.RCTRLO**

Errors:

**WARNING**

If the terminal is set to XOFF (no scroll or hold session), .RCTRLO will not clear the XOFF condition.

Example:

```
        .TITLE  ERCTRL
;+
; .RCTRLO - This is an example in the use of the .RCTRLO request.
; 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 assured via the .RCTRLO request prior to the
; CSI call.
;-
        .MCALL  .RCTRLO,.CSIGEN

START:  .RCTRLO                         ;Make sure TT: output is enabled
        .CSIGEN #DSPACE,#DEXT,#0        ;Issue a .CSIGEN request to get
                                        ;command
                                        ;(CSI will prompt with '*')
        ;       ...                     ;Process the command...
        JMP     START                   ;Get another command...
DEXT:   .WORD   0,0,0,0                 ;No default extensions

DSPACE =:.                              ;Space for handlers starts here

        .END    START
```

# .RCVD/.RCVDC/.RCVDW

**EMT 375, Code 26**

The .RCVD (receive data) request allows a job to read messages or data sent by another job in a multijob environment.

Three forms of the .RCVD request are used with the .SDAT (send data) request. The send-data receive-data request combination provides 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 the .QSET request). Under a system job monitor, .RCVD requests and .SDAT requests remain valid for sending messages between background and foreground jobs in addition to the general read and write capability available to all jobs provided by MQ.

Be particularly careful if you use both synchronous (.RCVDW and .SDATW) and asynchronous (.RCVDC and .SDATC) requests in the same program. If you issue a mainline .SDATW while there is a pending .RCVDC, the .SDATW will wait until the .RCVDC is satisfied. If the completion routine for the .RCVDC issues another .RCVDC, the mainline .SDATW will never complete. In general, you should avoid the use of both synchronous and asynchronous message requests in the same program.

**.RCVD**

The request is posted and the issuing job continues execution. When the job needs to have the transmitted message, executing .MWAIT suspends the job until all .SDATx and .RCVDx requests for the job are complete.

Macro Call:

**.RCVD area,buf,wcnt[,BMODE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **buf** | is the address of the buffer into which the message length and message data are to be placed |
| **wcnt** | is the number of words in the buffer |

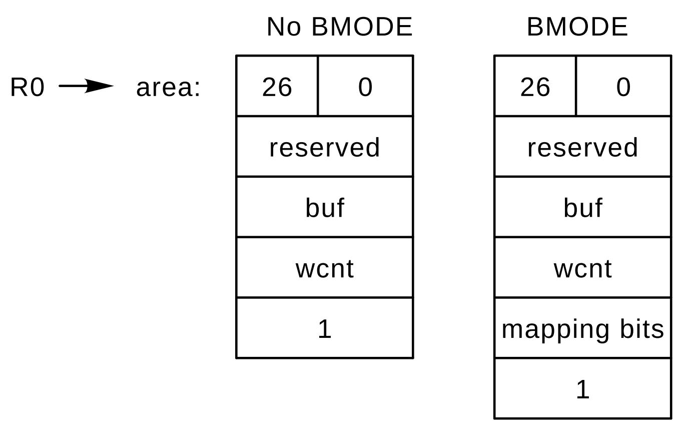
**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:




Upon completion of the .MWAIT, the first word of the message buffer contains the number of words sent. Thus, the space allocated for the message should always be at least one word larger than the actual message size expected. If the sending job attempts to send more words than the receiver specified in the *wcnt* argument of the .RCVD request, the first word of the buffer will contain the number of words that the sender specified, but only *wcnt* words will be actually transferred. The rest of the sender's message will be ignored.

Because *wcnt* (word count) is a variable number, the .SDAT/.RCVD combination can transmit a few words or entire buffers. The data transfer can only complete when a .SDATx is issued by 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. Messages are all processed in first-in first-out order. Thus, the receiver must be certain it is receiving the message it actually wants. Message handling does not check for a word count of zero before queuing a send or receive data request. Since RT–11 distinguishes a send from a receive by complementing the word count, a .SDAT of zero words is treated as a .RCVD of zero words. Avoid a word count of zero at all times when using a .RCVD request.

Errors:

| Code | Explanation |
|---|---|
| 0 | No other job exists in the system. (A job exists as long as it is loaded, whether or not it is active.) |

Example:
Refer to the example for the .SDAT request.

### .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[,BMODE=strg][,CMODE=strg]**

where:

**area** is the address of a five-word EMT argument block

**buf** is the address of the buffer into which the message length /message data is to be placed

**wcnt** is the number of words in the buffer

**crtn** is the address of a completion routine to be entered

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

**CMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| U | User space (default) |
| S | Supervisor space |

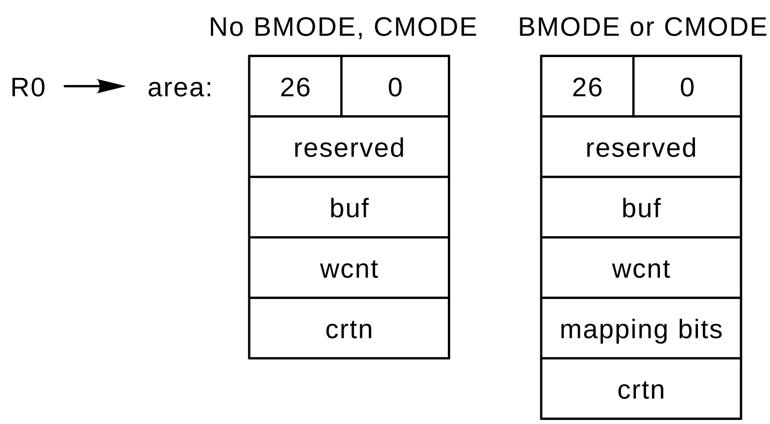
Specifying *CMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *CMODE*.
- Specifies the *space* for the *crtn* argument.
- Is a valid option only for fully mapped monitors.

Note that, if both BMODE and CMODE are specified, only one additional word can be added that contains both flags.

As in the .RCVD request, word 0 of the buffer contains the number of words transmitted when the transfer is complete.

Request Format:




Errors:

| Code | Explanation |
|---|---|
| 0 | No other job exists in the system. (A job exists as long as it is loaded, whether or not it is active.) |

Example:

```
        .TITLE  EREADC;2
;+
; .READC / .WRITC - This is an example in the use of the .READC /
; .WRITC requests. The example demonstrates event-driven I/O where
; a mainline program initiates a file transfer and completion routines
; continue it while the mainline proceeds with other processes. The
; example is another single file copy program, utilizing .CSIGEN to
; input the file specs, load the required handlers and open the files.
;-
        .MCALL  .READC,.WRITC,.CLOSE,.PRINT
        .MCALL  .CSIGEN,.EXIT,.WAIT,.SRESET
        .MCALL  .QSET

        $ERRBY  =:      52                  ;(.SYCDF)Error Byte in SYSCOM

        .ENABL  LSB
START:  MOV     SP,R5                       ;Save SP, since .C>SIGEN changes it
        .CSIGEN #DSPACE,#DEFEXT             ;Use CSIGEN to get handlers, files
        MOV     R5,SP                       ;Restore SP
        .QSET   #QUEUE,#2                   ;Add some queue elements
        CALL    IOXFER                      ;Start I/O
        .PRINT  #MESSG                      ;Now simulate other mainline process
        MOV     #-1,R5                      ;
10$:    DEC     R5                          ; (kill some time)
        BNE     10$                         ;
        TSTB    EOF                         ;Did I/O complete?
        BEQ     10$                         ;No...do some more mainline work
        INCB    EOF                         ;Check for read/write error
        BEQ     WERR                        ;EOF = 0 = Write error
        BLT     RERR                        ;EOF .lt. 0 = Read error
        .CLOSE  #0                          ;EOF > 0 = End of File
        MOV     #DONE,R0                    ;R0 => We're done messg
        BR      GBYE                        ;Merge to exit program

WERR:   MOV     #WRERR,R0                   ;Set up error messages here...
        BR      GBYE

RERR:   MOV     #RDERR,R0
GBYE:   .PRINT                              ;Print message
        .SRESET                             ;Dismiss fetched handlers
        .EXIT                               ;Exit program

WRDONE: .WAIT   #0                          ;Write compl rtne...write successful?
        BCS     30$                         ;Branch if not...
IOXFER: .READC  #AREA,#3,,,#RDDONE          ;Queue up a read
        BCC     60$                         ;Branch if ok...
        TSTB    @#$ERRBY                    ;Error - is it EOF?
        BEQ     50$                         ;Branch if yes
20$:    DECB    EOF                         ;User EOF Flag to indicate hard error
30$:    DECB    EOF                         ;EOF = -2 Read err / = -1 Write err
        RETURN                              ;Leave completion code
RDDONE: .WAIT   #3                          ;Compl rtne #2 - was read ok?
        BCS     20$                         ;Branch if not
        .WRITC  #AREA,#0,,,#WRDONE          ;Queue up a write...
        BCS     30$                         ;Branch if error
40$:    INC     BLOCK                       ;Bump block # for next read
        RETURN                              ;Leave Completion code...

50$:    INCB    EOF                         ;Set EOF flag
60$:    RETURN                              ;then return
```

```
AREA::  .WORD   0                       ;EMT Area block
BLOCK:  .WORD   0                       ;Block #,
        .WORD   BUFF                    ;Buffer addr & word count
        .WORD   256.                    ;already fixed in block...
        .WORD   0                       ;Completion rtne addr

BUFF:   .BLKW   256.                    ;I/O buffer

DEFEXT: .WORD   0,0,0,0                 ;No default extensions for CSIGEN

QUEUE:  .BLKW   2*10.                   ;Extra queue elements

EOF:    .BYTE   0                       ;EOF flag

DONE:   .ASCIZ  "!EREADC-I-I/O Transfer Complete"
MESSG:  .ASCIZ  "!EREADC-I-Simulating Mainline Processing"
WRERR:  .ASCIZ  "?EREADC-I-Write Error"
RDERR:  .ASCIZ  "?EREADC-I-Read Error"
        .EVEN
DSPACE  = .                             ;Handlers may be loaded starting here
        .END    START
```

### .RCVDW

A message request is posted and the job issuing the request is suspended until all pending .SDATx and .RCVDx requests for the job are complete. 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,[,BMODE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **buf** | is the address of the buffer into which the message length /message data is to be placed |
| **wcnt** | is the number of words to be transmitted |

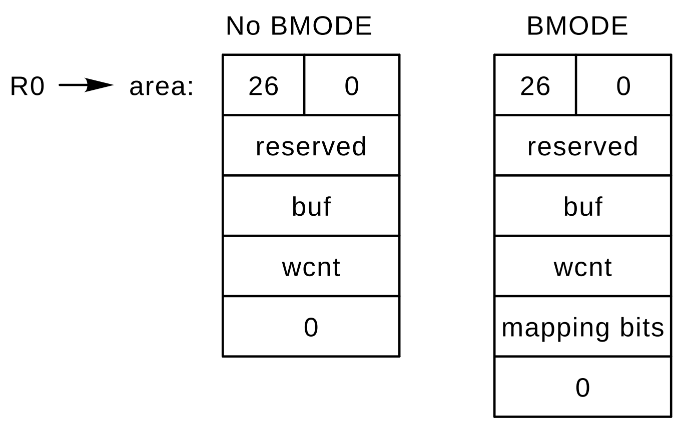
**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:




Errors:

| Code | Explanation |
|---|---|
| 0 | No other job exists in the system. (A job exists as long as it is loaded, whether or not it is active.) |

Example:
See .SDATW.

# .RDBBK

### Macro Expansion

The .RDBBK macro defines symbols for the region definition block and reserves space for it. The .RDBBK automatically invokes .RDBDF.

Use optional fourth parameter, *BASE=n*, to explicitly assign a base address to a global region. Because the BASE=n parameter is optional, you do not need to modify existing programs unless you want this new functionality. The .RDBBK macro has the following syntax:

Macro Call:

**.RDBBK rgsiz,rgsta,name[,BASE=n]**

where:

| | |
|---|---|
| **rgsiz** | is the size of the dynamic region needed (expressed in 32-word units) |
| **rgsta** | is the region status byte |
| **name** | is the name of the global region |
| **BASE=n** | specifies the starting address of the region, expressed in 32-word units. A value of 0 (or value omitted) means any available base address is acceptable. |

Errors:
None.

Example:

See .CRAW. See also the *RT–11 System Internals Manual* for a detailed description of the extended memory feature.

# .RDBDF

## Macro Expansion

The .RDBDF macro defines the symbolic offset names for the region definition block and the names for the region status word bit patterns. This macro also defines the length of the region definition block, but it does not reserve space for the region definition block.

Macro Call:

**.RDBDF**

The .RDBDF macro expands as follows:

```
R.GID =: 0              ;Region ID
R.GSIZ=: 2.             ;Size in chunks
R.GSTS=: 4.             ;Status
R.GLLN=: 6.             ;Length of simple RDB
R.GNAM=: 6.             ;Name of global region (optional)
R.GBAS=: 10.            ;Base physical address (optional)
R.GLGH=: 12.            ;Length of complex RDB
RS.CRR=: ^o100000       ;Region created successfully
RS.UNM=: ^o40000        ;One or more windows eliminated
RS.NAL=: ^o20000        ;Region newly allocated
RS.NEW=: ^o10000        ;New global region
RS.GBL=: ^o4000         ;Create within a global region
RS.CGR=: ^o2000         ;Create new global region if not found
RS.AGE=: ^o1000         ;Enable auto elimination of region
RS.EGR=: ^o400          ;Eliminate global region
RS.EXI=: ^o200          ;Eliminate global region on exit / abort
RS.CAC=: ^o100          ;Bypass memory cache
RS.BAS=: ^o40           ;R.GBAS supplied
RS.NSM=: ^o20           ;Non-system memory
RS.DSP=: ^o2            ;reserved
RS.PVT=: ^o1            ;reserved
```

# .READ/.READC/.READW

**EMT 375, Code 10**

Read operations for the three modes of RT–11 I/O use the .READ, .READC, and READW programmed requests.

Be particularly careful if you use both synchronous .READW and .SDATW and asynchronous .READC requests in the same channel. If you issue a mainline ..READW while there is a pending .READC, the .READW will wait until the .READC is satisfied. If the completion routine for the .READC issues another .READC, the mainline .READW will never complete. In general, you should avoid the use of both synchronous and asynchronous message requests in the same program.

For .READ and .READC, additional queue elements should be allocated for queued I/O operations (See the .QSET request).

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. If the read is from a random-access device (disk), R0 contains the actual number of words that will be read (.READ or .READC) or have been read (.READW), provided no error is reported. This number is less than the requested word count if an attempt is made to read past end-of-file, but a partial transfer of one or more blocks 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 $512_{10}$ 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 and shortens the request to $256_{10}$ words. On return from the request, R0 contains 256, indicating that a partial transfer occurred. Also, 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 starting 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, and R0 contains a zero.

**.READ**

The .READ request transfers to memory a specified number of words from the device associated with the specified channel. 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[,BMODE=strg]**

where:

**area** is the address of a five- or six-word EMT argument block

**chan** is a channel number in the range 0-$376_8$

**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. Note the first block of a file or device is block number 0. The user program normally updates *blk* before it is used again. If input is from TT: and *blk=0*, TT: issues an up-arrow (^) prompt (This is true for all .READ* requests.)

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

**Notes**

.READ and .READC requests instruct the monitor to do a read from the device by queuing a request for the device, then immediately returning control to your program.

In general the order in which I/O requests are completed is not guaranteed by the operating system. A direct access device in a system which has the UB handler active, the UB handler may reorder requests to optimize the use of unibus mapping registers. In a system without UB active, requests within a job are handled on a FIFO basis; but requests between jobs are done on a priority basis. The handler for a nondirect access device should allow only one job to attach to a unit and the handler should be marked as requiring serialization to preclude UB from reordering operations.

Under certain circumstances, completion routines can be entered even if the .READC request returns an error. Multiple asynchronous read request to a device can cause a completion routine to be entered even when .READC returns an error. A high speed device can return an error during the short window existing between two sections of hardware processing code, the first of which checks the previous request, the second of which checks the .READC request. The completion routine can be entered in the interim before .READC returns for the second error check. Therefore, you should exercise care when making multiple asynchronous read requests to a device if any of the requests call a completion routine.

Read errors are returned 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 (.READ/BCS) and after the wait (.WAIT/BCS). The wait request returns an error, but it does not indicate which read caused the error.

Errors reported on the return from the read request are as follows:

- Nonexistent device/unit
- Nonexistent block
- In general, errors that do not require data transfers but are controller errors or EOF errors

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 value 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.

If .READ/C/W requests are used to receive messages under a system job monitor, the buffer must be one word longer than the number of words expected to be read. Upon completion of the data transfer, the first word of the buffer will contain a value equal to the number of words actually transferred (as for .RCVD/C/W).

Request Format:

R0 → area:

| No BMODE | |
|---|---|
| 10 | chan |
| blk | |
| buf | |
| wcnt | |
| 1 | |

| BMODE | |
|---|---|
| 10 | chan |
| blk | |
| buf | |
| wcnt | |
| mapping bits | |
| 1 | |

When the user program needs to access the data read on the specified channel, issue a .WAIT request as a check that the data has been read completely. If an error occurred during the transfer, the .WAIT request indicates the error.

The handler for nondirect-access devices should allow only one job to attach to a given unit. The handler should be marked, as requiring serialization to prevent UB from reordering operations.

Errors:

| Code | Explanation |
|---|---|
| 0 | Attempt to read past end-of-file. |
| 1 | Hard error occurred on channel. |
| 2 | Channel is not open. |

Example:

```
        .TITLE  EREAD
;+
; .READ / .WRITE - This is an example in the use of the .READ / .WRITE
; requests. The example demonstrates asynchronous I/O where a mainline
; program initiates input via .READ requests, does some other processing
; makes sure input has completed via the .WAIT request, then outputs
; the block just read. Another .WAIT is issued before the next read
; is issued to make sure the previous write has finished. This example
; is another single file copy program, utilizing .CSIGEN to input the
; file specs, load the required handlers and open the files.
;-
        .MCALL  .READ,.WRITE,.CLOSE,.PRINT
        .MCALL  .CSIGEN,.EXIT,.WAIT,.SRESET

        $ERRBY  =: 52                   ;(.SYCDF)Error Byte in SYSCOM
```

```
        .ENABL  LSB                     ;Enable local symbol block
START:  .CSIGEN #DSPACE,#DEFEXT         ;Use CSIGEN to get handlers, files
        MOV     #AREA,R5                ;R5 => EMT Argument list
        CLR     IOBLK                   ;Start reads with Block #0
1$:     .READ   R5,#3                   ;Read a block...
        BCS     6$                      ;Branch on error
        ;       .                       ;Then simulate
        BIT     #1,IOBLK                ;some other
        BNE     2$                      ;meaningful(?)
        .PRINT  #MESSG                  ;process...
        ;       .
2$:     .WAIT   #3                      ;Did read finish OK?
        BCS     5$                      ;Branch if not (must be hard error!)
        .WRITE  R5,#0                   ;Now write the block just read
        BCS     3$                      ;Branch on error
        INC     IOBLK                   ;Bump Block #
        ;       .                       ;We could do some more processing here
        .WAIT   #0                      ;Wait for write to finish
        BCC     1$                      ;Branch if write was successful
3$:     MOV     #WERR,R0                ;R0 => Write error msg
4$:     .PRINT                          ;Report error
        BR      7$                      ;Merge to exit program
5$:     MOV     #RERR,R0                ;R0 => Read error msg
        BR      4$                      ;Branch to report error
6$:     TSTB    @#$ERRBY                ;Read error...EOF?
        BNE     5$                      ;Branch if not
        .PRINT  #DONE                   ;Yes...announce completion
        .CLOSE  #0                      ;Make output file permanent
7$:     .SRESET                         ;Dismiss fetched handlers
        .EXIT                           ;then exit program

AREA::  .WORD   0                       ;EMT Area block
IOBLK:  .WORD   0                       ;Block #,
        .WORD   BUFF                    ;Buffer addr & word count
        .WORD   256.                    ;already fixed in block...
        .WORD   0                       ;nowait type I/O

BUFF:   .BLKW   256.                    ;I/O buffer

DEFEXT: .WORD   0,0,0,0                 ;No default extensions for CSIGEN

DONE:   .ASCIZ  "!EREAD-I-I/O Transfer Complete"
MESSG:  .ASCIZ  "!EREAD-I-Simulating Mainline Processing"
WERR:   .ASCIZ  "?EREAD-F-Write Error"
RERR:   .ASCIZ  "?EREAD-F-Read Error"
        .EVEN
DSPACE:                         ;Handlers may be loaded starting here

        .END    START
```

### .READC

The .READC request transfers to memory a specified number of words from the device associated with the specified channel. 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 a RETURN is executed in the completion routine, control returns to the user program.

Macro Call:

**.READC area,chan,buf,wcnt,crtn,blk[,BMODE][,CMODE]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **chan** | is a channel number in the range 0-$376_8$ |
| **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 $500_8$ |
| **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 |
| **BMODE = strg** | where *strg* is: |

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

**CMODE = strg** where *strg* is:

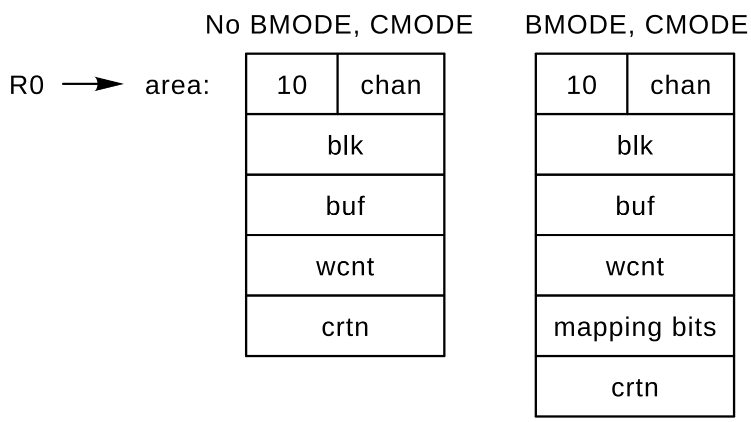
| Value | Description |
|---|---|
| U | User space (default) |
| S | Supervisor space |

Specifying *CMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *CMODE*.
- Specifies the *space* for the *crtn* argument.
- Is a valid option only for fully mapped monitors.

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. The completion routine will never be entered if the .READC request returns an error.

Request Format:




When a .READC completion routine is entered, the following conditions are true:

- R0 contains the contents of the channel status word for the operation. If HDERR$ of R0 is set, a hardware error occurred during the transfer; consequently, the data may not be reliable. The end-of-file bit, EOF$ may be set.
- R1 contains the channel number of the operation. This is useful when the same completion routine is to be used for transfers on different channels.
- On a file-structured transfer, a shortened read is reported when the .READC request is returned, not when the completion routine is called.

- 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:

```
        .TITLE  EREADC;2
;+
; .READC / .WRITC - This is an example in the use of the .READC /
; .WRITC requests. The example demonstrates event-driven I/O where
; a mainline program initiates a file transfer and completion routines
; continue it while the mainline proceeds with other processes. The
; example is another single file copy program, utilizing .CSIGEN to
; input the file specs, load the required handlers and open the files.
;-
        .MCALL  .READC,.WRITC,.CLOSE,.PRINT
        .MCALL  .CSIGEN,.EXIT,.WAIT,.SRESET
        .MCALL  .QSET

        $ERRBY  =:      52        ;(.SYCDF)Error Byte in SYSCOM

        .ENABL  LSB
START:  MOV     SP,R5             ;Save SP, since .C>SIGEN changes it
        .CSIGEN #DSPACE,#DEFEXT   ;Use CSIGEN to get handlers, files
        MOV     R5,SP             ;Restore SP
        .QSET   #QUEUE,#2         ;Add some queue elements
        CALL    IOXFER            ;Start I/O
        .PRINT  #MESSG            ;Now simulate other mainline process
        MOV     #-1,R5            ;
10$:    DEC     R5                ; (kill some time)
        BNE     10$               ;
        TSTB    EOF               ;Did I/O complete?
        BEQ     10$               ;No...do some more mainline work
        INCB    EOF               ;Check for read/write error
        BEQ     WERR              ;EOF = 0 = Write error
        BLT     RERR              ;EOF .lt. 0 = Read error
        .CLOSE  #0                ;EOF > 0 = End of File
        MOV     #DONE,R0          ;R0 => We're done messg
        BR      GBYE              ;Merge to exit program
WERR:   MOV     #WRERR,R0         ;Set up error messages here...
        BR      GBYE

RERR:   MOV     #RDERR,R0
GBYE:   .PRINT                    ;Print message
        .SRESET                   ;Dismiss fetched handlers
        .EXIT                     ;Exit program
```

```
WRDONE: .WAIT   #0                 ;Write compl rtne...write successful?
        BCS     30$                ;Branch if not...
IOXFER: .READC  #AREA,#3,,,#RDDONE ;Queue up a read
        BCC     60$                ;Branch if ok...
        TSTB    @#$ERRBY           ;Error - is it EOF?
        BEQ     50$                ;Branch if yes
20$:    DECB    EOF                ;User EOF Flag to indicate hard error
30$:    DECB    EOF                ;EOF = -2 Read err / = -1 Write err
        RETURN                     ;Leave completion code
RDDONE: .WAIT   #3                 ;Compl rtne #2 - was read ok?
        BCS     20$                ;Branch if not
        .WRITC  #AREA,#0,,,#WRDONE  ;Queue up a write...
        BCS     30$                ;Branch if error
40$:    INC     BLOCK              ;Bump block # for next read
        RETURN                     ;Leave Completion code...

50$:    INCB    EOF                ;Set EOF flag
60$:    RETURN                     ;then return

AREA::  .WORD   0                  ;EMT Area block
BLOCK:  .WORD   0                  ;Block #,
        .WORD   BUFF               ;Buffer addr & word count
        .WORD   256.               ;already fixed in block...
        .WORD   0                  ;Completion rtne addr

BUFF:   .BLKW   256.               ;I/O buffer

DEFEXT: .WORD   0,0,0,0            ;No default extensions for CSIGEN

QUEUE:  .BLKW   2*10.              ;Extra queue elements

EOF:    .BYTE   0                  ;EOF flag

DONE:   .ASCIZ  "!EREADC-I-I/O Transfer Complete"
MESSG:  .ASCIZ  "!EREADC-I-Simulating Mainline Processing"
WRERR:  .ASCIZ  "?EREADC-I-Write Error"
RDERR:  .ASCIZ  "?EREADC-I-Read Error"
        .EVEN
DSPACE  = .                 ;Handlers may be loaded starting here
        .END    START
```

### .READW

The .READW request transfers to memory a specified number of words from the device associated with the specified channel. When the .READW is complete or an error is detected, control returns to the user program.

Macro Call:

**.READW area,chan,buf,wcnt,blk[,BMODE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **chan** | is a channel number in the range 0-$376_8$ |
| **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

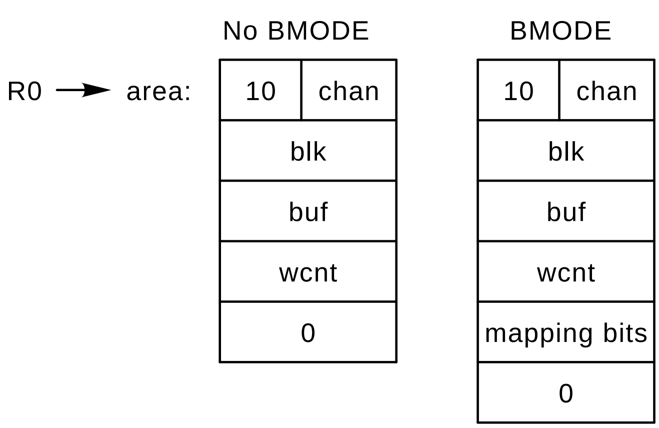
**BMODE=strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:




If no error occurred, the data is in memory at the specified address. In a multijob environment, the other job can be run while the issuing job is waiting for the I/O to complete. If a volume is opened with a non-file-structured lookup and the word count specified is greater than the number of words left on the volume, .READW returns a hard error.

Errors:

| Code | Explanation |
|---|---|
| 0 | Attempt to read past end-of-file. |
| 1 | Hard error occurred on channel. |
| 2 | Channel is not open. |

Example:

```
        .TITLE  EREADW
; .READW / .WRITW - This is an example in the use of the .READW / .WRITW
; requests. The example is a single file copy program. The file specs
; are input from the console terminal, and the input & output files opened
; via the general mode of the CSI. The file is copied using synchronous
; I/O, and the output file is made permanent via the .CLOSE request.
        .MCALL  .CSIGEN,.READW,.PRINT,.EXIT,.WRITW,.CLOSE,.SRESET
        $ERRBY=:52              ;(.SYCDF) Error Byte Location
START:  .CSIGEN #DSPACE,#DEXT   ;Get string from terminal
        CLR     IOBLK           ;Input block # starts with 0
        MOV     #AREA,R5        ;R5 => EMT Argument list
READ:   .READW  R5,#3           ;Read a block on Channel 3
                                ;Blk#, Buff addr & WC already in arg
        BCC     20$             ;Branch if no errors
        TSTB    @#$ERRBY        ;Is error EOF?
        BEQ     30$             ;Yes...
        MOV     #RERR,R0        ;R0 => Read Error Message
10$:    .PRINT                  ;Print the message
        BR      40$             ;Exit program
20$:    .WRITW  R5,#0           ;Write the block just read
        INC     IOBLK           ;Bump block # (doesn't affect C bit)
        BCC     READ            ;Branch if no error
        MOV     #WERR,R0        ;R0 => Write error message
        BR      10$             ;Branch to output the message
30$:    .CLOSE  #0              ;End-of-File...Close output channel
        .PRINT  #DONE           ;Announce successful copy
40$:    .SRESET                 ;Release handler(s) from memory
        .EXIT                   ;Exit the program
DEXT:   .WORD   0,0,0,0         ;No default extensions
AREA:   .WORD   0               ;EMT Argument block
IOBLK:  .WORD   0               ;Block #
        .WORD   BUFFR           ;I/O Buffer addr
        .WORD   256.            ;Word Count
        .WORD   0               ;
BUFFR:  .BLKW   256.            ;I/O Buffer
RERR:   .ASCIZ  "?EREADW-F-Read error"
WERR:   .ASCIZ  "?EREADW-F-Write error"
DONE:   .ASCIZ  "!EREADW-I-I/O Transfer Complete"
        .EVEN
DSPACE =:.              ;Handler(s) can be loaded starting here
        .END    START
```

# .RELEAS

See .FETCH/.RELEAS.

# .RENAME

**EMT 375, Code 4**

The .RENAME request changes the name of the file specified.

Macro Call:

**.RENAME area,chan,dblk**

where:

**area** is the address of a two-word EMT argument block

**chan** is an available channel number in the range 0-$376_8$

**dblk** is the address of a block that specifies the file to be renamed followed by the new file name

Request Format:

R0 → area:

| | |
|---|---|
| 4 | chan |
| dblk | |

The *dblk* argument consists of two consecutive Radix–50 device and file specifications. For example:

```
        .TITLE  ERENA1

        .RENAME #AREA,#7,#DBLK  ;Rename using chan 7
        BCS     RENERR          ;failed
        BR      RENOK           ;success

DBLK:   .RAD50  "DK "           ;old file name
        .RAD50  "ERENA1"
        .RAD50  "TMP"
        .RAD50  "DK "           ;new file name
        .RAD50  "ZRENA1"
        .RAD50  "TMP"
```

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 logical name SRC 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 valid only on files on block-replaceable devices (disks). In magtape operations, the handler returns an invalid operation code in byte 52 if a .RENAME request is attempted. A .RENAME request to other devices is ignored.

Files cannot be protected or unprotected using the .RENAME request. To change the protection status of a file, use the .FPROT or .SFSTA requests or the PROTECT and UNPROTECT commands.

File dates can be changed using the .SFDAT request.

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel not available. |
| 1 | File not found. |
| 2 | Invalid operation. |
| 3 | A file by that name already exists and is protected. A .RENAME was not done. |

Example:

```
        .TITLE  ERENAM;2
; .RENAME - This is an example in the use of the .RENAME request. The
; example renames a file according to filespecs input thru the .CSISPC.
;-
        .MCALL  .RENAME,.PRINT,.EXIT
        .MCALL  .CSISPC,.FETCH,.SRESET
        $ERRBYT =: 52             ;(.SYCDF) Error byte location

START:  .CSISPC #FILESP,#DEFEXT   ;Use .CSISPC to get file specs
        .FETCH  #HANLOD,#FILESP   ;Get Handler from outspec
        BCS     20$               ;Branch if failed
        MOV     #FILESP,R2        ;R2 => Outspec
        MOV     #FILESP+46,R3     ;R3 => Inspec
        MOV     @R2,FILESP+36     ;Copy device spec to inspec
        .REPT   4                 ;Copy outspec behind inspec
        MOV     (R2)+,(R3)+       ;for .RENAME...
        .ENDR
        .RENAME #AREA,#0,#FILESP+36     ;Rename input file
        BCC     10$               ;Operation successful
        DECB    @#$ERRBY          ;Make error code -1,0 or +1
        BEQ     30$               ;Branch if File-Not-Found
        MOV     #ILLOP,R0         ;Illegal operation-set up msg
        BR      40$               ;Branch to report error

10$:    .SRESET                   ;Dismiss handlers
        .EXIT                     ;Exit program

20$:    MOV     #NOHAN,R0         ;Fetch failed-set up message
        BR      40$               ;Branch to report error

30$:    MOV     #NOFIL,R0         ;File not found-setup message
40$:    .PRINT                    ;Print error message
        BR      10$               ;Then exit via .SRESET

AREA:   .BLKW   5                 ;EMT Argument block

DEFEXT: .WORD   0,0,0,0           ;No default extensions

NOFIL:  .ASCIZ  "?ERENAM-F-File not found"
ILLOP:  .ASCIZ  "?ERENAM-F-Illegal Operation"
NOHAN:  .ASCIZ  "?ERENAM-F-.FETCH Failed"
        .EVEN
FILESP: .BLKW   39.               ;CSISPC Input Area
HANLOD  = .                       ;Handlers can load here...
        .END    START
```

# .REOPEN

**EMT 375, Code 6**

The .REOPEN request associates the channel that was specified 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-$376_8$

**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 not available. The .REOPEN has not been done. |

Example:
Refer to the example for the .SAVESTATUS request.

# .RSUM

See .SPND/.RSUM.

# .SAVESTATUS

### EMT 375, Code 5

.SAVESTATUS stores, in a user-specified area of memory, the five-word channel status information RT–11 requires to completely define a file. .SAVESTATUS places data words in memory, frees the specified channel and closes the file. When the saved channel data is required, the .REOPEN request is used. The five words returned by .SAVESTATUS contain the following information:

| Name | Offset | Contents |
|---|---|---|
| C.CSW | 0 | Channel status word |
| C.SBLK | 2 | Starting block number |
| C.LENG | 4 | Length of file |
| C.USED | 6 | Highest block written |
| C.DEVQ | 10 | Number of pending requests |
| C.UNIT | 11 | Device unit number |

.SAVESTATUS can only be used if a file has been opened with .LOOKUP. If .ENTER was used, .SAVESTATUS is invalid and returns an error. Note that .SAVESTATUS is not valid for magtape files because additional status information in the device handler is not available to .SAVESTATUS.

The .SAVESTATUS/.REOPEN requests used together can open many files on a limited number of channels or allow all .LOOKUPs to be done at once to avoid USR swapping. Although this is a useful combination, care must be observed when using it. In particular, the following cases should be avoided:

- When 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 supposedly saved file changes.
- 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.
- .SAVESTATUS and .REOPEN are commonly used 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 reopened one at a time as needed. USR swapping can be minimized by locking in the USR, doing .LOOKUP requests as needed, using .SAVESTATUS to save the file data, and then unlocking the USR. Be careful not to lock in the USR in a multijob environment. If the lower priority job locks in the USR when the higher priority job requires it, the lower priority job is delayed until the higher priority job unlocks the USR.

# .SAVESTATUS

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-$376_8$ |
| **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 open. |
| 1 | The file was opened with an .ENTER request or a .SAVESTATUS request was performed for a magtape file. |

Example:

```
        .TITLE  ESAVES;2
;+
; .SAVESTATUS / .REOPEN - This is an example in the use of the .SAVESTATUS
; /.REOPEN requests. These requests are most commonly used together to
; consolidate access to the USR at one place in the program or if the
; program must access more files than there are I/O channels available.
; Once a channel has been opened, its status may be saved, to be re-opened
; and used later as needed. This example merges 1-6 files into 1 file,
; reading all input files on one channel.
;-
        .MCALL  .CSIGEN,.SAVESTATUS,.REOPEN,.CLOSE,.EXIT
        .MCALL  .READW,.WRITW,.PRINT,.PURGE

        $ERRBY  =: 52           ;(.SYCDF)Error byte in SYSCOM

START:  MOV     SP,R5           ;Save SP, since .CSIGEN changes it
        .CSIGEN #DSPACE,#DEFEXT ;Get file specs,open files,load handlers
        MOV     R5,SP           ;Restore it
        MOV     #3,R4           ;R4 = 1st input channel
        MOV     #AREA,R3        ;R3 => EMT Argument block
        MOV     #SAVBLK,R5      ;R5 => Channel savestatus blocks
10$:    .SAVEST R3,R4,R5        ;Save channel status
        BCS     20$             ;Branch if channel never opened
        ADD     #12,R5          ;Adjust R5 to point to next status block
        INC     R4              ;Bump R4 to = next input channel
        CMP     #8.,R4          ;Done all input channels?
        BGE     10$             ;Branch if not
20$:    MOV     #SAVBLK,R5      ;R5 => to 1st saved channel status
        BEQ     60$             ;Branch if no input files
30$:    .REOPEN R3,#3,R5        ;Re-open input channel on Ch 3
```

```
        CLR     BLK                ;Start reading with block 0
40$:    .READW  R3,#3,#BUFFR,#256.,BLK  ;Read a block
        BCC     50$                ;Branch if no error
        TSTB    @#$ERRBY           ;Check if error = EOF
        BNE     70$                ;Branch if not EOF
        .PURGE  #3                 ;Clear input channel for re-use
        ADD     #12,R5             ;Point R5 to next saved ch status
        TST     @R5                ;Any more input channels?
        BNE     30$                ;Branch if yes
        .CLOSE  #0                 ;We're done...close output channel
        .PRINT  #DONE              ;Announce merge complete
        .EXIT                      ;Exit program

50$:    .WRITW  R3,#0,#BUFFR,#256.,WBLK ;Write block just read
        INC     WBLK               ;Bump to next output block
        INC     BLK                ;same for input blk (doesn't affect C bit)
        BCC     40$                ;Branch if no error on write
        MOV     #WERR,R0           ;Write error - R0 => message
        BR      80$                ;merge...

60$:    MOV     #NOINP,R0          ;R0 => No input files message
        BR      80$                ;merge...

70$:    MOV     #RERR,R0           ;R0 => Read error msg
80$:    .PRINT                     ;Report error
        .EXIT                      ;then exit program

AREA:   .BLKW   5                  ;EMT Argument block

BLK:    .WORD   0                  ;Current read block
WBLK:   .WORD   0                  ;Current write block

SAVBLK::.BLKW   30.                ;Saved channel status area

DEFEXT: .WORD   0,0,0,0            ;No default extensions for CSIGEN

NOINP:  .ASCIZ  "?ESAVES-F-No input files"
WERR:   .ASCIZ  "?ESAVES-F-Write Error"
RERR:   .ASCIZ  "?ESAVES-F-Read Error"
DONE:   .ASCIZ  "!ESAVES-I-I/O Transfer Completed"
        .EVEN
BUFFR:  .BLKW   256.               ;I/O buffer
DSPACE  = .                        ;Handlers start here...

        .END    START
```

# .SCCA

**EMT 375, Code 35, Subcode 0, 1**

The .SCCA programmed request:

- Inhibits a CTRL/C abort
- Indicates when a double CTRL/C is initiated at the keyboard
- Distinguishes between single and double CTRL/C commands

Macro Call:

**.SCCA area,addr[,TYPE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a two-word parameter block |
| **addr** | is the address of a terminal status word (an address of 0 reenables double CTRL/C aborts). In a fully-mapped monitor, if you set the low bit of *addr* on, it will be treated as a Supervisor/Data space; otherwise, it is treated as a User/Data space address. |
| **TYPE=strg** | Optional parameter that specifies mode of SCCA operation as LOCAL (default) or GLOBAL. |

Request format for LOCAL:

R0 → area:

| | |
|---|---|
| 35 | 0 |
| addr | |

Request format for GLOBAL:

R0 → area:

| | |
|---|---|
| 35 | 1 |
| addr | |

When .SCCA is in effect, CTRL/C characters are placed in the input ring buffer and treated as normal characters without specific system functions. The request requires a terminal status word address (*addr*) that is used to report consecutive CTRL/C input sequences. AS.CTC of the status word is set when consecutive CTRL/C characters are detected. The program must clear that bit. An .SCCA request with a status word address of 0 disables the intercept and reenables CTRL/C system action.

Normally, the .SCCA request only affects the job currently running. When the program exits, CTRL/C aborts are automatically reenabled. However, if your monitor includes global SCCA support enabled through system generation, you can choose to disable CTRL/C aborts for as long as you need. Set the TYPE argument to GLOBAL and set *addr* to any valid SCCA control word. (The word pointed to by *addr* is described in TAS.DF.) Thereafter, all CTRL/C aborts will be inhibited until another global .SCCA request is issued to set *addr* to 0. Only background jobs can issue global .SCCA requests, and these do not affect foreground or system job operation. Global .SCCA requests issued by foreground and system jobs act as local .SCCA requests.

#### Notes

There are three cautions to observe when using .SCCA:

- The request can cause CTRL/C to appear in the terminal input stream, and the program must provide a way to handle it.
- The request makes it impossible to terminate program loops from the terminal; therefore, it should be used only in thoroughly tested, reliable programs.

  When .SCCA is in effect and the program enters an infinite loop, the system must be halted and rebootstrapped.
- CTRL/Cs from indirect command files or indirect control files are not intercepted by the .SCCA.

Errors:
None.

Example:

```
        .TITLE  ESCCA;1
;+
; ESCCA - this is an example of .SCCA
;-
        .MCALL  .SCCA,.PRINT

VALCNT = : 100.                         ;wait count

START:  .SCCA   #AREA,#ADDR             ;Disable Control/C
        MOV     #VALCNT,COUNT           ;init counter

 LOOP1:  MOV    AREA,AREA               ;waste time
         .PRINT #MSG1                   ;^C has no effect
         DEC    COUNT
         BNE    LOOP1

        .SCCA   #AREA,#0                ;Enable Control/C
 LOOP2:  .PRINT #MSG2                   ;^C will now work
         BR     LOOP2

AREA:   .BLKW   4
ADDR:   .WORD   0
COUNT: .BLKW 1
MSG1:   .ASCIZ  "!ESCCA-I-Ctrl/C is disabled"
MSG2:   .ASCIZ  "!ESCCA-I-Ctrl/C is enabled"
        .END START
```

# .SDAT/.SDATC/.SDATW

**EMT 375, Code 25**

The .SDAT/.SDATC/.SDATW requests are used with the .RCVD/.RCVDW/.RCVDC calls to allow message transfers between a foreground job and a background job under multijob monitors. .SDAT transfers are similar to .WRITE requests, where 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).

Message handling in the monitor does not check for a word count of zero before queuing a send or receive data request. Since RT–11 distinguishes a send from a receive by complementing the word count, a .SDAT* of zero words is treated as a .RCVD* of zero words. Therefore, avoid a word count of zero at all times when using a .SDAT* request.

You should avoid the use of both synchronous and asynchronous message requests in the same program. Be particularly careful if you use both synchronous (.RCVDW and .SDATW) and asynchronous (.RCVDC and .SDATC) requests in the same program. Issuing a mainline .SDATW while there is a pending .RCVDC, causes .SDATW to wait until the .RCVDC is satisfied. If the completion routine for .RCVDC issues another .RCVDC, the mainline .SDATW will never complete.

**.SDAT**

Macro Call:

**.SDAT area,buf,wcnt[,BMODE=strg]**

where:

**area** is the address of a five- or six-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

**BMODE=strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:

R0 → area:

No BMODE

| | |
|---|---|
| 25 | 0 |
| unused | |
| buf | |
| wcnt | |
| 0 | |

BMODE

| | |
|---|---|
| 25 | 0 |
| unused | |
| buf | |
| wcnt | |
| mapping bits | |
| 0 | |

Errors:

| Code | Explanation |
|---|---|
| 0 | No other job exists. (A job exists as long as it is loaded, whether or not it is active.) |

Example:

```
        .TITLE  ESDATF

;+
; .SDAT/.RCVD - This is an example in the use of the .SDAT/.RCVD
; requests. The example is actually two programs, a Background job
; which sends messages, and a Foreground job, which receives them.
; NOTE: Each program should be assembled and linked separately.
;-

;+
; Foreground Program...
;-

        .MCALL  .RCVD,.MWAIT,.PRINT,.EXIT
```

```
STARTF: .RCVD   #AREA,#MBUFF,#40. ;Request a message up to 80 char.
        ;       ...               ;No error possible - always a BG
                                  ;Do some other processing
        .PRINT  #FGJOB            ;like announcing FG active...
        .MWAIT                    ;Wait for message to arrive...
        TST     MBUFF+2           ;Null message?
        BEQ     FEXIT             ;Yes...exit the program
        .PRINT  #FMSG             ;Announce we got the message...
        .PRINT  #MBUFF+2          ;and echo it back
        BR      STARTF            ;Loop to get another one

FEXIT:  .EXIT                     ;Exit program

AREA:   .BLKW   5                 ;EMT Argument Block

MBUFF:  .BLKW   41.               ;Buffer - Msg length + 1
        .WORD   0                 ;Make sure 80 char message ends ASCIZ

FGJOB:  .ASCIZ  "!SDATF-I-Foreground running"
FMSG:   .ASCIZ  "!SDATF-I-Message from BG:"
        .END    STARTF
```

```
        .TITLE  ESDATB
;+
; Background Program - Send a 'null' message to stop both programs
;-
        .MCALL  .SDAT,.MWAIT,.GTLIN,.EXIT,.PRINT

STARTB: CLR     BUFF              ;Clear 1st word
        .GTLIN  #BUFF,#PROMT      ;Get something to send to FG from TTY
        .SDAT   #AREA,#BUFF,#40.  ;Send input as message to FG
        BCS     1$                ;Branch on error - No FG
        .MWAIT                    ;Wait for message to be sent
        TST     BUFF              ;Sent a null message?
        BNE     STARTB            ;No...loop to send another message.
        .EXIT                     ;Yes...exit program

1$:     .PRINT  #NOFG             ;No FG !
        .EXIT                     ;Exit program

AREA:   .BLKW   5                 ;EMT Argument Block

BUFF:   .BLKW   40.               ;Up to 80 char message
PROMT:  .ASCIZ  "Enter Message for FG:"
NOFG:   .ASCIZ  "?ESDATB-F-No FG"
        .END    STARTB
```

### .SDATC

Macro Call:

**.SDATC area,buf,wcnt,crtn[,BMODE=strg][,CMODE=strg]**

where:

**area** is the address of a five-word or six-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

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

**CMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| U | User space (default) |
| S | Supervisor space |

Specifying *CMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *CMODE*.
- Specifies the *space* for the *crtn* argument.
- Is a valid option only for fully mapped monitors.

Request Format:

R0 → area:

No BMODE, CMODE

| | |
|---|---|
| 25 | 0 |
| unused | |
| buf | |
| wcnt | |
| crtn | |

BMODE, CMODE

| | |
|---|---|
| 25 | 0 |
| unused | |
| buf | |
| wcnt | |
| mapping bits | |
| crtn | |

Errors:

| Code | Explanation |
|---|---|
| 0 | No other job exists. (A job exists as long as it is loaded, whether or not it is active.) |

Example:
See DEMOFG.MAC and DEMOBG.MAC programs on the installation kit.

#### .SDATW

Macro Call:

**.SDATW area,buf,wcnt[,BMODE=strg]**

where:

**area** is the address of a five- or six-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

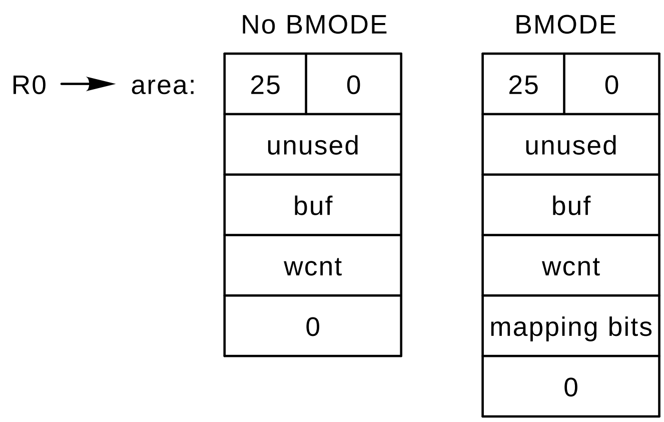
**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:




Errors:

| Code | Explanation |
|---|---|
| **0** | No other job exists. (A job exists as long as it is loaded, whether or not it is active.) |

Example:

```
        .TITLE  ESDAWF
;+
; .SDATW/RCVDW - This is an example in the use of the .SDATW/.RCVDW
; requests. The example consists of two programs; a Foreground job
; which creates a file and sends a message to a Background program
; which copies the FG channel and reads a record from the file. Both
; programs must be assembled and linked separately.
;-
;+
; This is the Foreground program...
;-
        .MCALL  .ENTER,.PRINT,.SDATW,.EXIT,.RCVDW,.CLOSE,.WRITW

STARTF: MOV     #AREA,R5                ;R5 => EMT argument block
        .ENTER  R5,#0,#FILE,#5          ;Create a 5 block file
        .WRITW  R5,#0,#RECRD,#256.,#4 ;Write a record BG is interested in
        BCS     ENTERR                  ;Branch on error
        .SDATW  R5,#BUFR,#2             ;Send message with info to BG
        ;       .                       ;Do some other processing
        .RCVDW  R5,#BUFR,#1             ;When it’s time to exit, make sure
        .CLOSE  #0                      ;BG is done with the file
        .PRINT  #FEXIT                  ;Tell user we’re exiting
        .EXIT                           ;Exit the program

ENTERR: .PRINT  #ERMSG                  ;Print error message
        .EXIT                           ;then exit

FILE:   .RAD50  /DK ESDAWF/             ;File spec for .ENTER
        .RAD50  /TMP/

AREA:   .BLKW   5                       ;EMT argument block

BUFR:   .WORD   0                       ;Channel #
        .WORD   4                       ;Block #
RECRD:  .BLKW   256.                    ;File record

ERMSG:  .ASCIZ  "?ESDAWF-F-Enter Error"
FEXIT:  .ASCIZ  "!ESDAWF-I-FG Job exiting"
        .END    STARTF

        .TITLE  ESDAWB
;+
; This is the Background program ...
;-
        .MCALL  .CHCOPY,.RCVDW,.READW,.EXIT,.PRINT,.SDATW

STARTB: MOV     #AREA,R5                ;R5 => EMT arg block
        .RCVDW  R5,#MSG,#2              ;Wait for message from FG
        BCS     10$                     ;Branch if no FG
        .CHCOPY R5,#0,MSG+2             ;Channel # (1st word of message)
        BCS     20$                     ;Branch if FG channel not open
        .READW  R5,#0,#BUFF,#256.,MSG+4 ;Read block (2nd word of msg)
        BCS     30$                     ;Branch if read error
        ;       .                       ;Continue processing...
        .SDATW  R5,#MSG,#1              ;Tell FG we’re thru with file
        .PRINT  #BEXIT                  ;Tell user we’re thru
        .EXIT                           ;then exit program

10$:    MOV     #NOJOB,R0               ;R0 => No FG error msg
        BR      40$                     ;Branch to print msg
```

```
20$:    MOV     #NOCH,R0                ;R0 => FG ch not open msg
        BR      40$                     ;Branch...

30$:    MOV     #RDERR,R0               ;R0 => Read err msg
40$:    .PRINT                          ;Print proper error msg
        .EXIT                           ;then exit.

AREA:   .BLKW   5                       ;EMT argument blk
MSG:    .BLKW   3                       ;Message buffer
BUFF:   .BLKW   256.                    ;File buffer

BEXIT:  .ASCIZ  "!ESDAWB-I-Channel-Record copy successful"
NOJOB:  .ASCIZ  "?ESDAWB-F-No FG Job"
NOCH:   .ASCIZ  "?ESDAWB-F-FG channel not open"
RDERR:  .ASCIZ  "?ESAWB-F-Read Error"
        .END    STARTB
```

# .SDTTM

**EMT 375, Code 40**

The .SDTTM (Set date and time) request allows your program to set the system date and time.

Macro Call:

**.SDTTM area,addr**

where:

| | |
|---|---|
| **area** | is the address of a two-word EMT argument block |
| **addr** | is the address of a three-word block in user memory that contains the new date and time |

Request Format:

R0 → area:

| | |
|---|---|
| 40 | 0 |
| addr | |

**Notes**

The first word of the three-word parameter block contains the new system date in internal format (See the .DATE programmed request). If this word is -1 (represents an invalid date), the monitor ignores it. Put a -1 in the first word of the parameter block if you want to change only the system time. If the first parameter word is positive (not –1), it becomes the new system date. Note that the monitor does no further checking on the date word. To be sure of a valid system date, you must specify a value between 1 and $12_{10}$ in the month field (bits 13-10) and a value between 1 and the month length in the day field (bits 9-5). Bits 14 and 15 are the age bits. See .DATE request for description of format.

The second and third words of the parameter block are the new high-order and low-order time values, respectively. This value is the double-precision number of ticks since midnight. If the high-order time word is negative, the monitor ignores the new time. Put a negative value in the second word of the parameter block if you want to change only the system date. If the second parameter word is positive, the new time becomes the system time. The monitor does no further checking on the new time. To be sure of a valid system time, you must specify a valid number of ticks for the system line frequency. For a 60 Hz clock, the high-order time may not be larger than $117_8$, and if it is equal to 117, the low-order time may not be equal to or larger than $15000_8$. For a 50 Hz clock, the high-order time may not be larger than $101_8$, and if it is equal to 101, the low-order time may not be equal to or larger than $165400_8$.

Changing the date and/or time has no effect on any outstanding mark time or timed wait requests.

Errors:
None.

Example:

```
        .TITLE  ESDTTM;2

;+
; .SDTTM - This is an example in the use of the .SDTTM request.
; The example is a Daylight/Standard Time utility - to switch the
; current system time from Standard to Daylight or vice versa, call
; the program as a subroutine at the proper entry point.
;-

        .MCALL  .SDTTM, .GTIM

.ENABL  LSB

STD::   MOV     #MINSHR+4,R3            ;Subtract an hour
        BR      10$

DALITE::MOV     #PLUSHR+4,R3            ;Add an hour
10$:    .GTIM   #AREA,#TIME             ;Get the current time
        CALL    JADD                    ;Adjust +/- 1 hour
        .SDTTM  #AREA,#NEWDT            ;Set the new system time
        RETURN                          ;Return to caller

JADD:                                   ;Double precision integer add
        MOV     #TIME+4,R4              ;R4 => Low order of System time + 2
        MOV     -(R4),R2                ;Put low order of 1st operand in R2
        ADD     -(R3),R2                ;Add in low order of operand #2
        MOV     -(R4),R5                ;*C*Put high order of operand #1 in R5
        ADC     R5                      ;Add in carry (no overflow possible!)
        ADD     -(R3),R5                ;Add in high order of operand #2
                                        ;(ditto!)
        MOV     R5,(R4)+                ;Store result where wanted
        MOV     R2,(R4)+
        RETURN                          ;Return to caller

NEWDT:  .WORD   -1                      ;.SDTTM arguments - No new date
TIME:   .WORD   0,0                     ;New time

PLUSHR: .WORD   3                       ;One hour in clock ticks
        .WORD   45700                   ; (60 Hz clock!)

MINSHR: .WORD   ^c3                     ;Minus One hour in clock ticks
        .WORD   -45700

AREA:   .WORD   0,0                     ;EMT Argument Block
        .END
```

# .SERR

See .HERR/.SERR

## .SETTOP

**EMT 354**

The .SETTOP request specifies a new address as a program's upper limit. Using .SETTOP offers significant performance improvement in running your program.

Macro Call:

**.SETTOP addr**

where:

**addr** is the address of the highest word of the area desired; that is, the last word the program will modify, not the first word it leaves untouched

### Notes

- A program should never do a .SETTOP and assume its new upper limit is the address it requested. It must always examine the returned contents of R0 or location $50_8$ to determine its actual high address.
- The value returned in R0 or location $50_8$ must be used as the absolute upper limit. If this value is exceeded, vital parts of the monitor can be destroyed.

When .SETTOP specifies a new address as a program's upper limit, the monitor determines whether the address is valid and whether or not a memory swap is necessary when the USR is required. When a program specifies an upper limit below the start address of USR (normally specified in $USRLC in the resident monitor), no swapping is necessary, because the program does not overlay the USR. 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. The use of .SETTOP in an extended memory environment is described at the end of this section.

Careful use of the .SETTOP request provides a significant improvement in the performance of your program. An approach that is used by several of the system-supplied programs is as follows:

- A .SETTOP is done to the high limit of the code in a program before buffers or work areas are allocated. If the program aborts, minimal writing of the user program to the swap blocks (SWAP.SYS) occurs. However, the program is allowed to be restarted successfully.
- A user command line is read through .CSISPC or .GTLIN. An appropriate USR swap address is set in $UFLOT. 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 the procedure.
- Buffers and work areas are allocated as needed with appropriate .SETTOP requests being issued 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, issuing another .SETTOP will cause the USR to become resident again to close files (Remember to set $UFLOT to zero so that the USR again swaps in the normal area). On return from .SETTOP, both R0 and the word in $USRTO contain the highest memory address allocated for use.

  When a requested address is higher than the highest address legal for the requesting job, the address returned will be the highest legal address for the job, not 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 to the swap blocks (SWAP.SYS) 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. The monitor command SET EXIT NOSWAP also disables program swapping.

Errors:
None.

Example:

See .LOCK.

#### .SETTOP in an Extended Memory Environment

You can enable the extended memory feature of the .SETTOP programmed request with the linker /V option or the LINK command with the /XM option (See *RT–11 System Utilities Manual* or *RT–11 Commands Manual*). The *RT–11 System Internals Manual* describes in detail the .SETTOP request in an extended memory environment. The .SETTOP request operates in privileged and virtual jobs as follows:

*Privileged Jobs*

- A .SETTOP that requests an upper limit below the virtual high limit of the program will always return the virtual high limit of the program. The virtual high limit is the last address in the highest PAR that the program uses. In this case, a value can never be returned below the job's virtual high limit.
- A .SETTOP that requests a job's upper limit above the program's virtual high limit will return the highest available address as follows:
  - Either the address requested or SYSLOW-2 (last used address, SYSLOW is next address available) is returned, whichever is lower. SYSLOW is defined as the start of the USR in the XM monitor.
  - If the program's virtual high limit is greater than SYSLOW (the user program maps over the monitor or USR), the virtual high limit of the program will always be returned.

*Virtual Jobs*

- As in privileged jobs, a .SETTOP request can never get less than the virtual high limit of the job.
- If a .SETTOP requests an upper limit greater than the virtual high limit, the following occurs:
  - If the virtual high limit equals 177776, this value is returned since this is the address limit in virtual memory. Otherwise, a new region and window will be created. The size of the region and window will be determined by the argument specified to the .SETTOP or by the amount of extended memory that is available, whichever value is smaller. The .SETTOP argument rounded to a 32-word boundary minus the high .LIMIT value for the program equals the size of the region and window (See the *RT–11 System Utilities Manual* and the *RT–11 System Internals Manual* for a description of the .LIMIT directive in extended memory). If there are no region control blocks, window control blocks, or extended memory available, the program's virtual high limit is returned. The .SETTOP request uses one of the region and window control blocks allocated to the user, thus one less block is available to the program if the linker /V option is used.
  - Additional .SETTOP requests can only remap the original window created by the first .SETTOP. Thus, additional requests will return an address no higher than that established by the first request and no lower than the program virtual high limit. An additional .SETTOP request whose argument is higher than the first request will cause the entire first window to be mapped. An additional .SETTOP request whose argument specifies a value below the virtual high limit eliminates the region and window. If another .SETTOP request then follows, it may create a new region and window.

# .SFDAT

**EMT 375, Code 42**

The .SFDAT programmed request allows a program to set or modify the creation date in a file's directory entry. Dates on protected as well as unprotected files can be changed. .SFDAT is not supported for distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.SFDAT area, chan, dblk[,date]**

where:

**area** is the address of a three-word EMT argument block

**chan** is a channel number in the range $0\text{-}376_8$

**dblk** is the address of a four-word block containing a filespec in Radix–50

**date** is the value of the new date, in RT–11 format. If this argument is #0 or omitted, the system date is used. No other check is made for an invalid date

Request Format:

| | | |
|---|---|---|
| R0 → area: | 42 | chan |
| | dblk | |
| | date | |

Errors:

| Code | Explanation |
|---|---|
| 0 | Channel not available |
| 1 | File not found |
| 2 | Invalid operation (device not file structured) |

Example:
Refer to the example for the .FPROT request.

# .SFINF

**EMT 375, Code 44**

The .SFINF programmed request saves in R0 and then modifies the contents of the directory entry offset you specify from a file's directory entry. .SFINF is not supported for the distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.SFINF area,chan,dblk,value,type,offse,ucode**

where:

**area** is the address of a 4-word or 5-word (see *dblk*) EMT argument block

**chan** is a channel number in the range of 0 to $376_8$

**dblk** is a four-word Radix–50 descriptor block that specifies the physical device, file name, and file type to be operated upon.

**value** is the value used to modify the specified offset location.

For RT–11 file structured volume directories,

- If the offset is 0 (E.STAT) and the operation is a BIC or BIS, E.STAT bits 000400, 001000, and 004000, must be clear.
- If the offset is E.STAT and the operation is a MOV, only the bottom 4 bits of E.STAT are moved.

For special directory volumes, no bit restrictions are enforced. The operation is dependent on the handler.

**type** is the name indicating the operation to be performed:

| Name | Value | Meaning |
|---|---|---|
| GET | 0 | Get value; a .GFDAT, .GFINF, or .GFSTA operation |
| BIC | 1 | A bit clear (BIC) operation |
| BIS | 2 | A bit set (BIS) operation |
| MOV | 3 | A word move (MOV) operation |
| | 4-177 | Reserved for Digital |
| USER | ucode | Reserved for the user |

**offse** is the octal byte offset for the directory entry word for this operation. The offset must be even and cannot be 10 (E.LENG). For example, specifying offset 12 saves the current contents of E.USED in R0 and modifies that location according to the *value* and *type* arguments

**ucode** is the user operation code containing the value for the *type* parameter. Specify *ucode* only when the *type* parameter argument is USER. Specify values in the range $200–377_8$.

Request Format:

R0 → area:

| | |
|---|---|
| 44 | chan |
| dblk | |
| value | |
| offse | type |

Errors:

Special directory handlers define their own codes. For RT–11 file structure devices:

| Code | Explanation |
|---|---|
| 0 | Channel is not available |
| 1 | File not found, or not a file-structured device. If it is necessary to determine what condition returned the error code, issue a .DSTAT request to determine if a device is file structured. |
| 2 | Operation is invalid |
| 3 | Offset is invalid |
| 4 | Value is invalid |

Example:

```
.TITLE  ESFINF
;
; This program modifies the selected directory entry.
; It places in offset 12(8) of the entry, the current
; number of seconds since midnight divided by 3.
;
.MCALL  .GTIM   .CSISPC .FETCH  .RELEAS .PRINT  .EXIT
.MCALL  .SFINF

.GLOBL  $DIVTK                  ;divide by number of ticks in a second
.GLOBL  $DIVNN                  ;divide by value in R4
```

```
START::
        .CSISPC #OUTSPC,#DEFEXT ;Get a file name
        BCS     CSIERR          ;Error
        .FETCH  LIMIT+2,#INSPC  ;Fetch the handler
        BCS     FETERR          ;Error
        .GTIM   #AREA,#TIME     ;Get the current time
        MOV     TIME+0,R0       ;Into registers for divide
        MOV     TIME+2,R1       ;...
        CALL    $DIVTK          ;Get number of seconds past midnight
        MOV     #3,R4           ;Now divide that by 3
        CALL    $DIVNN          ;... result in R1
        .SFINF  #AREA,#0,#INSPC,R1,MOV,#12 ;Modify
                                ;directory entry
        BCS     SFIERR          ;Error
        .RELEASE #INSPC         ;Dismiss handler
        BR      START           ;And again

CSIERR: MOV     #CSIMSG,R0
        BR      DONE

FETERR: MOV     #FETMSG,R0
        BR      DONE

SFIERR: MOV     #SFIMSG,R0
DONE:   .PRINT  R0
        .EXIT

OUTSPC: .BLKW   3*5             ;CSISPC return area
INSPC:  .BLKW   6*4             ;   "      "    "
DEFEXT: .RAD50  "             " ;Default extensions (none)
LIMIT:  .LIMIT                  ;Memory usage (macro directive)
AREA:   .BLKW   10.             ;EMT request block area
TIME:   .BLKW   2               ;Ticks since midnight

CSIMSG: .ASCIZ  "?ESFINF-E-CSI error"
FETMSG: .ASCIZ  "?ESFINF-E-Fetch error"
SFIMSG: .ASCIZ  "?ESFINF-E-SFINF error"

        .END    START
```

# .SFPA

**EMT 375, Code 30**

By selecting .SFPA during SYSGEN, users with floating-point hardware can set trap addresses to be entered when a floating-point exception occurs. Issue this request with #1 as the *addr* argument to provide swapping of floating-point context without setting a trap address. 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. In a fully-mapped monitor, if you set the low bit of *addr* on, it will be treated as a Supervisor/Data space; otherwise, it is treated as a User/Data space address. |

Request Format:

| | | |
|---|---|---|
| R0 → area: | 30 | 0 |
| | addr | |

### Notes

- The user trap routine must save and restore any registers it uses. It exits with an RTI instruction.
- If the address argument is #0, user floating-point routines are disabled and the fatal *?MON-F-FPU trap error* is produced by any further traps.
- In the multijob 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 single-job monitor is equivalent to an address of #0.
- When the user routine is activated, it is necessary to re-execute an .SFPA request, as the monitor disables user traps as soon as one is serviced. It does this to prevent a possible infinite loop from being set up by repeated floating-point exceptions.
- 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  ESFPA;2
;+
; .SFPA - This is an example in the use of the .SFPA request. This
; example is a skeleton program which demonstrates how to set up a
; Floating Point trap routine, and the minimum action that routine
; must take before dismissing the error trap.
;-
        .MCALL  .SFPA,.EXIT,.PRINT

        $SYPTR  =: 54                   ;(.SYCDF)Loc of beginning of Monitor
        $CNFG1  =: 300                  ;(.FIXDF)Offset to first config
        HWFPU$  =: 100                  ;(.CF1DF)FPU present bit
START:  ;       .                       ;Mainline program...
        ;       .
        .SFPA   #AREA,#FPTRAP           ;Set up FPU error trap
        ;       .                       ;continue mainline program
        DIVF    F0,%1                   ;cause a divide by 0 interrupt
        .EXIT                           ;Exit program

FPTRAP:                                 ;FPU exception routine
        MOV     R0,-(SP)
        ;       .                       ;Handle exception...
        .PRINT  #FPTMSG                 ;Indicate it happened
        ;       .
CKFPU:  MOV     @#$SYPTR,R0             ;R0 => base of RMON
        BIT     #HWFPU$,$CNFG1(R0)      ;Check for FPU hdwe
        BEQ     10$                     ;Branch if none
        MOV     (SP)+,R0
        CMP     (SP)+,(SP)+             ;Must pop status regs off stack!
        RTI

10$:    MOV     (SP)+,R0
        RTI                             ;Before returning from interrupt

F0:     .WORD   0,0
AREA:   .BLKW   10.

FPTMSG: .ASCIZ  "?ESFPA-W-FPU trap occurred"

        .END    START
```

# .SFSTAT

**EMT 375, Code 44**

The .SFSTAT programmed request saves in R0 and then modifies the contents of the directory entry E.STAT offset from a file's directory entry. .SFSTA is not supported for the distributed special directory handlers LP, LS, MM, MS, MT, MU, and SP.

Macro Call:

**.SFSTAT area,chan,dblk,value,type,ucode**

where:

**area** is the address of a 4-word EMT argument block

**chan** is a channel number in the range of 0 to 376(octal)

**dblk** Four-word Radix–50 descriptor block that specifies the physical device, file name, and file type to be operated upon.

**value** is the value to be placed in the E.STAT offset location.

For RT–11 file-structured volume directories,

- If the operation is a BIC or BIS, E.STAT bits 000400, 001000, and 004000 must be clear.
- If the operation is a MOV, only the bottom 4 bits of E.STAT are moved.

For special directory volumes, no bit restrictions are enforced.

**type** is the name indicating the operation to be performed:

| Name | Value | Meaning |
|---|---|---|
| GET | 0 | Get value; a .GFDAT, .GFINF, or .GFSTA operation |
| BIC | 1 | A bit clear (BIC) operation |
| BIS | 2 | A bit set (BIS) operation |
| MOV | 3 | A word move (MOV) operation |
| | $4\text{-}177_8$ | Reserved for Digital |
| USER | ucode | Reserved for the user ($200–322_8$) |

**ucode** is the user operation code containing the value from the *type* parameter. Specify *ucode* only when the *type* parameter argument is USER.

Request Format:

R0 → area:

| | |
|---|---|
| 44 | chan |
| dblk | |
| value | |
| 0 | type |

Errors:

Special directory handlers define their own codes. For RT–11 file structure devices:

| Code | Explanation |
|---|---|
| 0 | Channel is not available |
| 1 | File not found, or not a file-structured device. (If it is necessary to determine what condition returned the error code, issue a .DSTAT request to determine if a device is file structured) |
| 2 | Operation is invalid |
| 3 | Invalid EMT argument block |
| 4 | Invalid EMT argument block |

Example:

```
.Title  ESFSTA  -- sample program for .SFSTA
;
; This program accepts 1 or more (possibly wild) file
; specifications, all on a single device, and a switch
; (/W or /R).  It sets (/R) or clears (/W) the "readonly"
; bit in the directory entries for each matching file.
;
; The command line must be of the form:
;
;            dev:file.typ[,...]{/R}
;                              {/W}
;

.MCALL  .GTLIN  .CSISPC .FETCH  .SFSTA  .PRINT  .EXIT

.GLOBL  IGTDIR  IGTENT

.ENABL  LSB
```

```
START::
        .GTLIN  #CBUFFER,#PROMPT ;Get a command line
        MOV     #CBUFFER,R0     ;setup to prune the command line
        CLR     R1              ;pointer for char past ":"
10$:
        CMPB    @R0,#':         ;end of device spec?
        BNE     20$             ;no
        TST     R1              ;first ":"?
        BNE     CMDERR          ;no, invalid command line
        MOV     R0,R1           ;remember location
        INC     R1              ;of NEXT char
20$:
        CMPB    @R0,#'/         ;Switch introducer?
        BNE     30$             ;no
        MOV     R0,R2           ;yes, save address
        BR      40$             ;and quit scan
30$:
        TSTB    (R0)+           ;point to next char,
                                ; and look for EOS
        BNE     10$             ;more to look at
40$:
        TST     R1              ;was "dev:" found?
        BEQ     DEVERR          ;no, required
        .CSISPC #OUTSPC,#DEFEXT,#CBUFFER
                                ;Parse any device spec
                                ;  and switches
        BCS     CSIERR          ;error
        MOV     #2,R4           ;Assume /R, use BIS
        MOV     (SP)+,R0        ;get switch count
        BEQ     100$            ;no switches, nothin' to do
50$:
        TST     R0
        BEQ     80$             ;checked all switches
        BICB    #040,@SP        ;force uppercase
        CMPB    @SP,#'R         ;/R?
        BEQ     60$             ;yes
        CMPB    @SP,#'W         ;/W?
        BNE     70$             ;no
        MOV     #1,R4           ;yes, use BIC
60$:
        DEC     R0              ;reduce switch count
        TST     (SP)+           ;pop stack, check for value
        BPL     50$             ;no value specified
70$:    BR      SWIERR          ;unknown switch, or value specified

80$:    MOV     R4,TYPE         ;save operation type
```

```
        .FETCH  LIMIT+2,#DBLK    ;Fetch the handler
        BCS     FETERR           ;error
        CLR     DBLK+2           ;make DBlk Non-file structured
        MOV     R1,STRING        ;where list of file specs starts
        CLRB    @R2              ;Terminate at first "/"
        MOV     #PGTDIR,R5       ;point to arg list
        CALL    IGTDIR           ;setup for wildcard search
        TST     R0               ;any errors?
        BNE     GTDERR           ;yes
90$:
        MOV     #PGTENT,R5       ;point to arg list
        CALL    IGTENT           ;try for an entry
        TST     R0               ;any errors?
        BMI     100$             ;error, or no (more) matches
        .PRINT  #FILNAM          ;display selected names
        MOV     DBLK,ENTRY       ;fill in device name
                                 ;Set or clear readonly bit:
        .SFSTA  #AREA,#1,#ENTRY,#040000,USER,TYPE
        BCC     90$              ;no error
SFSERR: .PRINT  #SFSMSG          ;SFSTA error
        BR      100$
DEVERR: .PRINT  #DEVMSG          ;DEV: not specified
        BR      100$
CSIERR: .PRINT  #CSIMSG          ;CSI error
        BR      100$
FETERR: .PRINT  #FETMSG          ;Fetch error
        BR      100$
CMDERR: .PRINT  #CMDMSG          ;Command semantic error
        BR      100$
SWIERR: .PRINT  #SWIMSG          ;Invalid switch or
                                 ; a value specified
        BR      100$
GTDERR: .PRINT  #GTDMSG          ;IGTDIR error
100$:   .EXIT

.DSABL  LSB
DEFEXT: .RAD50  "            "   ;default extensions (none)
LIMIT:  .LIMIT                   ;Program limits (macro directive)
LIT64.: .WORD   64.              ;literal 64.
LIT0:   .WORD   0                ;literal 0.

PGTENT: .WORD   5                ;IGTENT argument list
        .WORD   WORK
        .WORD   ENTRY
        .WORD   -1
        .WORD   -1
        .WORD   FILNAM

PGTDIR: .WORD   7                ;IGTDIR argument list
        .WORD   LIT64.
        .WORD   WORK
        .WORD   LIT0
        .WORD   BUFFER
        .WORD   -1
        .WORD   DBLK
STRING: .BLKW   1
```

```
OUTSPC: .BLKW   3*5             ;output file specs (unused)
DBLK:   .BLKW   4               ;first input file spec
        .BLKW   5*4             ;rest of file specs (unused)

AREA:   .BLKW   4               ;EMT area
WORK:   .BLKW   64.             ;IGT(DIR,ENT) work area
BUFFER: .BLKW   512.            ;Directory segment buffer
ENTRY:  .BLKW   7.              ;returned directory entry
TYPE:   .BLKW   1.              ;BIC/BIS operation code

CBUFFER:.BLKB        81.        ;Command line buffer

FILNAM: .ASCIZ  "xxxxxx.xxx"
PROMPT: .ASCII  "ESFSTA>"
        .BYTE   200             ;No CR LF
DEVMSG: .ASCIZ  "?ESFSTA-E-No device specified"
CSIMSG: .ASCIZ  "?ESFSTA-E-CSISPC error"
FETMSG: .ASCIZ  "?ESFSTA-E-Fetch error"
SFSMSG: .ASCIZ  "?ESFSTA-E-SFSTAT error"
CMDMSG: .ASCIZ  "?ESFSTA-E-More than 1 device specified"
SWIMSG: .ASCIZ  "?ESFSTA-E-Invalid switch or value"
GTDMSG: .ASCIZ  "?ESFSTA-E-IGtDir error"

        .END    START
```

# SOB

### Macro Expansion

The SOB macro simulates the SOB instruction (subtract one and branch if not equal) by generating the code:

```
DEC     REG
BNE     ADDR
```

You can use the SOB macro on all processors, but it is especially useful for processors that do not have the hardware SOB instruction. If you are running on a processor that supports the SOB instruction, simply eliminate the MACRO call to SOB (.MCALL SOB), and the SOB instruction executes. Note that SOB is not preceded by a dot (.).

Macro Call:

**SOB reg,addr**

where:

**reg** is the register whose contents will be decremented by 1

**addr** is the location to branch to if the register contents do not equal 0 after the decrement

In the following example, R0 is decremented by 1 and then tested. If the contents do not equal 0, the program branches to the label HERE.

```
        .TITLE  ESOB2

        .MCALL  SOB

        MOV     #count,R0
        MOV     #source,R1
        MOV     #dest,R2
LOOP:
        MOV     (R1)+,(R2)+
        SOB     R0,LOOP
```

**NOTE**

The SOB instruction does not change any condition codes. The SOB macro can change the N, Z, and V (but not the C) condition codes.

# .SPCPS

**EMT 375, Code 41**

Support for this request must be selected during SYSGEN. The .SPCPS (save/set mainline PC and PS) request allows a program's completion routine to change the flow of control of the mainline code.

.SPCPS saves the mainline code PC and PS, and changes the mainline PC to a new value. If the mainline code is performing a monitor request, the monitor allows that request to finish before doing any rerouting. The actual rerouting is deferred until the mainline code is about to run. Therefore, the .SPCPS request returns an error if it is reissued before an earlier request has been honored. Furthermore, the data saved in the user block is not valid until the new mainline code is running.

Macro Call:

**.SPCPS area,addr**

where:

**area** is the address of a two-word EMT argument block

**addr** is the address of a three-word block in user memory that contains the new mainline PC, and that is to contain the old mainline PC and PS

Request Format:

| | | |
|---|---|---|
| R0 → area: | 41 | 0 |
| | addr | |

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | The program issued the .SPCPS call from the mainline code rather than a completion routine. |
| 1 | A previous .SPCPS request is outstanding. |

When the program issues the .SPCPS request, the monitor saves the old mainline PS in the third word of the three-word block and the old mainline PC in the second word of the block. The monitor then changes the mainline PC to the contents of the first word of the block.

Example:

```
        .TITLE  ESPCPS
        .ENABL  LC
;+
; .SPCPS - This is an example in the use of the .SPCPS request. In this
; example .SPCPS is used to reroute the mainline code after an I/O
; error or EOF is detected by a completion routine.
;-
        .MCALL  .READC,.WRITC,.CLOSE,.PRINT,.CSIGEN,.EXIT,.WAIT,.SRESET
        .MCALL  .SPCPS  .QSET

        $ERRBY  =: 52               ;(.SYCDF)Error Byte in SYSCOM

        .ENABL  LSB
START:  .CSIGEN #DSPACE,#DEFEXT     ;Use CSIGEN to get handlers, files
        .QSET   #QEL,#QELNUM        ;Allocate queue elements
        CALL    IOXFER              ;Start I/O
        .PRINT  #MESSG              ;Now simulate other mainline process
1$:     DEC     R5                  ; (Kill some time)
        BR      1$

FINI:   .CLOSE  #0                  ;EOF > 0 = End of File
        MOV     #DONE,R0            ;R0 -> We're done message
        BR      GBYE                ;Merge to exit program

WERR:   MOV     #WRERR,R0           ;Set up error messages here...
        BR      GBYE

RERR:   MOV     #RDERR,R0
GBYE:   .PRINT                      ;Print message
        .SRESET                     ;Dismiss fetched handlers
        .EXIT                       ;Exit program

WRDONE: .WAIT   #0                  ;Write compl rtne...write successful?
        BCS     3$                  ;Branch if not...
IOXFER: .READC  #AREA,#3,,,#6$      ;Queue up a read
        BCC     5$                  ;Branch if ok...
        TSTB    @#$ERRBY            ;Error - is it EOF?
        BEQ     4$                  ;Branch if yes
2$:     MOV     #RERR,SBLOK         ;Move Read err rtne addr to arg block
        BR      4$                  ;Merge...

3$:     MOV     #WERR,SBLOK         ;Move Write err rtne addr to arg block
4$:     TSTB    SPCALL              ;Already done a .SPCPS?
        BNE     5$                  ;Yes...don't do another
        .SPCPS  #AREA,#SBLOK        ;De-rail mainline code
        INCB    SPCALL              ;Flag we've done this
        BCS     7$                  ;Ooops! Something's amiss!
5$:     RETURN                      ;Leave completion code

6$:     .WAIT   #3                  ;Completion routine #2 - was read ok?
        BCS     2$                  ;Branch if not
        .WRITC  #AREA,#0,,,#WRDONE  ;Queue up a write...
        BCS     3$                  ;Branch if error
        INC     BLOK                ;Bump block # for next read
        RETURN                      ;Leave Completion code...

7$:     .PRINT  #SPERR              ;Print .SPCPS failed message
        RETURN

AREA::  .WORD   0                   ;EMT Area block
BLOK:   .WORD   0                   ;Block #.
        .WORD   BUFF                ;Buffer addr & word count
        .WORD   256.                ;already fixed in block...
        .WORD   0                   ;Completion routine addr
```

```
SBLOK:  .WORD   FINI,0,0            ;.SPCPS Argument block (FINI default)
QELNUM  =:      3.
QEL:    .BLKW   10.*QELNUM          ;Queue elements

BUFF:   .BLKW   256.                ;I/O buffer

DEFEXT: .WORD   0,0,0,0             ;No default extensions for CSIGEN

SPCALL: .BYTE   0                   ;.SPCPS called flag in case I/O error
                                    ;(compl rtne gets sched. regardless!)
        .NLIST  BEX
DONE:   .ASCIZ  "!ESPCPS-I-I/O Transfer Complete"
MESSG:  .ASCIZ  "!ESPCPS-I-Simulating Mainline Processing"
WRERR:  .ASCIZ  "?ESPCPS-F-Write Error"
RDERR:  .ASCIZ  "?ESPCPS-F-Read Error"
SPERR:  .ASCIZ  "?ESPCPS-F-.SPCPS Error"
        .EVEN
DSPACE  = .                              ;Handlers may be loaded starting here
        .END    START
```

# .SPFUN

**EMT 375, Code 32**

This request is used with certain device handlers to do device specific functions, such as rewind and backspace. It can be used with some disks to allow reading and writing of absolute sectors. It is commonly used to determine the size of devices of variable size.

For device-specific information, refer to the *RT–11 Device Handlers Manual*.

Macro Call:

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

where:

| | |
|---|---|
| **area** | is the address of a six-word EMT argument block |
| **chan** | is a channel number in the range 0 to $376_8$ |
| **func** | is the numerical code of the function to be performed; these codes must be negative |
| **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) |
| >500 | Completion routine |

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

**CMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| U | User space (default) |
| S | Supervisor space |

Specifying *CMODE*:

- Loads a word in the EMT request block area containing a bit pattern matching the code specified for *CMODE*.
- Specifies the *space* for the *crtn* argument.
- Is a valid option only for fully mapped monitors.

Request Format:

R0 → area:

<table>
<tr><td>32</td><td>chan</td></tr>
<tr><td colspan="2">blk</td></tr>
<tr><td colspan="2">buf</td></tr>
<tr><td colspan="2">wcnt</td></tr>
<tr><td>func</td><td>flags</td></tr>
<tr><td colspan="2">crtn</td></tr>
</table>

The *chan, blk,* and *wcnt* arguments are the same as those defined for .READ/.WRITE requests. If the *crtn* argument is left blank, the requested operation completes before control returns to the user program. Specifying *crtn* as #1 is similar to executing a .READ or .WRITE in that the function is initiated and returns immediately to the user program. Use a .WAIT on the channel to make sure that the operation is completed. The *crtn* argument is a completion routine address to be entered when the operation is complete. See the *RT–11 Device Handlers Manual* for device-specific information on *buf*, *wcnt*, and *blk*.

When using .SPFUN, be sure values in arguments are correct; otherwise, enter zero (0) values into arguments to ensure the results of the operation.

Errors:

| **Code** | **Explanation** |
|---|---|
| 0 | Attempt to read or write past end-of-file, or invalid function value. |
| 1 | Hard error occurred on channel. |
| 2 | Channel is not open. |

Example:

```
        .TITLE  ESPFUN

;+
; This program demonstrates the use of .SPFUN to get the size
; of a disk unit.  To use it, enter a device name at the prompt.
;-

        .MCALL  .CSIGEN   .SPFUN     .DEBUG       .DPRINT        .EXIT
        .MCALL  .BR

.ENABL  LSB
        .DEBUG  SWITCH=ON,VALUE=YES

SF.SIZ  =:        373               ;(.SFDDF) Size request
```

```
START::
        .CSIGEN #DEVSPC,#DEFEXT    ;Get the device to size
        BCS     CSIERR                      ;failure
        CLR     REPLY                         ;Assume nothing
        .SPFUN  #AREA,#3,#SF.SIZ,#REPLY,#1,#0,#0 ;Ask for size
        BCS     SPFERR                      ;failure
        .DPRINT ^/Block count = /,REPLY,DEC
        BR      START

CSIERR: .DPRINT  ^/?ESPFUN-F-CSI error/
        BR      EXIT

SPFERR: .DPRINT  ^/?ESPFUN-F-SPFUN error/
        .BR     EXIT
EXIT:   .EXIT

DEFEXT: .WORD    0,0,0,0               ;No default extensions

REPLY:  .BLKW     1            ;size

AREA:   .BLKW      10.         ;EMT block area

DEVSPC: .END     START
```

# .SPND/.RSUM

**EMT 374, Code 1/Code 2**

The .SPND/.RSUM requests control execution of a job's mainline code; that is, the code that is not executing as a result of a completion routine:

- .SPND suspends the mainline and allows only completion routines (for I/O and mark time requests) to run.
- .RSUM (from a completion routine) resumes the mainline code.

These functions enable a program to wait for a particular I/O or mark time request by suspending the mainline code and having the selected event's completion routine issue a .RSUM. This differs from the .WAIT request, which suspends the mainline code until all I/O operations on a specific channel have completed.

Macro Calls:

**.SPND**
**.RSUM**

Request Formats:

| | | |
|---|---|---|
| (.SPND) R0 = | 1 | 0 |

| | | |
|---|---|---|
| (.RSUM) R0 = | 2 | 0 |

**Notes**

- The monitor maintains a suspension counter for each job. This counter is decremented by .SPND and incremented by .RSUM. A job is suspended only if this counter is negative. Thus, if a .RSUM is issued before a .SPND, the latter request returns immediately.
- A program must issue an equal number of .SPND and .RSUM requests.
- A .RSUM request from the mainline code increments the suspension counter.
- A .SPND request from a completion routine decrements the suspension counter, but does not suspend the mainline. If a completion routine does a .SPND, the mainline continues until it also issues a .SPND, at which time it is suspended and requires two .RSUMs to proceed.
- 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 mainline to continue past a timed wait before the entire time interval has elapsed.
- 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:
See RCVDC.

# .SRESET

**EMT 352**

Macro Call:

**.SRESET**

The .SRESET (software reset) request:

- Cancels any messages sent by the job.
- Waits for all job I/O to complete, which includes waiting for all completion routines to run.
- Removes from memory any device handlers brought into memory via .FETCH calls by this job. Handlers loaded by the keyboard monitor LOAD command remain resident, as does the system device handler.
- Purges any currently open files. Files opened for output with .ENTER are never made permanent.
- Reverts to using only $16_{10}$ I/O channels. Any channels defined with .CDFN are discarded. A .CDFN must be reissued to open more than 16 channels after a .SRESET is performed.
- Clears the job's .SPND/.RSUM counter.
- Resets the I/O queue to one element. A .QSET request must be reissued to allocate extra queue elements.
- Cancels all outstanding .MRKT requests.

Errors:
None.

Example:

See .RENAME.

# .SYNCH

**Macro Expansion**

This macro call enables your program to issue programmed requests from an interrupt service routine. Code following the .SYNCH call runs at priority level 0 as a completion routine in the issuing job's context. Programmed requests issued from interrupt routines are not supported by the system and should not be performed unless a .SYNCH is used. .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 you must set aside for use by .SYNCH. This argument, area, represents a special seven-word block used by .SYNCH as a queue element. 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 Offset | Contents |
|---|---|
| 0 | RT–11 maintains this word; its contents should not be altered by the user |
| 2 | The current job's number. Must be set up by the user program. Obtained by a .GTJB call or from the I/O queue element in a device handler |
| 4 | Unused. |
| 6 | Unused. |
| 10 | R0 contains this argument after successful return. |
| 12 | Must be -1. |
| 14 | Must be 0. |

**pic** is an optional argument that, if non-blank, causes 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. After failure of the .SYNCH routine, the routine is still unable to issue programmed requests, and R4 and R5 are available for use. An error is returned if another .SYNCH that specified the same seven-word block is still pending.

### Notes

The monitor dismisses the interrupt without returning to the .SYNCH routine if one of the following conditions occur:

- You specified an invalid job number.
- The job number does not exist (for example, you specify 2, and there is no foreground job).
- The job is exited or terminated with an .EXIT programmed request.

You can find out if the block is in use by:

- Checking location QS.CUP (offset $14_8$). If this location contains a zero, the block is available.
- Performing a .SYNCH call. If the block is busy, an error return will be performed.

Normal return is to the word after the error return. At this point, the routine is in user state and is thus allowed to issue programmed requests. R0 contains the argument that was in offset $10_8$ of the block. R0 and R1 are free for use 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. Enter a RETURN to exit from the routine.

Example:

```
        .TITLE  SYNCH.MAC

;+
; .SYNCH - This is an example of the .SYNCH request.
; The example is a skeleton of a program which could input data
; from the outside world by means of an in-line interrupt service routine,
; buffer it until a whole block's worth has been input, then use
; a .WRITE request to store the data on an RT-11 device.
;-
        .MCALL  .GTJB,.INTEN,.WRITE,.WAIT,.SYNCH,.EXIT,.PRINT
```

```
START:  MOV     #JOB,R5                 ;Results of .GTJB go here
        .GTJB   #AREA,R5                ;Get job number (either FG or BG)
        MOV     (R5),SYNBLK+2           ;Store job number in .SYNCH block
        ;       .                       ;Here we open an RT-11 output
        ;       .                       ;device, then initiate input from
        ;       .                       ;a "foreign" device, interrupts to
        ;       .                       ;be handled by our in-line interrupt
        ;       .                       ;service routine....
INTRPT:                                 ;INTERRUPT SERVICE ROUTINE
        .INTEN  5                       ;Notify RT-11 and drop to priority 5
        ;       .                       ;Process interrupt and buffer input
        ;       .                       ;Time to write a buffer - switch
        ;       .                       ;buffers (should be double buffered
        ;       .                       ;so that interrupt processing can
        ;       .                       ;continue during write operation).
        .SYNCH  #SYNBLK                 ;Do a .SYNCH so we can use a .WRITE
        BR      SYNFAIL                 ;Return here if a .SYNCH block in use
        ;       .                       ;Return here if okay...
        .WAIT   #1                      ;See if error on last write
        BCS     WRFAIL                  ;Branch if there was
        .WRITE  #AREA,#1,OBUFF,#256.,BLK
                                        ;Queue a write to store the data
        INC     BLK                     ;and bump the block number.
        RETURN                          ;Re-enable interrupts and leave

SYNBLK: .WORD   0                       ;.SYNCH block
        .WORD   0                       ;Job number goes here
        .WORD   0                       ;Next two words reserved
        .WORD   0                       ;
        .WORD   5                       ;R0 contains 5 on successful .SYNCH
        .WORD   -1,0                    ;Required values for the Monitor
SYNFAIL:                                ;.SYNCH failed...
        ;       .                       ;This can be a problem if the
        ;       .                       ;next interrupt came in before the
        ;       .                       ;buffer was written out!

WRFAIL: MOV     #WERR,R0                ;R0 -> error message text
ERRM:   .PRINT                          ;Output the error message
        .EXIT                           ;then exit.

BLK:    .WORD   0                       ;Block number to write
AREA:   .BLKW   5                       ;EMT Argument block
JOB:    .BLKW   8.                      ;Area for .GTJB data
OBUFF:  .WORD   0                       ;Pointer to current output buffer
IBUFF:  .WORD   0                       ;Pointer to current input buffer
BUFF1:  .BLKW   256.                    ;Buffer 1
BUFF2:  .BLKW   256.                    ;Buffer 2
WERR:   .ASCIZ  "?ESYNCH-F-Write Error"
SYNERR: .ASCIZ  "?ESYNCH-F-SYNCH Failed"
        .EVEN

        .END    START
```

# .TIMIO

**Macro Expansion**

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

Macro Call:

**.TIMIO tbk,hi,lo**

where:

**tbk** is the address of the timer block, a seven-word timer queue element. (See timer block format shown under the .CTIMIO request.) You must set up the address of the completion routine in the seventh word of the timer block in a position-independent manner.

**hi** is the high-order word of a two-word time interval

**lo** is the low-order word of a two-word time interval

Example:

```
        .TITLE  TIMIO.MAC

;+
; TIMIO.MAC - This is an example of a simple, RT-11 device driver,
; to illustrate the use of the .TIMIO/.CTIMIO requests. The timeout
; completion routine will be entered if a character hasn't been
; successfully transmitted in 1/10 sec (approx. 110 baud). In this
; example the completion routine takes no explicit action; the fact
; that the timeout occurred is enough to be considered a "hard" error.
;-

        .MCALL  .DRBEG,.DRAST,.DRFIN,.DREND,.QELDF,.TIMIO,.CTIMIO

.IIF NDF MMG$T, MMG$T=0                 ;Define these in case not
.IIF NDF ERL$G, ERL$G=0                 ;assembled with SYSCND.MAC
.IIF NDF RTE$M, RTE$M=0                 ;...
TIM$IT=1

.IIF NDF SP$VEC, SP$VEC=304             ;Define default vector
.IIF NDF SP$CSR, SP$CSR=176504          ;Define default CSR addr
.IIF NDF SP$PRI, SP$PRI=4               ;Define default device priority

        IOERR = 1 ;Hard I/O error bit definition
        SPSTS = 20000                   ;Device Status = Write only
        SPSIZ = 0 ;Device Size = 0 (Char device)
        TIME = 6                        ;Timeout interval = 1/10 sec
        COD = 377                       ;Device i.d. code
```

```
        .QELDF                          ;Use .QELDF to define Q-Elem offsets
        .DRBEG  SP,SP$VEC,SPSIZ,SPSTS   ;Begin driver code with .DRBEG
        MOV     SPCQE,R4                ;R4 => Current Q-Element
        ASL     Q$WCNT(R4)              ;Make word count byte count
        BCC     SPERR                   ;A read from a write/only device?
        BEQ     SPDUN                   ;Zero word count...just exit
SPRET:  MOV     PC,R5                   ;Calculate PIC address
        ADD     #SPTOUT-.,R5            ;completion routine
        MOV     R5,TBLK+14              ;Move it to argument block
        .TIMIO  TBLK,0,TIME             ;Schedule a marktime
        BIS     #100,@#SP$CSR           ;Enable DL-11 interrupt
        RETURN                          ;Return to monitor

; INTERRUPT SERVICE ROUTINE

        .DRAST  SP,SP$PRI               ;Use .DRAST to define Int Svc Sect.
        MOV     SPCQE,R4                ;R4 => Q-Element
        TST     @#SP$CSR                ;Error?
        BMI     SPRET                   ;Yes...'hang' until ready
        TSTB    @#SP$CSR                ;Is device ready?
        BPL     SPRET                   ;No...go wait 'till it is
        .CTIMIO TBLK                    ;Cancel completion routine
        BCS     SPERR                   ;Too late - it timed out!
        MOVB    @Q$BUFF(R4),@#SP$CSR+2  ;Xfer byte from buffer to DL-11
        INC     Q$BUFF(R4)              ;Bump the buffer pointer
        INC     Q$WCNT(R4)              ;and the word count (it's negative!)
        BEQ     SPDUN                   ;Branch if done
        BR      SPRET                   ;Go wait 'till char xmitted

SPTOUT: ;       .                       ;Timeout completion routine
        ;       .                       ;In this example, it does nothing.
        ;       .                       ;In real life it may want to try
        RETURN                          ;to take some corrective action...

SPERR:  BIS     #IOERR,@Q$CSW(R4)       ;Set error bit in CSW
SPDUN:  .DRFIN  SP                      ;Use .DRFIN to return to Monitor

TBLK:   .WORD   0,TIME,0,0,177000+COD   ;.TIMIO argument block

        .DREND  SP                      ;Use .DREND to end code

        .END
```

# .TLOCK

**EMT 374, Code 7**

The .TLOCK (test lock) request is used in an multijob environment to attempt to gain ownership of the USR. It is identical to .LOCK in the single-job monitor. It is similar to .LOCK in that, if successful, the user job returns with the USR in memory. However, if a job attempts to .LOCK the USR while another 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.

Macro Call:

**.TLOCK**

Request Format:

| | | |
|---|---|---|
| R0 = | 7 | 0 |

Errors:

| Code | Explanation |
|---|---|
| 0 | USR is already in use by another job. |

Example:

```
        .TITLE  ETLOCK

;+
; .TLOCK - This is an example in the use of the .TLOCK request.
; 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 (that perhaps doesn't need the USR for execution).
; This type of procedure is useful to schedule several sub-jobs within
; a single background or foreground program.
;-

        .MCALL  .TLOCK,.LOOKUP,.UNLOCK,.EXIT,.PRINT

START:                                ;Begin Mainline program
        .TLOCK                        ;Try to get the USR for 1st "job"
        BCS     SUSPND                ;Failed...branch to "suspend" 1st job
        .LOOKUP #AREA,#4,#FILE        ;Succeeded...proceed with 1st job
        BCS     LKERR                 ;Branch if error on LOOKUP
        ;       .                     ;1st job involves file processing...do
                                      ;it!
        .PRINT  #J1MSG                ;Tell user we executed...
        .UNLOCK                       ;1st job finished...release USR
        TSTB    J2SW                  ;Check if we ran Job #2 while USR busy
        BNE     10$                   ;Yup - we did
        CALL    JOB2                  ;Nope - do it now
10$:    .EXIT
```

```
SUSPND:                                 ;"Suspend" current "job"
        TSTB    J2SW                    ;Did we already run Job #2
        BNE     START                   ;Yes - don't do it again
        JSR     PC,JOB2                 ;"Run" other "job"
        INC     J2SW                    ;Set switch that says we ran Job #2
        BR      START                   ;When it's finished, try 1st job again

LKERR:  .PRINT  #LKMSG                  ;Error on .LOOKUP - Report it!
        .EXIT

JOB2:   .PRINT  #J2MSG                  ;2nd "Job" - Doesn't need USR
        RETURN                          ;Return when done

AREA:   .BLKW   5                       ;EMT argument block
FILE:   .RAD50  "SRC"                   ;File spec for Job #1
        .RAD50  "ETLOCK"                ;
        .RAD50  "MAC"                   ;
LKMSG:  .ASCIZ  "?ETLOCK-F-File Not Found"
J1MSG:  .ASCIZ  "!ETLOCK-I-Job #1 Executed"
J2MSG:  .ASCIZ  "!ETLOCK-I-Job #2 Executed"
J2SW:   .BYTE   0                       ;Switch to control Job #2 execution
        .EVEN

        .END    START
```

# .TRPSET

**EMT 375, Code 3**

.TRPSET enables a 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 status 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 (check your processor handbook). These traps are not intercepted by the .TRPSET request, but they do cause job abort and a printout of the message *?MON-F-Trap to 4*.

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, trap interception is disabled. In a fully-mapped monitor, if you set the low bit of *addr* on, it will be treated as a Supervisor/Data space; otherwise, it is treated as a User/Data space address. |

Request Format:

| | | |
|---|---|---|
| R0 → area: | 3 | 0 |
| | addr | |

**Notes**

Reissue a .TRPSET request whenever an error trap occurs and the user routine is entered. The monitor disables user trap interception prior to entering the 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 mapped 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 error traps by trying to steal the trap vectors must be carefully designed to handle programs that are *virtual* jobs and those 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 bits 14 and 15 of the PS set to 1, user mode is set, and the trap routine executes in user mapping.

Errors:
None.

Example:

```
        .TITLE  TRPSET.MAC

;+
; .TRPSET - This is an example in the use of the .TRPSET request.
; In this example a user trap routine is set, then deliberate
; traps to 4 & 10 are caused (not very practical but it demonstrates
; that .TRPSET really works!).
;-
        .MCALL  .TRPSET,.EXIT,.PRINT

        DIVZ =: 67                      ;Divide by zero - illegal instruction

START:                                  ;Begin example
        .TRPSET #AREA,#TRPLOC           ;Set up a trap routine to handle traps
                                        ;to 4 & 10...
        DIVZ                            ;Illegal instruction - Trap to 10
        MOV     R0,R0                   ;Legal instruction
        TST     @#160000                ;Non-existent memory - Trap to 4
        .EXIT                           ;Exit program

TRPLOC:                                 ;Trap routine
        BCS     10$                     ;C bit set = TRAP 10
        .PRINT  #TRP4                   ;Report Trap to 4
        BR      20$                     ;Branch to reset trap routine
10$:    .PRINT  #TRP10                  ;Report trap to 10
        .TRPSET #AREA,#TRPLOC           ;Reset trap routine address
20$:    RTI                             ;Return to offending code

AREA:   .WORD   0,0                     ;EMT argument block
TRP4:   .ASCIZ  "?ETRPSE-W-Trap to 4"
TRP10:  .ASCIZ  "?ETRPSE-W-Trap to 10"

        .END    START
```

# .TTYIN/.TTINR

**EMT 340**

The requests .TTYIN and .TTINR transfer a character from the console terminal to the user program. The character thus obtained appears right-justified (low byte) in R0. The user can cause the characters to be returned in R0 only, or in R0 and another location.

Macro Call:

**.TTYIN char**

**.TTINR**

where:

**char** is the location where the character in R0 is to be stored.

If *char* is specified, the character is in both R0 and the address represented by char. If char is not specified, the character is in R0

The expansion of .TTYIN is:

```
EMT     340
BCS     .-2
```

The expansion of .TTINR is:

```
EMT     340
```

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. .TTINR does not return the carry bit set, unless TCBIT$ of the job status word (JSW) was on when the request was issued.

The choice of two modes of doing console terminal input is determined by TTSPC$ of the job status word. If TTSPC$ is 0, normal I/O is performed. In this mode, the following conditions apply:

- The monitor echoes all characters typed.
- CTRL/U and the DELETE key perform line deletion and character deletion, respectively.
- A carriage return, line feed, CTRL/Z, or CTRL/C must be typed before characters on the current line are available to the program. As you type these, characters on the line are sequentially passed to the user program.

If TTSPC$ is 1, the console is in special mode. The effects are:

- The monitor does not echo characters typed except for CTRL/C and CTRL/O.
- CTRL/U and the DELETE key do not perform special functions.
- Characters are immediately available to the program.

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. TTSPC$ in the JSW must be set by the user program. This bit is cleared when the program terminates.

Regardless of the setting of TTSPC$, when a carriage return is entered, both carriage return and line feed characters are passed to the program; if TTSPC$ is 0, these characters will be echoed.

Lowercase conversion is determined by the setting of TTLC$ in the JSW:

- If TTLC$ is 0, lowercase characters are converted to uppercase before being echoed (if TTSPC$ is 0) and passed to a program.
- If TTLC$ is 1, lowercase characters are echoed (if TTLC$ is 0) and passed as received. TTLC$ is cleared when the program terminates.

CTRL/F and CTRL/B (and CTRL/X in system job monitors) are not affected by the setting of TTSPC$. 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 the SET TT NOPAGE command is issued.

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 TCBIT$ of the Job Status Word before the .TTINR request. TCBIT$ is cleared when a program terminates.

If the single-line editor has been enabled by the commands SET SL ON and SET SL TTYIN, and if EDIT$ and TTSPC$ of the JSW are 0, input from a .TTYIN or .TTINR request will be edited by SL. If either EDIT$ or TTSPC$ is set, SL will not edit input. If SL is editing input, the state of TCBIT$ (inhibit TT wait) is ignored and a .TTINR request will not return until an edited line is available.

**NOTE**

The .TTYIN request does not get characters from indirect files. If this function is desired, the .GTLIN request must be used.

Errors:

| Code | Explanation |
|---|---|
| 0 | No characters available in ring buffer. |

Example:
See .TTYOUT/.TTOUTR.

# .TTYOUT/.TTOUTR

**EMT 341**

The requests .TTYOUT and .TTOUTR cause a character to be transmitted 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 is not output.

Macro Call:

**.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

The expansion of .TTYOUT is:

```
EMT     341
BCS     .-2
```

The expansion of .TTOUTR is:

```
EMT     341
```

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. .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 TCBIT$ of the Job Status Word before issuing the request.

The .TTINR and .TTOUTR requests have been supplied to help those users who want to continue rather than 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.

If a foreground job leaves TCBIT$ set in the Job Status Word, any further foreground .TTYIN or .TTYOUT requests cause the system to lock out the background until a character is available. Note also that each job in the foreground/background environment has its own Job Status Word, and therefore can be in different terminal modes independently of the other job.

Errors:

| Code | Explanation |
|---|---|
| 0 | Output ring buffer full. |

Example:

```
        .TITLE  ETTYIN

;+
; .TTYIN / .TTYOUT - This is an example in the use of the .TTYIN
; & .TTYOUT requests. The example accepts a line of input from the
; console keyboard, then echoes it on the terminal. Using .TTYIN &
; .TTYOUT requests illustrate Synchronous terminal I/O; i.e., the
; Monitor retains control (the job is blocked) until the requests
; are satisfied.
;-

        .MCALL  .TTYIN,.TTYOUT

START:  MOV     #BUFFER,R1      ;R1 => Character buffer
        CLR     R2              ;Clear character count
INLOOP: .TTYIN  (R1)+           ;Read char into buffer
        INC     R2              ;Bump count
        CMPB    #12,R0          ;Was last char a LF ?
        BNE     INLOOP          ;No...get next character
        MOV     #BUFFER,R1      ;Yes...point R1 to beginning of buffer
OUTLOOP:.TTYOUT (R1)+           ;Print a character
        DEC     R2              ;Decrease count...
        BEQ     START           ;Done if count = 0
        BR      OUTLOOP         ;Loop to print another character
BUFFER: .BLKW   64.             ;Character buffer...

        .END    START
```

```
        .TITLE  ETTINR;
;+
; .TTINR / .TTOUTR - This is an example in the use of the .TTINR &
; .TTOUTR requests. Like ETTYIN.MAC, this example accepts lines of
; input from the console keyboard, then echoes it on the terminal.
; 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 to allow
; other processing to be carried out if a wait condition is reached.
;-
        .MCALL  .TTYIN,.TTYOUT,.RCTRLO
        .MCALL  .TTINR,.TTOUTR

        $JSW =: 44                  ;(.SYCDF)Location of Job Status Word
         TCBIT$ =: 100              ;(.JSWDF)Nowait bit in JSW

START:  MOV     #BUFFER,R1          ;Point R1 to buffer
        CLR     R2                  ;Clear character count
        BIS     #TCBIT$,@#$JSW      ;Set Bit in JSW so .TTINR/.TTOUTR will
        .RCTRLO                     ;return C bit set if no char/no room
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 char
        MOV     #BUFFER,R1          ;Yes...point R1 to beginning of buffer
```

```
OUTLOOP:MOVB    (R1),R0             ;Put char in R0
        .TTOUTR                     ;Try to print it
        BCS     NOROOM              ;Branch if no room in output buffer
CHROUT: DEC     R2                  ;Decrease count
        BEQ     START               ;Done if count=0
        INC     R1                  ;Bump buffer pointer
        BR      OUTLOOP             ;then branch to print next char

NOCHAR:                             ;Comes here if no char avail
        .TTINR                      ;try to again to get one
        BCC     CHRIN               ;There's one avail this time!
        ;       .                   ;Do other processing
        BR      NOCHAR              ;Try again

NOROOM:                             ;Comes here if no room in buffer
        MOVB    (R1),R0             ;Put char in R0
        .TTOUTR                     ;Try to print it again
        BCC     CHROUT              ;Successful !
        ;       .                   ;Code to be executed while waiting
        ;       .                   ;Now we must hang to wait...
        BIC     #TCBIT$,@#$JSW      ;Clear bit in JSW
        .RCTRLO
        .TTYOUT (R1)                ;Use .TTYOUT to wait for room
        BIS     #TCBIT$,@#$JSW      ;Finally successful - reset bit
        .RCTRLO
        BR      CHROUT              ;then return to output loop

BUFFER: .BLKW   64.                 ;Buffer
        .END    START
```

# .TWAIT

**EMT 375, Code 24**

Support for this request must be selected at SYSGEN. The .TWAIT request suspends the user's job for a specified length of time. .TWAIT requires a queue element and should be a consideration when the .QSET request is issued.

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, if your system has a line frequency clock. This must be kept in mind when the time interval is specified. Check CLK50$ (.CF1DF) in $CNFG1 (.FIXDF) to see if this bit is set to 50Hz; if not, the frequency is 60 Hz.

Errors:

| Code | Explanation |
|---|---|
| 0 | No queue element was available. |

Example:

```
        .TITLE  TWAIT.MAC

;+
; .TWAIT - This is an example in the use of the .TWAIT request.
; .TWAIT is useful in applications where a program must be only
; activated periodically. This example will 'wake up' every five seconds
; to perform a simulated "task", and then 'sleep' again. (For example
; purposes this cycle will be repeated for a maximum of about 7 sec).
;-

        .MCALL  .TWAIT,.EXIT,.PRINT

START:  CALL    TASK                    ;Perform task...
10$:     .TWAIT  #AREA,#TIME             ;Go to sleep for a second
        BCS     NOQ                     ;Branch if no queue element
        CALL    TASK                    ;Perform task again
        DEC     COUNT                   ;Bump counter - example good for 7 sec
        BNE     10$                      ;Branch if time's not up
        .PRINT  #BYE                    ;Say we're thru
        .EXIT                           ;Exit program

TASK:                                   ;Periodic task simulated here
        INC     TCNT                    ;Bump a counter
        BIT     #1,TCNT                 ;Is it odd?
        BEQ     10$                      ;Branch if not
        .PRINT  #TICK                   ;Odd counter prints "tick..."
        RETURN                          ;Return to caller

10$:     .PRINT  #TOCK                   ;Even counter prints "tock"
        RETURN                          ;Return to caller

NOQ:    .PRINT  #QERR                   ;Print error message
        .EXIT                           ;Exit program

AREA:   .WORD   0,0                     ;EMT Argument block
TIME:   .WORD   0,60.*1                 ;60 ticks/sec * 1 seconds
COUNT:  .WORD   7                       ;Maximum cycles for example
TCNT:   .WORD   0                       ;Tick,tock count
TICK:   .ASCII  "Tick..."
        .BYTE   200
TOCK:   .ASCIZ  "Tock"
BYE:    .ASCIZ  "!ETWAIT-I-Example Concluded"
QERR:   .ASCIZ  "?ETWAIT-F-No Q-Element Available"
        .END    START
```

# .UNLOCK, .UNMAP, .UNPROTECT

See .LOCK, .MAP, .PROTECT.

# .WAIT

**EMT 374, Code 0**

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 through its error returns. An error is returned if the channel is not open or the last I/O operation resulted in a hardware error.

If an asynchronous operation on a channel results in end-of-file, the following .WAIT programmed request will not detect it. The .WAIT request detects only hard error conditions. A subsequent operation on that channel will detect end-of-file and will return to the user immediately with the carry bit set and the end-of-file code in byte 52 ($ERRBY). Under these conditions, the subsequent operation is not initiated.

In a multijob system, executing a .WAIT when I/O is pending causes that job to be suspended and another 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:
See .READ.

# .WDBBK

### Macro Expansion

The .WDBBK macro defines symbols for the window definition block and reserves space for it. Information provided to the arguments of this macro permits the creation and mapping of a window through the use of the .CRAW request. Note that .WDBBK automatically invokes .WDBDF.

Macro Call:

**.WDBBK wnapr,wnsiz[,wnrid][,wnoff][,wnlen][,wnsts]**

where:

**wnapr** is the number of the Active Page Register set that includes the window's base address. A window must start on a 4K-word boundary. The valid range of values is from 0 through 7

**wnsiz** is the size of this window (expressed in 32-word units)

**wnrid** is the identification for the region to which this window maps. This argument is optional; supply it if you need to map this window. Use the value of R.GID from the region definition block for this argument after you create the region to which this window must map

**wnoff** is the offset into the region at which to start mapping this window (expressed in 32-word units). This argument is optional; supply it if you need to map this window. The default is 0, which means that the window starts mapping at the region's base address

**wnlen** is the amount of this window to map (expressed in 32-word units). This argument is optional; supply it if you need to map this window. The default value is 0, which maps as much of the window as possible

**wnsts** is the window status word. This argument is optional; supply it if you need to map this window when you issue the .CRAW request. Set bit 8, called WS.MAP, to cause .CRAW to perform an implied mapping operation

Example:

See .CRAW. See also the *RT–11 System Internals Manual* for a detailed description of the extended memory feature.

# .WDBDF

**Macro Expansion**

The .WDBDF macro defines the symbolic offset names for the window definition block and the names for the window status word bit patterns. In addition, this macro also defines the length, but doesn't reserve any space for the definition block. (See .WDBBK).

Macro Call:

**.WDBDF**

The .WDBDF macro expands as follows:

```
W.NID   =: 0            ;Window ID
W.NAPR  =: 1            ;PAR number
W.NBAS  =: 2.           ;Base address
W.NSIZ  =: 4.           ;Window size
W.NRID  =: 6.           ;Region ID
W.NOFF  =: ^o10         ;Window offset
W.NLEN  =: ^o12         ;Window length
W.NSTS  =: ^o14         ;Window status
W.NLGH  =: ^o16         ;Length of WDB
WS.CRW  =: ^o100000     ;Window created
WS.UNM  =: ^o40000      ;One or more windows unmapped
WS.ELW  =: ^o20000      ;One or more windows eliminated
WS.DSI  =: ^o10000      ;D-space inactive WS.D=WS.I=1
WS.IDD  =: ^o4000       ;D-space window different WS.D=WS.I=1
WS.OVR  =: ^o2000       ;reserved
WS.RO   =: ^o1000       ;Window is read only
WS.MAP  =: ^o400        ;Create and map window
WS.SPA  =: ^o14         ;Bit field for address space(s)
WS.D    =: ^o10         ;Map into data space
WS.I    =: ^o4          ;Map into instruction space
WS.MOD  =: ^o3          ;Field to indicate mode
WS.U    =: ^o0          ;User
WS.S    =: ^o1          ;Supervisor
WS.C    =: ^o2          ;Current
```

# .WRITE/.WRITC/.WRITW

**EMT 375, Code 11**

The three modes of write operations for RT–11 I/O use the .WRITE, .WRITC, and .WRITW programmed requests.

Note that in the case of .WRITE and .WRITC, additional queue elements should be allocated for buffered I/O operations (See .QSET programmed request).

Under a monitor the with system job feature, .WRITE/C/W requests may be used to send messages to other jobs in the system.

**.WRITE**

The .WRITE request transfers a specified number of words from memory to the specified channel. Control returns to your program immediately after the request is queued.

Macro Call:

**.WRITE area,chan,buf,wcnt,blk[,BMODE=strg]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **chan** | is a channel number in the range 0 to $376_8$ |
| **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. Some devices, such as LP, may assign the *blk* argument special meaning. For example, if *blk* = 0, LP issues a form feed. |

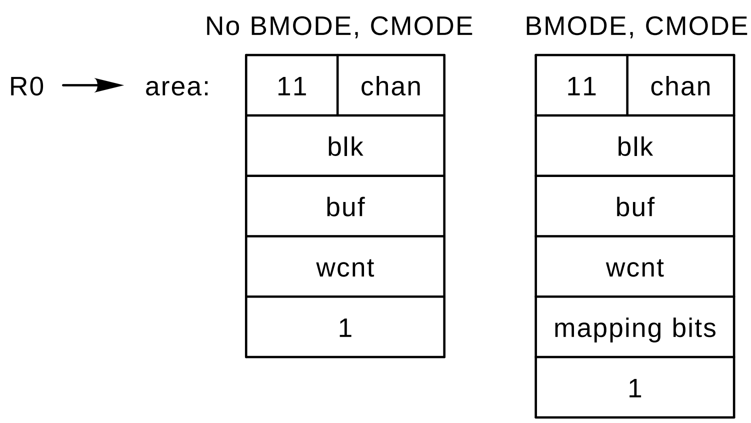
**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:




**Notes**

.WRITE and .WRITC instruct the monitor to do a write to a sequential- access device (for example, magtape), then immediately return control to your program.

If the write is to a random-access device (disk), R0 contains the number of words that will write (.WRITE or .WRITC) or have been written (.WRITW). If a write goes past EOT on magtape, an error is returned and R0=0. 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).

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.

Errors:

| Code | Explanation |
|---|---|
| 0 | Attempted to write past end-of-file. |
| 1 | Hardware error. |
| 2 | Channel was not opened. |

Example:
See .READ.

#### .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 request is fulfilled, then control passes to the routine specified in the request. When the completion routine executes a RETURN instruction, control returns to the user program.

Macro Call:

**.WRITC area,chan,buf,wcnt,crtn,blk[,BMODE][,CMODE]**

where:

| | |
|---|---|
| **area** | is the address of a five-word EMT argument block |
| **chan** | is a channel number in the range 0 to $376_8$ |
| **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 (>500 octal) of the completion routine to be entered |
| **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. Your program should normally update *blk* before it is used again. See the *RT–11 Device Handlers Manual* for the significance of the block number for devices such as line printers. |

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

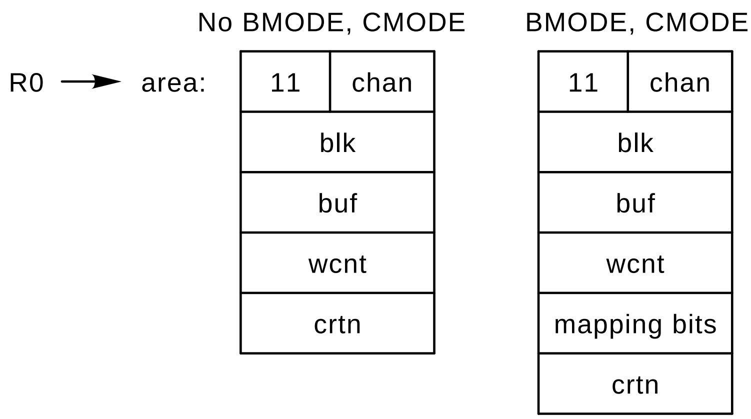
**CMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| U | User space (default) |
| S | Supervisor space |

Specifying *CMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *CMODE*.
- Specifies the *space* for the *crtn* argument.
- Is a valid option only for fully mapped monitors.

Request Format:




**Notes**

.WRITC instructs the monitor to do a write to a sequential- access device (for example, magtape), then immediately returns control to your program.

If the write is to a random-access device (disk), R0 contains the number of words that will write (.WRITE or .WRITC) or have been written (.WRITW). If a write goes past EOT on magtape, an error is returned and R0=0. 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).

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. When a .WRITC completion routine is entered, the following conditions are true:

- R0 contains the contents of the channel status word for the operation. If bit 0 (HDERR$) of R0 is set, a hardware error occurred during the transfer and data may be unreliable.
- 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.
- 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 example following .READC.

### .WRITW

The .WRITW request transfers a specified number of words from memory to the specified channel. Control returns to your program when the .WRITW is complete.

Macro Call:

**.WRITW area,chan,buf,wcnt,blk[,BMODE=strg]**

where:

**area** is the address of a five-word EMT argument block

**chan** is a channel number in the range 0 to $376_8$

**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. Your program should normally update *blk* before it is used again. See the *RT–11 Device Handlers Manual* for the significance of the block number for devices such as line printers.

**BMODE = strg** where *strg* is:

| Value | Description |
|---|---|
| UD | User data space (default) |
| UI | User instruction space |
| SD | Supervisor data space |
| SI | Supervisor instruction space |
| CD | Kernel data space |
| CI | Kernel instruction space |

Specifying *BMODE*:

- Loads an additional word in the EMT request block area containing a bit pattern matching the code specified for *BMODE*.
- Specifies *mode* and *space* for the *buff* argument.
- Is a valid option only in a fully mapped environment.

Request Format:

R0 → area:

No BMODE

| | |
|---|---|
| 11 | chan |
| blk | |
| buf | |
| wcnt | |
| 0 | |

BMODE

| | |
|---|---|
| 11 | chan |
| blk | |
| buf | |
| wcnt | |
| mapping bits | |
| 0 | |

**Notes**

.WRITW instructs the monitor to do a write to a sequential- access device (for example, magtape) or to a random-access device (disk), then returns control to your program.

If the write is to a random-access device (disk), R0 contains the number of words that 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.

If a write on magtape goes past EOT, an error is returned and R0=0. 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. |

Example:
See .READW.

## Appendix A

# Summary of Added and Changed Functionality

Table A–1 lists all new and changed program requests for previous releases of the RT–11 Operating System, Version 4.0 through Version 5.6.

**Table A–1: Summary of Added and Changed Functionality**

| Request | Action | Description |
|---|---|---|
| **Version 4.0** | | |
| .CHCOPY | Changed | Jobname parameter |
| .DRBOT | Added | Define handler boot area |
| .DRDEF | Added | Define handler information |
| .DRSET | Added | Generate SET table entry |
| .DRVTB | Added | Generate vector table entry |
| .GTJB | Changed | 12-word return and jobname parameter |
| .MTSTAT | Added | Return system-wide information about multiterminal |
| .SDTTM | Added | Set system date and/or time |
| .SPCPS | Added | Derail to specified address |
| **Version 5.0** | | |
| .ABTIO | Added | Abort I/O on a channel |
| .AUDIT | Added | <Reserved to Digital> |
| .DRBOT | Changed | Added CONTROL and SIDES parameters |
| .DREND | Changed | Added FORCE parameter |
| .FETCH | Changed | Available under XM |
| .FPROT | Added | Protect file from deletion |
| .GTLIN | Changed | Added TYPE parameter |
| .MODULE | Added | <Reserved to Digital> |
| .NLCSI | Added | <Reserved to Digital> |
| .PEEK | Added | Access Kernel memory |
| .POKE | Added | Modify Kernel memory |

**Table A–1 (Cont.): Summary of Added and Changed Functionality**

| Request | Action | Description |
|---|---|---|
| **Version 5.0** | | |
| .PVAL | Added | Modify monitor |
| .RMODULE | Added | <Reserved to Digital> |
| .SFDAT | Added | Set file date |
| .TWAIT | Changed | Available under SJ |
| **Version 5.1** | | |
| .ADDR | Added | Generate PIC address |
| .ASSUME | Added | Check assembly assumption |
| .BR | Added | Check assembly assumption on "drop-through" |
| .DREND | Changed | Added PSECT parameter |
| .DRINS | Added | Set up install area for handlers |
| .DRVTB | Changed | Added *slotid* for PRO support |
| .SCCA | Changed | Added TYPE parameter |
| SOB | Added | Simulate SOB instruction on old machines |
| **Version 5.2** | | |
| .CKxx | Added | Check register contents assumptions |
| .DRBOT | Changed | Added OFFSET parameter |
| .DREND | Changed | Made some symbols local by default (L1 and L2) |
| .MODULE | Changed | Provided default for VERSION parameter |
| .RDBBK | Changed | Added NAME parameter |
| **Version 5.3** | | |
| .CKxx | Changed | Documented |
| .DREST | Added | Supply extended handler status |
| .DRPTR | Added | Added (later deleted) FORMAT and SHOW parameters |
| .DRSPF | Added | Generate .SPFUN code table(s) in handlers |

**Table A–1 (Cont.): Summary of Added and Changed Functionality**

| Request | Action | Description |
|---|---|---|
| **Version 5.4** | | |
| .DRBEG | Changed | Added ADDRCK,SPFUN, CODE, and L1 parameters |
| .DRBOT | Changed | Added FORCE, and PSECT parameters |
| .DRDEF | Changed | Added UNIT64, DMA, and PERMUMR parameters |
| .DREST | Changed | Added STAT2 parameter |
| .DRFMS | Added | <Reserved to Digital> |
| .DRFMT | Added | <Reserved to Digital> |
| .DRPTR | Changed | Deleted FORMAT and SHOW parameters |
| .DRTAB | Added | <Reserved to Digital> |
| .DRUSE | Added | Point to table(s) in handlers |
| .HERR | Changed | Added return of previous .HERR/.SERR state |
| .RDBBK | Changed | Added base parameter |
| .SERR | Changed | Added return of previous .HERR/.SERR state |
| **Version 5.5** | | |
| .CALLK | Added | Transfer control to Kernel mapping routine |
| .CLOSZ | Added | Specify size on new file closure |
| .DRBEG | Changed | Added LDTBL and NSPFUN parameters |
| .DRDEF | Changed | Added SERIAL parameter |
| .DRSPF | Changed | Added TYPE parameter |
| .FPROT | Changed | Return previous setting status |
| .GFDAT | Added | Return file date |
| .GFINF | Added | Return file directory entry information |
| .GFSTAT | Added | Return file directory status |
| .PEEK | Changed | Address 177776 accesses PS |
| .POKE | Changed | Added TYPE parameter |
| .POKE | Changed | Address 177776 accesses PS |
| .PVAL | Changed | Added TYPE parameter |
| .QELDF | Changed | Added definition of Q.MEM and Q$MEM |
| .SFINF | Added | Set file directory entry information |
| .SFSTAT | Added | Set file directory status |

**Table A–1 (Cont.): Summary of Added and Changed Functionality**

| Request | Action | Description |
|---|---|---|
| **Version 5.6** | | |
| .CALLS | Added | Transfer control to Supervisor mapping routine |
| .CKxx | Changed | Added LIST parameter |
| .CMAP | Added | Control mode and space mapping |
| .CMPDF | Added | Define masks for .CMAP, .GCMAP and .MSDS |
| .DEBUG | Added | Setup for .DPRINT |
| .DPRINT | Added | Display debugging information |
| .DPSEC | Added | <Reserved to Digital> |
| .GCMAP | Added | Return settings of mode and space mapping |
| .MSDS | Added | Control User / Supervisor D-space locking |
| .QELDF | Changed | Add LIST and E parameters; define more bits |
| .RCVD | Changed | Added BMODE parameter |
| .RCVDC | Changed | Added BMODE and CMODE parameters |
| .RCVDW | Changed | Added BMODE parameter |
| .RDBDF | Changed | Added LIST and E parameters |
| .READ | Changed | Added BMODE parameter |
| .READC | Changed | Added BMODE and CMODE parameters |
| .READW | Changed | Added BMODE parameter |
| .SDAT | Changed | Added BMODE parameter |
| .SDATC | Changed | Added BMODE and CMODE parameters |
| .SDATW | Changed | Added BMODE parameter |
| .SPFUN | Changed | Added BMODE and CMODE parameters |
| .WDBDF | Changed | Added LIST and E parameters |
| .WRITC | Changed | Added BMODE and CMODE parameters |
| .WRITE | Changed | Added BMODE parameter |
| .WRITW | Changed | Added BMODE parameter |

# Index

## A


.ABTIO
  summary, 1–33
.ABTIO programmed request
  description, 2–3
  EMT 374, Code 13, 2–3
.ADDR
  summary, 1–33
.ADDR macro
  description, 2–4
Argument summary
  list, 2–1
.ASSUME
  summary, 1–33
.ASSUME macro
  description, 2–5


## B


.BR
  summary, 1–33
.BR macro
  description, 2–6


## C


.CALL
  summary, 1–33
.CALLK programmed request
  Description, 2–7
  EMT 373, 2–7
.CALLS
  summary, 1–33
.CALLS macro
  description, 2–9
.CDFN
  summary, 1–34
.CDFN programmed request
  description, 2–11
  EMT 375, Code 15, 2–11
CHAIN.
  summary, 1–34
.CHAIN programmed request
.CHAIN programmed request (Cont.)
  Description, 2–13
  EMT 374, Code 10, 2–13
.CHCOPY
  summary, 1–40
.CHCOPY programmed request
  description, 2–16
  EMT 375, Code 13, 2–16
.CKxx
  summary, 1–34
.CKXX macro
  description, 2–19
.CLOSE
  summary, 1–34
.CLOSE programmed request
  description, 2–22
  EMT 374, Code 6, 2–22
.CLOSZ
  summary, 1–34
.CLOSZ programmed request
  description, 2–24
  EMT 375, Code 45, 2–24
.CMAP
  summary, 1–40
.CMAP macro
  description, 2–26
  mapping context (I.CMAP), 2–27
.CMKT
  summary, 1–34
.CMKT programmed request
  description, 2–32
  EMT 375, Code 23, 2–32
.CNTXSW
  summary, 1–40
.CNTXSW programmed request
  description, 2–34
  EMT 375, Code 33, 2–34
Communications
  See also .SDAT and .RCVD
  foreground/background, 1–26
Consistency checking
  .ASSUME, 1–33

Consistency checking (Cont.)
.Ck.Rn, 1–33
.CKxx, 1–33
.CRAW
summary, 1–40
.CRAW programmed request
description, 2–36
EMT 375, Code 36, 2–36
.CRRG
summary, 1–40
.CRRG programmed request
description, 2–40
EMT 375, Code 36, 2–40
.CSIGEN
summary, 1–34
.CSIGEN programmed request
description, 2–41
EMT 344, 2–41
.CSISPC
summary, 1–34
.CSISPC programmed request
description, 2–47
EMT 345, 2–47
.CSTAT
summary, 1–34
.CSTAT programmed request
description, 2–51
EMT 375, Code 27, 2–51
.CTIMIO
summary, 1–34
.CTIMIO macro
description, 2–53

## D

.DATE
summary, 1–34
.DATE programmed request
description, 2–55
EMT 374, Code 12, 2–55
.DEBUG
summary, 1–34
.DEBUG macro
description, 2–57
.DELETE
summary, 1–34
.DELETE programmed request
description, 2–60
EMT 375, Code 0, 2–60
.DEVICE
.DEVICE (Cont.)
summary, 1–34
Device handlers, 1–31
Device lookup
bypass, 1–32
.DEVICE programmed request
description, 2–61
EMT 375, Code 14, 2–61
.DPRINT macro
description, 2–59
.DRAST
summary, 1–34
.DRAST macro
description, 2–64
.DRBEG
summary, 1–34
.DRBEG macro
description, 2–67
.DRBOT
summary, 1–34
.DRBOT macro
description, 2–69
.DRDEF
summary, 1–34
.DRDEF macro
description, 2–70
.DREND
summary, 1–34
.DREND macro
description, 2–75
.DREST
summary, 1–34
.DREST macro
description, 2–77
.DRFIN
summary, 1–35
.DRFIN macro
description, 2–83
.DRINS
summary, 1–35
.DRINS macro
description, 2–84
.DRPTR
summary, 1–35
.DRPTR macro
description, 2–86
.DRSET
summary, 1–35
.DRSET macro
description, 2–88

.DRSPF
summary, 1–35
.DRSPF macro
description, 2–89
.DRTAB
summary, 1–35
.DRTAB macro
description, 2–93
.DRUSE
summary, 1–35
.DRUSE macro
description, 2–95
.DRVTB
summary, 1–35
.DRVTB macro
description, 2–97
.DSTAT
summary, 1–35
.DSTAT programmed request
description, 2–98
EMT 342, 2–98

## E

.ELAW
summary, 1–40
.ELAW programmed request
description, 2–100
EMT 375, Code 36, 2–100
.ELRG
summary, 1–40
.ELRG programmed request
description, 2–101
EMT 375, Code 36, 2–101
.ENTER
summary, 1–35
.ENTER programmed request
description, 2–102
EMT 375, Code 2, 2–102
.EXIT
summary, 1–35
.EXIT programmed request
description, 2–106
EMT 350, 2–106

## F

.FETCH
summary, 1–35
.FETCH programmed request
description, 2–108
.FETCH programmed request (Cont.)
EMT 343, 2–108
File operations
See also .FETCH, .LOAD
file handlers, 1–22
input/output, 1–24
.REOPEN, 1–22
.SAVESTATUS, 1–22
Fixed-offset area
See also .GVAL, .PVAL, .PEEK, .POKE
accessing, 1–4
.FORK
summary, 1–35
.FORK macro
description, 2–110
.FPROT
summary, 1–35
.FPROT programmed request
description, 2–112
EMT 375, CODE 43, 2–112
Functions
extended memory functions, 1–29
list, 1–29

## G

.GCMAP
description, 2–26
summary, 1–40
.GCMAP macro, 2–26
.GFDAT
summary, 1–35
.GFDAT programmed request
description, 2–115
EMT 375, Code 44, 2–115
.GFINF
summary, 1–35
.GFINF programmed request
description, 2–117
EMT 375, Code 44, 2–117
.GFSTA
summary, 1–35
.GFSTA programmed request
description, 2–120
EMT 375, Code 44, 2–120
.GMCX
summary, 1–40
.GMCX programmed request
description, 2–123
EMT 375, Code 36, 2–123

.GTIM
summary, 1–35
.GTIM programmed request
description, 2–124
EMT 375, Code 21, 2–124
.GTJB
summary, 1–35
.GTJB programmed request
description, 2–126
EMT 375, Code 20, 2–126
.GTLIN
summary, 1–36
.GTLIN programmed request
description, 2–129
EMT 345, 2–129
.GVAL
summary, 1–36
.GVAL programmed request
description, 2–132
EMT 375, Code 34, 2–132

## H

.HERR
summary, 1–36
.HERR programmed request
description, 2–136
EMT 374, Code 5, 2–136
.HRESET
summary, 1–36
.HRESET programmed request
See also .SRESET
description, 2–141
EMT 357, 2–141

## I

Input/output operations
asynchronous I/O, 1–24
completion routines, 1–24
event-driven I/O, 1–24
multiterminal requests, 1–26
synchronous I/O, 1–24
terminal input/output, 1–26
.INTEN
summary, 1–36
.INTEN macro
description, 2–142
Interrupt service routines, 1–31

## J

Job communications
See also .RCVD, .MWAIT, .SDAT
sending, receiving, 1–29

## L

.LOCK
summary, 1–36
.LOCK programmed request
description, 2–144
EMT 346, 2–144
.LOOKUP
summary, 1–36
.LOOKUP programmed request
description, 2–147
EMT 375, Code 1, 2–147

## M

.MACS
summary, 1–36
.MAP
summary, 1–40
.MAP programmed request
description, 2–153
EMT 375, Code 36, 2–153
.MFPS
summary, 1–36
.MFPS macro
description, 2–155
Monitor services
list of, 1–2
.MRKT
summary, 1–36
.MRKT programmed request
description, 2–158
EMT 375, Code 22, 2–158
.MSDS
summary, 1–40
.MSDS macro
description, 2–161
.MTATCH
summary, 1–36
.MTATCH programmed request
description, 2–162
EMT 375, Code 37, 2–162
.MTDTCH
summary, 1–36
.MTDTCH programmed request

.MTDTCH programmed request (Cont.)
description, 2–165
EMT 375, Code 37, 2–165
.MTGET
summary, 1–36
.MTGET programmed request
description, 2–167
EMT 375, Code 37, 2–167
.MTIN
summary, 1–36
.MTIN programmed request
description, 2–171
EMT 375, Code 37, 2–171
.MTOUT
summary, 1–36
.MTOUT programmed request
description, 2–173
EMT 375, Code 37, 2–173
.MTPRNT
summary, 1–36
.MTPRNT programmed request
description, 2–175
EMT 375, Code 37, 2–175
.MTPS
summary, 1–36
.MTPS macro
description, 2–155
.MTRCTO
summary, 1–37
.MTRCTO programmed request
description, 2–177
EMT 375, Code 37, 2–177
.MTSET
summary, 1–37
.MTSET programmed request
description, 2–178
EMT 375, Code 37, 2–178
.MTSTAT
summary, 1–37
.MTSTAT programmed request
description, 2–180
EMT 375, Code 37, 2–180
.MTWAIT
summary, 1–40
Multiterminal operation, 1–4
.MWAIT programmed request
description, 2–181
EMT 375, Code 37, 2–181

## P

.PEEK
summary, 1–37
.PEEK programmed request
See also .POKE
description, 2–183
EMT 375, Code 34, 2–183
.POKE
summary, 1–37
.POKE programmed request
See also .PEEK
description, 2–185
EMT 375, Code 34, 2–185
.PRINT
summary, 1–37
.PRINT programmed request
description, 2–187
EMT 351, 2–187
Programmed requests
command interpretation, 1–21
devices, 1–20
EMT instructions, 1–5
error processing, 1–20
extended memory functions, 1–29
for all environments, 1–33
implementation, 1–5
input/output access, 1–19
mapped environments, 1–40
memory allocation, 1–18
multijob environments, 1–40
summary, 1–1
types, 1–29
Program termination
See also .SRESET, .HRESET
job reset, 1–28
suspension, 1–28
.PROTECT
summary, 1–37
.PROTECT programmed request
See also .UNPROTECT
description, 2–188
EMT 375, Code 31, 2–188
.PUNROTECT programmed request
See also .PROTECT
.PURGE
summary, 1–37
.PURGE programmed request
description, 2–191

.PURGE programmed request (Cont.)
EMT 374, Code 3, 2–191
.PVAL
summary, 1–37
.PVAL programmed request
description, 2–132
EMT 375, Code 34, 2–132

## Q

.QELDF
summary, 1–37
.QELDF macro
description, 2–193
.QSET
summary, 1–37
.QSET programmed request
description, 2–194
EMT 353, 2–194

## R

.RCTRLO
summary, 1–37
.RCTRLO programmed request
description, 2–196
EMT 355, 2–196
.RCVD
summary, 1–40
.RCVDC
summary, 1–40
.RCVDC programmed request
See also .SDATC
description, 2–197
EMT 375, Code 26, 2–197
.RCVD programmed request
See also .SDAT
description, 2–197
EMT 375, Code 26, 2–197
.RCVDW
summary, 1–40
.RCVDW programmed request
See also .SDATW
description, 2–197
EMT 375, Code 26, 2–197
.RDBBK
summary, 1–40
.RDBBK macro
See also .RDBDF
description, 2–205
.RDBDF
summary, 1–40
.RDBDF macro
See also .RDBBK
description, 2–206
.READ
summary, 1–37
.READC
summary, 1–37
.READC programmed request
See also .SDATC
description, 2–207
EMT 375, Code 10, 2–207
.READ programmed request
See also .SDAT
description, 2–207
EMT 375, Code 10, 2–207
.READW
summary, 1–37
.READW programmed request
See also .SDATW
description, 2–207
EMT 375, Code 10, 2–207
Regions
mapped, 1–29
unmapped, 1–29
.RELEASE
summary, 1–37
.RENAME
summary, 1–37
.RENAME programmed request
description, 2–219
.REOPEN
summary, 1–37
.REOPEN programmed request
See also .SAVESTATUS
description, 2–221
EMT 375, Code 6, 2–221
Request types
.CMAP, .GCMAP
.CRAW, .CRRG
.ELAW, .ELRG, 1–29
general mapping control, 1–29
map, 1–29
.MAP, .UNMAP, 1–29
region, 1–29
status, 1–29
window, 1–29
.RSUM

.RSUM (Cont.)
summary, 1–37
.RSUM programmed request
description, 2–261
EMT 374, Code 2, 2–261
RT–11 Monitors
See also FB monitor
See also SB monitor
See also XB and ZB monitors
See also XM and XB monitors
multi-job mapped monitors, 1–3
multi-job unmapped monitors, 1–3
operational configurations, 1–2
single-job mapped monitors, 1–3
single-job unmapped monitors, 1–2

## S

.SAVESTATUS
summary, 1–38
.SAVESTATUS programmed request
See also .REOPEN
description, 2–223
EMT 375, Code 5, 2–223
.SCCA
summary, 1–38
.SCCA programmed request
description, 2–226
EMT 375, Code 35, 2–226
.SDAT[3]
summary, 1–40
.SDATC[3]
summary, 1–40
.SDATC programmed request
See also .RCVDC
description, 2–228
EMT 375, Code 25, 2–228
.SDAT programmed request
See also .RCVD
description, 2–228
EMT 375, Code 25, 2–228
.SDATW[3]
summary, 1–40
.SDATW programmed request
See also .RCVDW
description, 2–228
EMT 375, Code 25, 2–228
.SDTTM
summary, 1–38
.SDTTM programmed request
description, 2–236
EMT 375, Code 40, 2–236
.SERR
summary, 1–38
.SERR programmed request
description, 2–136
EMT 374, Code 4, 2–136
table of error codes, 2–138
.SETTOP
summary, 1–38
.SETTOP programmed request
description, 2–239
EMT 354, 2–239
.SFDAT
summary, 1–38
.SFDAT programmed request
description, 2–242
EMT 375, Code 42, 2–242
.SFINF
summary, 1–38
.SFINF programmed request
description, 2–243
EMT 375, Code 44, 2–243
.SFPA
summary, 1–38
.SFPA programmed request
description, 2–246
EMT 375, Code 30, 2–246
.SFSTA
summary, 1–38
.SFSTAT programmed request
description, 2–248
EMT 375, Code 44, 2–248
SOB
summary, 1–38
SOB macro
description, 2–253
.SPCPS
summary, 1–38
.SPCPS programmed request
description, 2–254
EMT 375, Code 41, 2–254
.SPFUN
summary, 1–38
.SPFUN programmed request
description, 2–257
EMT 375, Code 32, 2–257
.SPND

.SPND (Cont.)
summary, 1–38
.SPND programmed request
description, 2–261
EMT 374, Code 1, 2–261
.SRESET
summary, 1–38
.SRESET programmed request
description, 2–263
EMT 352, 2–263
.SYNCH
summary, 1–38
.SYNCH macro
description, 2–264
System communication areas, 1–4
System control
path flow, 1–6
System Conventions
addressing modes, 1–12
blank arguments, 1–11
Channels and channel numbers, 1–14
device blocks, 1–14
keyword macro arguments, 1–13
programmed request errors, 1–15
Program request format, 1–8
user service routine requirements, 1–15
System information
examining and reporting status, 1–21
System job environment
program requests in, 1–4

## T

Timer support
See also .MRKT
selecting, 1–27
.TIMIO
summary, 1–38
.TIMIO macro
description, 2–267
.TLOCK
summary, 1–38
.TLOCK programmed request
description, 2–269
EMT 374, Code 7, 2–269
.TRPSET
summary, 1–38
.TRPSET programmed request
description, 2–271
EMT 375, Code 3, 2–271
.TTINR
summary, 1–38
.TTINR programmed request
description, 2–273
EMT 340, 2–273
.TTOUTR
summary, 1–38
.TTOUTR programmed request
description, 2–275
EMT 341, 2–275
.TTYIN
summary, 1–38
.TTYIN programmed request
description, 2–273
EMT 340, 2–273
.TTYOUT
summary, 1–38
.TTYOUT programmed request
description, 2–275
EMT 341, 2–275
.TWAIT
summary, 1–38
.TWAIT programmed request
See also .QSET
description, 2–278
EMT 375, Code 24, 2–278

## U

.UNLOCK
summary, 1–39
.UNLOCK programmed request
description, 2–145
EMT 347, 2–145
.UNMAP
summary, 1–40
.UNMAP programmed request
description, 2–153
EMT 375, Code 36, 2–153
.UNPROTECT
summary, 1–39
.UNPROTECT programmed request
description, 2–190
EMT 375, Code 31, 2–190
Using programmed requests, 1–18
initialization and control, 1–18
USR
background job, 1–15
monitor offset 374, 1–15
protecting program areas, 1–15

USR (Cont.)
swapping, 1–15
value of location 46, 1–15

## V

..V1..
summary, 1–39
..V2..
summary, 1–39

## W

.WAIT
summary, 1–39
.WAIT programmed request
description, 2–281
EMT 374, Code 0, 2–281
See also .READ/.WRITE, 2–281
.WDBBK
summary, 1–40
.WDBBK macro
See also .WDBDF
description, 2–282
.WDBDF
summary, 1–40
.WDBDF macro
See also .WDBBK
description, 2–283
.WRITC
summary, 1–39
.WRITC programmed request
description, 2–284
EMT 375, Code 11, 2–284
.WRITE
summary, 1–39
.WRITE programmed request
description, 2–284
EMT 375, Code 11, 2–284
.WRITW
summary, 1–39
.WRITW programmed request
description, 2–284
EMT 375, Code 11, 2–284