RT–11 Programmer's Reference Manual

AA–H378B–TC

March 1983

This manual is a reference document for advanced RT–11 users, including FORTRAN IV and MACRO–11 assembly language programmers. This manual supersedes the *RT–11 Programmer's Reference Manual*, Order No. AA–5378A–TC.

Operating System: RT–11 Version 5.0

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

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

digital equipment corporation · maynard, massachusetts

First Printing, March 1980
Updated, March 1981
Revised, March 1983

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

© Digital Equipment Corporation 1980, 1981, 1983.
All Rights Reserved.

Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:

	digital™	
DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem–10	P/OS	VMS
DECSYSTEM–20	Professional	VT
DECUS	Rainbow	Work Processor
DECwriter	RSTS	
DIBOL	RSX	

Contents

Page

Preface xi

Chapter 1 Introduction to Advanced RT–11 Programming

1.1 Programmed Requests 1–1
1.1.1 Programmed Request Implementation 1–3
1.1.1.1 EMT Instructions 1–3
1.1.1.2 System Control Path Flow 1–4
1.1.2 System Conventions 1–6
1.1.2.1 Programmed Request Format 1–6
1.1.2.2 Blank Arguments 1–9
1.1.2.3 Addressing Modes 1–10
1.1.2.4 Keyword Macro Arguments 1–11
1.1.2.5 Channels and Channel Numbers 1–11
1.1.2.6 Device Blocks 1–12
1.1.2.7 Programmed Request Errors 1–12
1.1.2.8 User Service Routine (USR) Requirement 1–13
1.1.3 Using Programmed Requests 1–15
1.1.3.1 Initialization and Control 1–15
1.1.3.2 Examining System Information and Reporting Status 1–18
1.1.3.3 Command Interpretation 1–18
1.1.3.4 File Operations 1–19
1.1.3.5 Input/Output Operations 1–20
1.1.3.6 Foreground/Background Communications 1–23
1.1.3.7 Timer Support 1–24
1.1.3.8 Program Termination or Suspension 1–24
1.1.3.9 System Job Communications 1–25
1.1.3.10 Extended Memory Functions 1–25
1.1.3.11 Interrupt Service Routines 1–26
1.1.3.12 Device Handlers 1–27
1.1.4 Compatibility with Previous RT–11 Versions 1–27
1.1.4.1 Version 1 Programmed Requests 1–27
1.1.4.2 Version 2 Programmed Requests 1–28
1.1.4.3 Version 3 Programmed Requests 1–28
1.1.4.4 Version 4 Programmed Requests 1–29
1.1.4.5 Version 5 Programmed Requests 1–29
1.1.5 Programmed Request Conversion 1–29
1.1.5.1 Macro Calls Not Requiring Conversion 1–29
1.1.5.2 Macro Calls That Can Be Converted 1–30
1.1.6 Programmed Request Summary 1–32
1.2 Using the System Subroutine Library 1–37
1.2.1 System Conventions 1–38
1.2.1.1 Channel Numbers 1–39
1.2.1.2 Completion Routines 1–39

1.2.1.3 Device Blocks 1–41
1.2.1.4 INTEGER*4 Support Functions. 1–41
1.2.1.5 User Service Routine (USR) Requirements 1–42
1.2.1.6 Subroutines Requiring Additional Queue Elements . 1–45
1.2.1.7 System Restriction 1–45
1.2.2 Calling SYSLIB Subroutines 1–46
1.2.3 FORTRAN/MACRO Interface 1–47
1.2.3.1 Subroutine Register Usage 1–48
1.2.3.2 FORTRAN Programs Calling MACRO Subroutines . 1–48
1.2.3.3 MACRO Routines Calling FORTRAN Programs . . 1–50
1.2.4 FORTRAN Programs in a Foreground/Background Environment. 1–52
1.2.4.1 Calculating Workspace for a FORTRAN Foreground Program 1–53
1.2.4.2 Running a FORTRAN Program in a Foreground/Background Environment 1–54
1.2.5 Linking with FORLIB 1–55
1.2.6 SYSLIB Services Not Provided by Programmed Requests . . . 1–56
1.2.6.1 Time Conversion and Date Access 1–56
1.2.6.2 Program Suspension 1–56
1.2.6.3 Two-Word Integer Support (INTEGER*4) 1–56
1.2.6.4 Radix–50 Conversion 1–57
1.2.6.5 Character String Operations 1–57
1.2.7 Character String Functions 1–57
1.2.7.1 Allocating Character String Variables 1–59
1.2.7.2 Passing Strings to Subprograms 1–60
1.2.7.3 Using Quoted-String Literals 1–61
1.2.8 System Subroutine Summary 1–61

Chapter 2 Programmed Request Description and Examples

2.1 .ABTIO 2–2
2.2 .CDFN. 2–3
2.3 .CHAIN 2–4
2.4 .CHCOPY 2–6
2.5 .CLOSE 2–8
2.6 .CMKT 2–9
2.7 .CNTXSW 2–10
2.8 .CRAW 2–12
2.9 .CRRG. 2–15
2.10 .CSIGEN 2–16
2.10.1 Passing Option Information 2–19
2.11 .CSISPC 2–21
2.12 .CSTAT 2–24
2.13 .CTIMIO. 2–25
2.14 .DATE. 2–26
2.15 .DELETE 2–27
2.16 .DEVICE 2–28
2.17 .DRAST 2–30

2.18 .DRBEG 2–32
2.19 .DRBOT 2–32
2.20 .DRDEF 2–33
2.21 .DREND 2–34
2.22 .DRFIN 2–35
2.23 .DRSET 2–35
2.24 .DRVTB 2–36
2.25 .DSTATUS 2–36
2.26 .ELAW 2–39
2.27 .ELRG 2–40
2.28 .ENTER 2–40
2.29 .EXIT 2–43
2.30 .FETCH/.RELEAS 2–45
2.31 .FORK 2–47
2.32 .FPROT 2–49
2.33 .GMCX 2–50
2.34 .GTIM 2–51
2.35 .GTJB 2–53
2.36 .GTLIN 2–55
2.37 .GVAL/.PVAL 2–57
2.38 .HERR/.SERR 2–58
2.39 .HRESET 2–61
2.40 .INTEN 2–61
2.41 .LOCK/.UNLOCK 2–62
2.42 .LOOKUP 2–65
2.42.1 Standard Lookup 2–65
2.42.2 System Job Lookup 2–67
2.43 .MAP 2–68
2.44 .MFPS/.MTPS 2–69
2.45 .MRKT 2–71
2.46 .MTATCH 2–73
2.47 .MTDTCH 2–75
2.48 .MTGET 2–76
2.49 .MTIN 2–79
2.50 .MTOUT 2–80
2.51 .MTPRNT 2–81
2.52 .MTPS 2–82
2.53 .MTRCTO 2–82
2.54 .MTSET 2–83
2.55 .MTSTAT 2–84
2.56 .MWAIT 2–85
2.57 .PEEK/.POKE 2–86
2.58 .POKE 2–87
2.59 .PRINT 2–87
2.60 .PROTECT/.UNPROTECT 2–88
2.61 .PURGE 2–90
2.62 .PVAL 2–91
2.63 .QELDF 2–92
2.64 .QSET 2–92
2.65 .RCTRLO 2–95
2.66 .RCVD/.RCVDC/.RCVDW 2–95
2.67 .RDBBK 2–99
2.68 .RDBDF 2–99

2.69 .READ/.READC/.READW 2–100
2.70 .RELEAS 2–107
2.71 .RENAME 2–107
2.72 .REOPEN 2–109
2.73 .RSUM 2–109
2.74 .SAVESTATUS 2–110
2.75 .SCCA 2–112
2.76 .SDAT/.SDATC/.SDATW 2–113
2.77 .SDTTM 2–117
2.78 .SERR 2–119
2.79 .SETTOP 2–119
2.79.1 .SETTOP in an Extended Memory Environment 2–121
2.80 .SFDAT 2–122
2.81 .SFPA 2–123
2.82 .SPCPS 2–124
2.83 .SPFUN 2–126
2.84 .SPND/.RSUM 2–130
2.85 .SRESET 2–131
2.86 .SYNCH 2–132
2.87 .TIMIO 2–135
2.88 .TLOCK 2–136
2.89 .TRPSET 2–137
2.90 .TTYIN/.TTINR 2–139
2.91 .TTYOUT/.TTOUTR 2–141
2.92 .TWAIT 2–143
2.93 .UNLOCK 2–144
2.94 .UNMAP 2–145
2.95 .UNPROTECT 2–145
2.96 .WAIT 2–145
2.97 .WDBBK 2–147
2.98 .WDBDF 2–148
2.99 .WRITE/.WRITC/.WRITW 2–148

Chapter 3 System Subroutine Description and Examples

3.1 AJFLT 3–1
3.2 CHAIN 3–1
3.3 CLOSEC/ICLOSE 3–2
3.4 CONCAT 3–4
3.5 CVTTIM 3–5
3.6 DEVICE 3–5
3.7 DJFLT 3–6
3.8 GETSTR 3–7
3.9 GTIM 3–7
3.10 GTJB/IGTJB 3–8
3.11 GTLIN 3–9
3.12 IABTIO 3–10
3.13 IADDR 3–10
3.14 IAJFLT 3–11
3.15 IASIGN 3–11
3.16 ICDFN 3–13
3.17 ICHCPY 3–14

3.18 ICLOSE 3–15
3.19 ICMKT 3–15
3.20 ICSI 3–16
3.21 ICSTAT 3–18
3.22 IDELET 3–18
3.23 IDJFLT 3–20
3.24 IDSTAT 3–20
3.25 IENTER 3–21
3.26 IFETCH 3–23
3.27 IFPROT 3–23
3.28 IFREEC 3–24
3.29 IGETC 3–24
3.30 IGETSP 3–25
3.31 IGTJB 3–26
3.32 IJCVT 3–26
3.33 ILUN 3–26
3.34 INDEX 3–27
3.35 INSERT 3–27
3.36 INTSET 3–28
3.37 IPEEK 3–30
3.38 IPEEKB 3–30
3.39 IPOKE 3–31
3.40 IPOKEB 3–31
3.41 IPUT 3–32
3.42 IQSET 3–32
3.43 IRAD50 3–33
3.44 IRCVD/IRCVDC/IRCVDF/IRCVDW 3–33
3.45 IREAD/IREADC/IREADF/IREADW 3–36
3.46 IRENAM 3–40
3.47 IREOPN 3–41
3.48 ISAVES 3–42
3.49 ISCHED 3–43
3.50 ISCOMP 3–44
3.51 ISDAT/ISDATC/ISDATF/ISDATW 3–44
3.52 ISDTTM 3–47
3.53 ISFDAT 3–47
3.54 ISLEEP 3–48
3.55 ISPFN/ISPFNC/ISPFNF/ISPFNW 3–49
3.56 ISPY 3–55
3.57 ITIMER 3–55
3.58 ITLOCK 3–57
3.59 ITTINR 3–57
3.60 ITTOUR 3–59
3.61 ITWAIT 3–59
3.62 IUNTIL 3–60
3.63 IVERIF 3–61
3.64 IWAIT 3–61
3.65 IWRITE/IWRITC/IWRITF/IWRITW 3–61
3.66 JADD 3–64
3.67 JAFIX 3–65
3.68 JCMP 3–65
3.69 JDFIX 3–66
3.70 JDIV 3–66
3.71 JICVT 3–67

3.72 JJCVT 3–68
3.73 JMOV 3–68
3.74 JMUL 3–69
3.75 JSUB 3–69
3.76 JTIME 3–70
3.77 LEN 3–70
3.78 LOCK 3–71
3.79 LOOKUP 3–72
3.80 MRKT 3–75
3.81 MTATCH 3–76
3.82 MTDTCH 3–78
3.83 MTGET 3–79
3.84 MTIN 3–79
3.85 MTOUT 3–80
3.86 MTPRNT 3–81
3.87 MTRCTO 3–81
3.88 MTSET 3–81
3.89 MTSTAT 3–82
3.90 MWAIT 3–83
3.91 PRINT 3–83
3.92 PURGE 3–84
3.93 PUTSTR 3–84
3.94 R50ASC 3–85
3.95 RAD50 3–85
3.96 RCHAIN 3–86
3.97 RCTRLO 3–86
3.98 REPEAT 3–87
3.99 RESUME 3–87
3.100 SCCA 3–88
3.101 SCOMP/ISCOMP 3–89
3.102 SCOPY 3–89
3.103 SECNDS 3–90
3.104 SETCMD 3–91
3.105 STRPAD 3–91
3.106 SUBSTR 3–92
3.107 SUSPND 3–93
3.108 TIMASC 3–94
3.109 TIME 3–95
3.110 TRANSL 3–95
3.111 TRIM 3–97
3.112 UNLOCK 3–97
3.113 VERIFY 3–98

Appendix A Display File Handler

A.1 Description A–1
A.1.1 Assembly Language Display Support A–1
A.1.2 Monitor Display Support A–2
A.2 Description of Graphics Macros A–3
A.2.1 .BLANK A–3
A.2.2 .CLEAR A–4
A.2.3 .INSRT A–5

A.2.4 .LNKRT A–5
A.2.5 .LPEN A–7
A.2.6 .NAME A–8
A.2.7 .REMOV A–9
A.2.8 .RESTR A–9
A.2.9 .SCROL A–9
A.2.10 .START A–10
A.2.11 .STAT A–10
A.2.12 .STOP A–11
A.2.13 .SYNC/.NOSYN A–11
A.2.14 .TRACK A–11
A.2.15 .UNLNK A–12

A.3 Extended Display Instructions A–13

A.3.1 DJSR Subroutine Call Instruction A–13
A.3.2 DRET Subroutine Return Instruction A–13
A.3.3 DSTAT Display Status Instruction A–14
A.3.4 DHALT Display Halt Instruction A–14
A.3.5 DNAME Load Name Register Instruction A–14

A.4 Using the Display File Handler A–15

A.4.1 Assembling Graphics Programs A–16
A.4.2 Linking Graphics Programs A–16

A.5 Display File Structure A–17

A.5.1 Subroutine Calls A–18
A.5.2 Main File/Subroutine Structure A–19
A.5.3 BASIC–11 Graphic Software Subroutine Structure A–20

A.6 Summary of Graphics MACRO Calls A–21
A.7 Display Processor Mnemonics A–23
A.8 Assembly Instructions A–24

A.8.1 General Instructions A–24
A.8.2 VTBASE A–24
A.8.3 VTCAL1 – VTCAL4 . . . . . A–24
A.8.4 VTHDLR A–24
A.8.5 Building VTLIB.OBJ A–24

A.9 VTMAC A–25
A.10 Examples Using GTON A–27

Appendix B System Macro Library

Index

Figures

1–1 System Flow During Programmed Request Execution 1–5
1–2 EMT 374 Argument 1–7
1–3 Stack Set by .CSIGEN Programmed Request 1–7

1–4 EMT 375 Argument Block 1–8
1–5 Subroutine Argument Block 1–47
1–6 Argument Block for Program INIARR 1–49

Tables

1–1 EMT Codes 1–4
1–2 Programmed Requests Requiring the USR 1–13
1–3 Programmed Request Conversions (Version 1 to Version 2) 1–31
1–4 Programmed Requests for All RT–11 Environments 1–32
1–5 Foreground/Background and Extended Memory Programmed Requests 1–36
1–6 FORTRAN Program PSECT Ordering 1–44
1–7 Return Value Conventions for Function Subroutines 1–48
1–8 SYSLIB Conversion Calls 1–57
1–9 Summary of SYSLIB Subroutines 1–61
2–1 Timer Block Format 2–25
2–2 Soft Error Codes (.SERR) 2–59
3–1 Functions and Function Codes (Octal) 3–50
A–1 Description of Display Status Words A–8

Preface

The *RT–11 Programmer's Reference Manual* describes the programmed requests and subroutines that are available in the system macro library (SYSMAC.SML) and system subroutine library (SYSLIB.OBJ). It provides programming examples that show how to use programmed requests and subroutines.

Chapter 1, Introduction to Advanced RT–11 Programming, describes the implementation and use of the programmed requests and the FORTRAN-callable subroutines.

Chapter 2, Programmed Request Description and Examples, describes the individual programmed requests in detail. Program examples are included for each request. In addition, macros and subroutines that are used in implementing device handlers and interrupt service routines are described.

Chapter 3, System Subroutine Description and Examples, describes in detail all the RT–11 FORTRAN-callable subroutines. This chapter also contains examples of the use of the calls to the system subroutine library.

Appendix A, Display File Handler, describes the graphics support for the RT–11 operating system. Program examples are included to help you develop your own display program.

Appendix B, System Macro Library, is a listing of the RT–11 System Macro Library (SYSMAC.SML).

This manual is written for an advanced-level user. If you have no RT–11 experience, or very little, read:

Introduction to RT–11
RT–11 System User's Guide
RT–11 System Utilities Manual
PDP–11 MACRO–11 Language Reference Manual

In addition, FORTRAN programmers should read:

RT–11/RSTS/E FORTRAN IV User's Guide
PDP–11/FORTRAN Language Reference Manual

If you are interested in additional programming techniques or other system programming topics, read the *RT–11 Software Support Manual.*

Chapter 1
Introduction to Advanced RT–11 Programming

Programmed requests and system subroutines are available as part of the RT–11 Operating System and can aid you in writing reliable and efficient programs.

Programmed requests provide a number of services to application programs. The requests function as calls to routines in the RT–11 monitor that perform these services. As system macros, they are defined in a system macro library that is stored on the system volume and named SYSMAC.SML. In addition, macro routines are available in the system macro library that can help you write device handlers and interrupt service routines.

If you are a FORTRAN programmer, you can access the RT–11 monitor services through calls to the system subroutine library called SYSLIB.OBJ, which is stored on the system volume. A character string manipulation package and two-word integer support routines are included in this library. The SYSLIB subroutines allow you to write almost all application programs in FORTRAN without having to do any assembly language coding.

This chapter tells you how to use programmed requests and subroutines effectively in your programs. Examples are provided to demonstrate their flexibility and value in working programs.

1.1 Programmed Requests

You issue a programmed request in your source program when a certain monitor service is required. The programmed request expands into the appropriate machine language code during assembly time. During program execution, this code requests the resident monitor to supply the service represented by the programmed request.

The services involve the following processes:

1. Initialization and control of operating system characteristics
2. Allocating system resources and reporting status
3. Command interpretation
4. File operations
5. Input/output operations
6. Foreground/background communications
7. Timer support
8. Program termination or suspension
9. System job communications
10. Extended memory functions

The system macro library (SYSMAC.SML) also contains several macros that are not programmed requests. These macros are provided to aid you in writing:

1. Interrupt service routines
2. Device handlers
3. Memory management control blocks

They are described in Chapter 2 along with the programmed requests.

Components of the RT–11 Operating System that support programmed requests are as follows:

1. Single-Job Monitor
   The single-job (SJ) monitor supports most of the programmed requests. Table 1–4 lists the programmed requests that are supported by the SJ monitor. These programmed requests can manipulate files, perform input and output, set timer routines, check system resources and status, and terminate program operations.
2. Foreground/Background Monitor
   Some programmed requests are provided for the foreground/background (FB) monitor only. Table 1–5 lists the programmed requests that are supported by the FB monitor in addition to those listed in Table 1–4. These programmed requests allow a program to set timer routines, suspend and resume jobs, and send messages and data between foreground and background jobs.
3. Extended Memory Monitor
   The extended memory (XM) monitor provides additional programmed requests and features above those found in the FB monitor. The XM monitor extends the memory support capability of RT–11 beyond the 28K-word (plus I/O page) restriction imposed by the 16-bit address size. The XM monitor's programmed requests extend a program's effective logical addressing space (see Table 1–5).
4. Multiterminal Feature
   The multiterminal feature of RT–11 allows your program to perform character input/output on up to 16 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.
5. System Job Support
   System job support allows users in a foreground/background or extended memory environment to extend the present standard foreground/background system to include up to eight jobs. Two system jobs are presently provided with the RT–11 system: the error logger and the device queue program. Programmed requests allow programs to

copy channels from other jobs, obtain job status information about jobs in the system, and send messages and data between any jobs in the system. The *RT–11 Software Support Manual* describes the system job feature in more detail.

Programmed requests perform most system resource control and interrogation functions. However, some communication is accomplished by directly accessing two memory areas: the system communication area and the monitor fixed-offset area.

The system communication area resides in locations 40 to 57(octal) and contains parameters that describe and control execution of the current job. This area holds information such as the Job Status Word, starting address of the job, User Service Routine (USR) swapping address, and the address of the start of the resident monitor. Some of this information is supplied by your program, while other data is supplied by the monitor and may not be changed.

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 used to control monitor operation. Your program can examine or modify these values to determine the condition of the operating environment while a job is running. The *RT–11 Software Support Manual* describes in detail both the system communication area and the fixed-offset area.

This manual specifically describes programmed requests for RT–11 Version 5. Programmed requests for earlier versions of RT–11 and guidelines for their conversion are treated in Sections 1.1.4 and 1.1.5.

1.1.1 Programmed Request Implementation

1.1.1.1 EMT Instructions — A programmed request is a macro call followed by the necessary number of arguments. The macro code that corresponds to the macro call of a programmed request is expanded by the MACRO assembler when the programmed request appears in your program. The expansion arranges the arguments of the programmed request for the monitor and generates an EMT (emulator trap) instruction.

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. This vector location is loaded at boot time with an address pointing to a location in the monitor. The monitor location contains the EMT processing code that services the interrupt caused by the EMT instruction.

Table 1–1 shows the codes that may appear in the low-order byte of an EMT instruction and the interpretation of these codes by the monitor.

Table 1–1: EMT Codes

Low-Order Byte	Interpretation
377	Reserved; RT–11 ignores this EMT and returns control to the user program immediately.
376	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.
360–373	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	Version 1 programmed requests with arguments both on the stack and in R0. They are supported for binary compatibility with Version 1 programs.

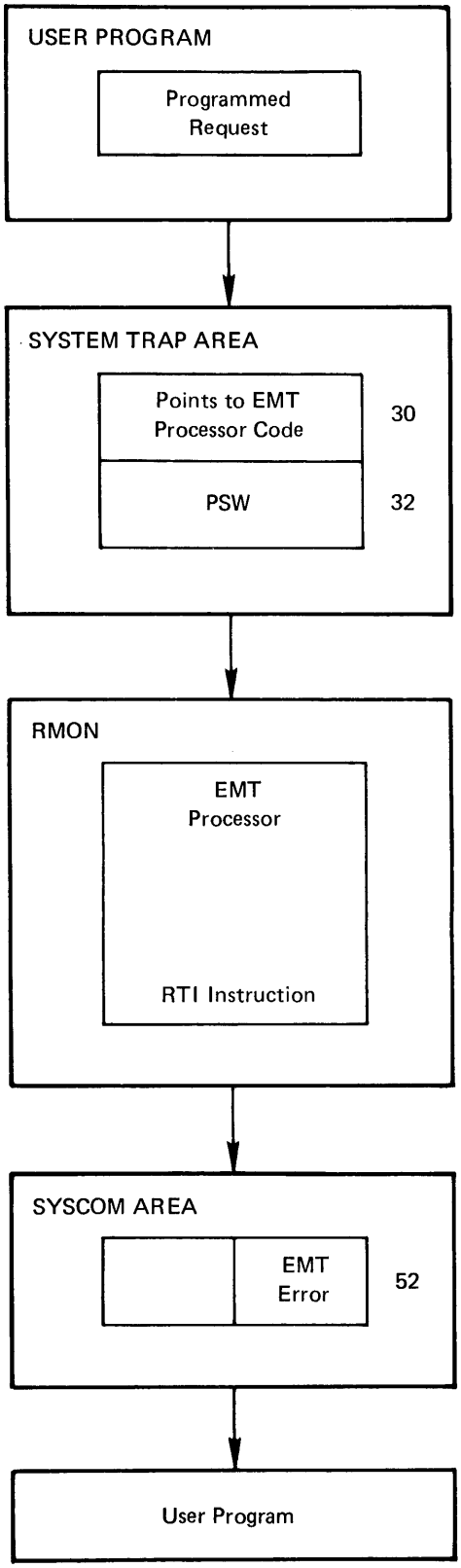
EMT instructions should never appear in your programs except through programmed requests.

1.1.1.2 System Control Path Flow — Figure 1–1 shows system flow when a programmed request is executed.

In Figure 1–1, a programmed request in an application (or system utility) program is implemented with an EMT instruction. When your program is executed, the EMT instruction transfers control to the EMT processor code in the monitor. The user program counter (PC) and processor status word (PSW) are pushed onto the stack, and the contents of location 30 are placed in the program counter. Location 30 points to the EMT processor code in the monitor. Location 32 contains the PSW for the EMT trap. Byte 52 of the system communication area is loaded with an error code by the monitor if the monitor detects any errors during the EMT processing. In addition, the EMT processor uses R0 to pass back information to the program. All other registers except SP are preserved; .CSIGEN and .CSISPC return information on the stack.

The monitor either processes a programmed request entirely when it is issued or it performs partial processing and queues the request for further processing. The requests that require queued processing support completion routines (see Section 1.1.3.5). If a request results in an error prior to 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 1–1: System Flow During Programmed Request Execution



1.1.2 System Conventions

This section describes the system conventions that must be followed for the correct operation of programmed requests.

1.1.2.1 Programmed 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. The macro names for all programmed requests start with a period (.) to distinguish them from symbols and macros you define.

Arguments supplied to a programmed request must be valid assembler expressions since the arguments are used as source fields in the instructions (such as a MOV instruction) when the macros are expanded at assembly time. All programmed requests that appear in your program must appear in a .MCALL directive to make the macro definition available from the system macro library, SYSMAC.SML. If the programmed request is specified by a .MCALL directive, the programmed request is obtained from the system macro library at assembly time. Alternatively, you can enable the automatic .MCALL feature of MACRO by using the .ENABL MCL directive.

Programmed requests have two formats that are accepted by the monitor. The first format specifies the programmed request followed by the arguments required by the request. The second format specifies the programmed request, the address of the argument block, and the arguments that will be contained in the argument block. Because the way you can set up the argument block and specify arguments to a programmed request can vary, read the sections on programmed request format and on blank arguments to be sure of correct programmed request operation.

FORMAT 1
The first format for programmed requests is as follows:

```
.PRGREQ Arg1,Arg2,...,Argn
```

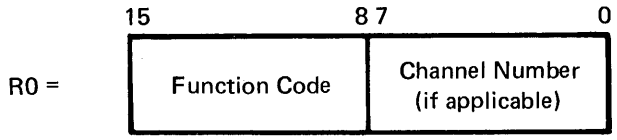
where:

.PRGREQ	is the name of the programmed request
Arg1,Arg2,...,Argn	are the 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 1–2.

Figure 1–2: EMT 374 Argument



One programmed request that generates an EMT 374 is .DATE. The macro for this programmed request appears in the system macro library as:

```
.MACRO .DATE
       MOV #10,#^O400,%0
       EMT ^O374
.ENDM
```

The function code, which in this case is 10(decimal) is placed in the high-order byte of R0. A channel code of 0 is placed in the low-order byte.

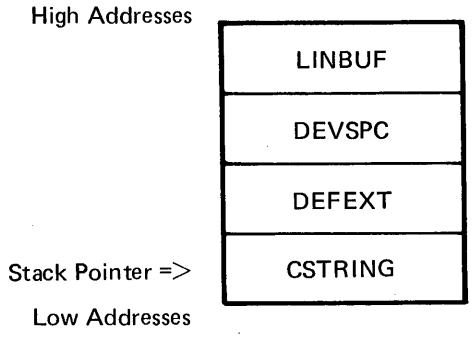
For EMT 340 through 357, if there are arguments, they are placed either on the stack, in R0, or in R0 and on the stack.

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

```
.MACRO .CSIGEN DEVSPC,DEFEXT,CSTRNG,LINBUF
.IIF NB <LINBUF>          MOV     LINBUF,-(6,)
        MOV     DEVSPC,-(6,)
.IIF NB <LINBUF>          INC     (6,)
        MOV     DEFEXT,-(6,)
.IF B CSTRNG
        CLR     -(6,)
.IFF
.IF IDN CSTRNG,#0
        CLR     -(6,)
.IFF
        MOV     CSTRNG,-(6,)
.ENDC
.ENDC
        EMT     ^O344
.ENDM
```

When this programmed request is executed, all the specified arguments are placed on the user stack. Thus, the user stack would appear as shown in Figure 1–3.

Figure 1–3: Stack Set by .CSIGEN Programmed Request



The EMT processor then uses these arguments in performing the function of the programmed request .CSIGEN.

FORMAT 2
The second format for programmed requests is as follows:

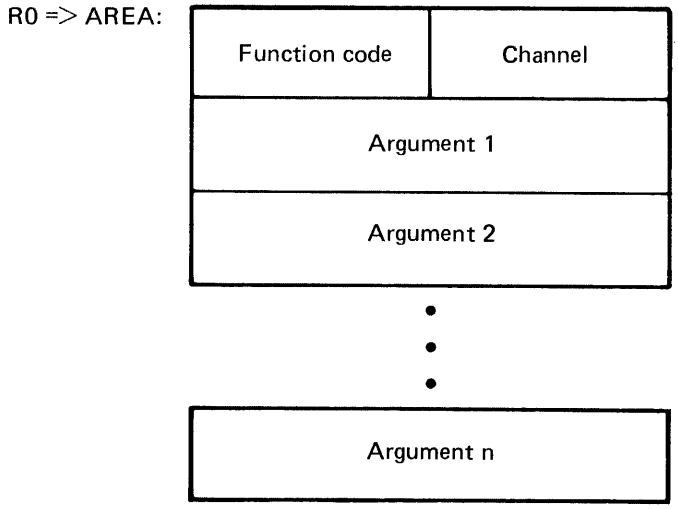
```
.PRGREQ Area,Arg1,Arg2,...,Argn
```

where:

.PRGREQ	is the name of the programmed request
Area	is the address of an argument block
Arg1,Arg2,...,Argn	are the 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. In general, the argument block appears as shown in Figure 1–4.

Figure 1–4: EMT 375 Argument Block



The programmed request format uses Area as a pointer to the argument block that contains the arguments Arg1 through Argn.

```
.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. Thus,

```
.PRGREQ Area,Arg1,Arg2
```

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

```
.PRGREQ Area
```

points R0 to the block and fills in the first word — that is, the function code and channel number — without filling in any other arguments. Arguments that are left blank are discussed in the following section.

1.1.2.2 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, in address R0 plus 2), while Arg2 is in R0 plus 4.

There are two ways to account for arguments. You can let the MACRO assembler generate the instructions needed to fill up the argument block at run time, or you can 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 MACRO generate the instructions, both for program clarity and for reduced chance of programming error.

The following examples are all equivalent in that the arguments have been accounted for either in the program instructions or in the programmed request.

```
MOV #ARG1,AREA+2
MOV #ARG2,AREA+4

.PRGREQ #AREA
```

is equivalent to

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

and also to

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

.PRGREQ
```

This last example sets up all the arguments for the programmed request prior to executing the programmed request.

The following example shows how arguments are specified to the .TWAIT programmed request.

```
            .TITLE      EXWAIT.MAC
            .MCALL      .PRINT,.TWAIT
START:
WAIT:       .TWAIT      #EMTLST
            .PRINT      #MSG
            BR          WAIT
EMTLST:     .BYTE       0,24
            .WORD       TIME
TIME:       .WORD       0,10.*60
MSG:        .ASCIZ      /PRINT THIS EVERY TEN SECONDS/
            .END        START
```

The .TWAIT programmed request suspends a program and requires two arguments. The first argument is area, which points to the address of a two-word EMT argument block; the second argument is Time, which is a pointer to two words of time (high-order first, low-order second) expressed in ticks. In the example shown above, EMTLST 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 symbolic 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.

1.1.2.3 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 [#]. 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. For example, when a direct numerical argument is required, the immediate mode causes the correct value to be put into the argument block. Thus

.PRGREQ #Area,#4

is correct, while

.PRGREQ #Area,4

is not correct since the contents of location 4 are placed into the argument block instead of the desired value 4.

However, the form

```
        .PRGREQ LIST,NUMBER
                .
                .
                .

LIST:   .WORD AREA
NUMBER: .WORD 4
```

is correct since the contents of LIST are the argument block pointer and the contents of NUMBER are the data value.

NOTE

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

You must be sure that the selected mode generates the correct value as a source operand in a MOV instruction. Check the programmed request macro in the Macro Library (SYSMAC.SML) and expand the programmed request by hand or with the macro assembler (by using the .LIST MEB directive) to be sure of correct results.

1.1.2.4 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 allows 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.

1.1.2.5 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(decimal) 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). Thus, 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. Your program refers to that file or device in all I/O operations thereafter by the assigned channel number. You can specify a device (non-file-structured) or a device and file name (file-structured).

Channel 255(decimal) is reserved for system use. Channel 15(decimal) is used by the system's overlay handler.

1.1.2.6 Device Blocks — A device block is a four-word block of Radix–50 information. You set up the block to specify a physical or logical device name, file name, and file type for use with a programmed request. This information is passed to the monitor when your program opens a file. The monitor 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: could be written as

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

The first word contains the device name, the second and third words contain the file name, and the fourth 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

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

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.

1.1.2.7 Programmed Request Errors — Programmed requests use three methods of reporting errors detected by the monitor:

1. Setting the carry bit of the processor status word (PSW)
2. Reporting the error code in byte 52 of the system communications area
3. 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 — that is, errors can be generated for different reasons — byte 52 in the system communications area is used to receive the error code. Thus, when the carry bit is set, check byte 52 to find out 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, since byte 53 has a different usage. The following example shows how byte 52 can be tested for the error code.

```
        ERRBYT=52
        .PRGREQ AREA,ARG1,,,,ARG2
        BCS ERROR
                    .
                    .
                    .
ERROR:  TSTB    @#ERRBYT
```

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).

1.1.2.8 User Service Routine (USR) Requirement — Many programmed requests require the USR to be in memory. Some of these requests always 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. Table 1–2 shows the programmed requests that require the USR.

Table 1–2: Programmed Requests Requiring the USR

Request	Monitor		
	SJ	FB	XM
.CDFN	yes*	no	no
.CLOSE (see Note 1)	yes	yes	yes
.CSIGEN	yes	yes	yes
.CSISPC	yes	yes	yes
.DELETE	yes	yes	yes
.DSTATUS	yes	yes	yes
.ENTER	yes	yes	yes
.EXIT	yes	yes	yes
.FETCH	yes	yes	yes
.FPROT	yes	yes	yes
.GTLIN	yes	yes	yes
.HRESET	yes	no	no
.LOCK (see Note 2)	yes	yes	yes
.LOOKUP	yes	yes	yes
.QSET	yes*	yes*	yes
.RELEAS	yes	yes	yes
.RENAME	yes	yes	yes
.SFDAT	yes	yes	yes
.SRESET	yes*	no	no
.TLOCK (see Note 3)	yes	yes	yes

Note 1: Only if channel was opened with an .ENTER programmed request

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

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

* The requests marked with an asterisk always require a fresh copy of the USR to be read in before they can be executed.

USR requirements for programmed requests differ between the SJ and FB 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.

The USR is not reentrant and cannot be shared by concurrent jobs. Thus, when the USR is in use by one job, another job requiring it must queue up for it. This is particularly important for concurrent jobs when devices such as magnetic tape or cassette are active. For example, USR file operations on tape devices require a sequential search of the tape. When a background program is running the USR, the foreground job is locked out until the tape operation is completed. You should be aware that this operation may take considerable time. The .SPFUN request can be used to perform asynchronous directory operations on tape. In the FB and XM monitors, the .TLOCK request can be used by a job to check 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, "swapped out" and stored in the file SWAP.SYS to provide room for the USR). The USR is then read into memory. Although swapping is invisible to you in normal operation, you must be aware of it and take some care 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 optimize programs so that they require little or no swapping, thereby saving time.

Consider the following items if a swap operation is necessary.

1. The background job
   If a .SETTOP request in a background job specifies an address beyond the point at which the USR normally resides, a swap is required when the USR is called. Section 2.79 details the operation of the .SETTOP request. This case is not encountered in XM because the USR is always resident.

2. The 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, this indicates that 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 occurs. The SET USR NOSWAP command makes the USR permanently resident.

   If you specify an alternate address in location 46, the SJ monitor does not verify the validity of the USR swap address. Thus, if the area to be swapped overlays the resident monitor, the system is destroyed.

3. Monitor offset 374
   The contents of monitor offset 374 indicate 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.

4. Protecting program areas
   Make sure that certain areas of your program do not get overlaid when you swap in the USR. These areas are the program stack, any parameter block for calls to the USR, the EMT instruction that invoked the USR, I/O buffers, device handlers, interrupt service routines, queue elements, defined channels, and completion routines in use when the USR is being called.

The *RT–11 Software Support Manual* provides additional information on the USR.

1.1.3 Using Programmed Requests

This section describes how to use and implement programmed requests to access the various monitor services. Chapter 2 contains, in alphabetical order, detailed descriptions of each request, including examples.

1.1.3.1 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, and error processing.

MEMORY ALLOCATION
The memory needs of a program are specified to the monitor by the .SETTOP request. When loaded, a program occupies the memory specified by its image created at link time. To obtain more memory, a .SETTOP request is executed, with R0 containing the highest address desired. The monitor returns the highest address available. Resident handlers or foreground jobs can prevent all the memory that is desired from being available to the program. If the memory requirements of the running program permit, the monitor retains the User Service Routine (USR) in memory, which reduces swapping. Otherwise, the monitor will automatically remove the USR from memory and then swap part of the user program to the swap file called SWAP.SYS on the system device whenever the USR must be reloaded to process a request. The .SETTOP request then allows you to determine how much memory is available and to control 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 Software Support Manual.*

If a program needs so much memory that the USR must swap, swapping will automatically occur whenever a USR call is made. However, if a program knows what file operations are necessary, and if these operations can be consolidated and performed at one time, the efficiency of the system can be enhanced in the following manner: request the USR to be swapped in, have it remain resident while a series of consecutive USR operations is performed, then 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. It can operate by: (1) requiring a portion of your program to be written to the swap blocks prior to reading in the USR; (2) only requiring a fresh copy of the USR if the USR buffer is overwritten; or (3) not requiring the USR to be read in if it finds the USR intact. 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 foreground/background programs, but 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(decimal) channels as part of the job's impure area — that is, 16 files can be active at one time. Up to 255(decimal) channels can be activated 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 allows the job to 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 (ten words [decimal] long when using the extended memory monitor). 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 (available only in FB or XM) allows you to 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.

In SJ, a hard reset is done at .EXIT or CTRL/C. This clears all devices.

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. RT–11 automatically prevents alteration of all 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. You use the .PROTECT programmed request to allow the foreground job to 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.

The request .SPFUN is available 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. Examples include directory I/O errors, monitor I/O errors on the system device, or I/O requests to nonexistent devices. Some programs cannot afford to allow the monitor to abort the job because of such errors. For example, in the case of RT–11 multi-user BASIC, 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. A .HERR request (normal default) indicates that the monitor will handle severe errors and stop the job. A .SERR request causes the monitor to return most errors to your program in byte 52 for appropriate action.

In addition to processing I/O errors through .HERR and .SERR requests, you can also handle certain fatal errors through the .TRPSET or .SFPA requests. You use these requests to prevent your program from aborting due to a trap to location 4 or 10(octal), 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 that is called when an exception trap occurs.

1.1.3.2 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.

The .DATE request obtains the system date, which then can be printed on a report or entered as a data record in a file. The time-of-day can be obtained with a .GTIM request and used in the same way. A program can set the system date and/or time by using the .SDTTM programmed request. Changing the date or time has no effect on any outstanding mark time or timed wait requests.

With a .GTJB request you can obtain information on whether the job is running in the foreground or background, the memory limits of the job, the virtual high limit for a job created with the linker /V option (XM only), the unit number of the job's console terminal (if you are using the multiterminal feature), the address of the job's channel area, the address of the job's impure area, and the job's logical job name (if you are using a monitor with the system job feature).

Status information on a file — such as its starting block, its length, and the device it is located on — can be obtained with a .CSTATUS request. Status information on a device — such as its block length and controller-assignment number — can be obtained with a .DSTATUS request.

The .MTGET and .MTSTAT programmed requests provide multiterminal status information when the multiterminal feature is being used.

The programmed requests .MFPS and .MTPS read the priority bits and set the priority and T-bits in the processor status word (PSW). These requests allow a program to run without change on any processor from an LSI–11 to a PDP–11/60.

1.1.3.3 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 in the following general form:

```
*Dev:Output/Option=Dev:Input/Option
```

The asterisk is printed on the terminal by the monitor. The RT–11 system programs use the same command string (see the *RT–11 System Utilities Manual*).

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. Thus, 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(octal) to accommodate six input file specifications.

The .CSISPC request provides you with the services of the command processor, but allows you to do your own device and file manipulation. When you use .CSISPC, the CSI obtains a command string, analyzes it syntactically, places the results in a table, and passes the table to your program for appropriate action.

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

1.1.3.4 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 (in FB and XM) 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. Section 1.1.3.5 of this manual describes the use of programmed requests for performing I/O operations. The *RT–11 Software Support 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:

1. One-half the size of the largest unused space or all of the second largest space, whichever is larger (the default)
2. A space of a specific size
3. As much space as possible

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

When file operations are completed, a .CLOSE request makes the new file permanent in the directory. 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. The .SAVESTATUS request 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. The .REOPEN request causes the inactive file to be reactivated on the specified

channel, and I/O continues on that channel. In this manner, you can open more files than there are channels. If, in addition, you lock the USR in memory, you can open all the files your job needs while maintaining system swapping efficiency. The procedure is:

1. Lock the USR in memory, and open the files that are needed.
2. Issue the .SAVESTATUS request.
3. Release the USR.
4. Issue a .REOPEN request each time a file is needed.
5. Lock USR, and use 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, improving program efficiency.

The creation date and protection status of a file can be modified by using the .SFDAT and .FPROT requests, respectively.

The .SFDAT request allows you to 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.

The .FPROT request 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.

1.1.3.5 Input/Output Operations — You can perform I/O in three different modes:

- synchronous
- asynchronous
- event-driven

These modes allow you to optimize the overlap of CPU and I/O processing.

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; thus the program and the I/O process are synchronized.

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

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 that a process is never completed.

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

COMPLETION ROUTINES

Completion routines are part of your program. They execute following the completion of some external operation, interrupting the normal program flow. On entry to an I/O completion routine, Register 0 contains the contents of the Channel Status Word and Register 1 contains the channel number for the operation. The carry bit is not significant.

Completion routines are handled differently, depending on whether the program is being run under the SJ monitor or the FB and XM monitors. Under the SJ monitor, completion routines are totally asynchronous and can interrupt each other. An interrupted completion routine is resumed when the interrupting routine is finished. Unlike completion routines run under FB and XM monitors, which are serialized and run at priority 0, completion routines run under an SJ monitor are nested. In addition, they execute not at priority 0, but at the same priority as the device whose interrupt scheduled them. For example, the completion routine resulting from a .WRITC programmed request to device TT: runs at priority 4. Completion routines from timer requests run at the same priority as the system clock. This is particularly important on LSI–11 and PDP 11/03 systems that have only two interrupt levels, ON and OFF, because clock interrupts may be lost while lengthy completion routines execute. Under the FB and XM monitors, completion routines do not interrupt one another. Instead, they are queued, and the next 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 in 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 under the FB and XM monitors, it is possible to request a completion routine from an in-line interrupt service routine through a .SYNCH programmed request. This allows the interrupt service routine to issue other programmed requests to the monitor.

Restrictions that must be observed when writing completion routines are as follows:

1. Requests that require the USR must not be issued within a completion routine. A fatal monitor error is generated if the USR is called from a completion routine.
2. 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.
3. 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.
4. Under the XM monitor, completion routines must remain mapped while the request is active and the routine can be called.
5. The completion routine must exit with an RTS PC instruction because the routine was called from the monitor with a JSR PC,ADDR, where ADDR is the user-supplied entry point address. If you exit completion routines with an .EXIT request, your job will abort. An exit from a completion routine in FB or XM can be done by using an .SPCPS request to change the mainline PC to point to an .EXIT in the main code. As soon as all completion routines are done, the exit will be executed.
6. Under the XM monitor, completion routines scheduled as a result of a .SYNCH run in kernel mapping, not user mapping.

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. The .SPCPS programmed request, which can only be used in a foreground/background or extended memory environment, 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 rerouting can occur.

TERMINAL INPUT/OUTPUT

Several programmed requests are available to provide an I/O capability with the console terminal: a .TTYIN request obtains a character from the console; a .TTYOUT request prints a character on the terminal; long strings of characters — even multiple lines — are output with the .PRINT request. Programs can also 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. 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.

You can use the .TTYIN/.TTINR requests in special (single-character) mode by setting bit 12 of the Job Status Word. See the .TTYIN programmed request for a description of special mode.

MULTITERMINAL REQUESTS

The RT–11 multiterminal feature allows your program to perform input/output on up to 17 terminals. Several programmed requests allow you 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.

The .MTIN request returns to a program characters that are typed at the terminal, while the .MTOUT and .MTPRNT requests send characters to a terminal. These requests are analogous to the .TTYIN, .TTYOUT, and .PRINT requests. Note that the .TTYIN/.TTINR, .TTYOUT/.TTOUTR, and .PRINT requests can only be used with the console terminal.

You can set 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 request provides multiterminal system status information.

1.1.3.6 Foreground/Background Communications — Communication between foreground and background jobs is obtained through the programmed requests .SDAT and .RCVD. These requests also have three modes (synchronous, asynchronous, and event driven) that allow transfer of buffers 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 in a foreground/background environment to read or write data in a file opened by the other 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 allows you to obtain channel information from the foreground job and to use that channel information to control a read or write request.

The foreground/background monitor always causes a context switch of critical items such as machine registers, the job status area, and floating-point processor registers 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.

1.1.3.7 Timer Support — Timer support by the monitor is provided through the .MRKT request. With the .MRKT request, you specify the address of a routine that is to 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 — as many as the queue can hold — are identified by number. Pending timer requests can be canceled with a .CMKT request. .MRKT requests are normally used as a scheduling tool where jobs are scheduled on the basis of clock events, detected by timer completion routines.

A job can be suspended for a specified time interval with a .TWAIT request. For example, the .TWAIT request will allow a compute-bound job to relinquish CPU time to the rest of the system, permitting other jobs to run.

1.1.3.8 Program Termination or Suspension — Many jobs come to an execution point where there is no further processing necessary until an external event occurs. In the FB or XM 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 continue the job at the point where it was suspended.

When a job is ready to terminate or reaches a serious error condition, it can reset the system with the .SRESET and .HRESET requests. .SRESET is a soft reset. That is, the monitor data base for the job is reinitialized, but queued I/O is allowed to run to completion. .HRESET is a hard reset where all I/O for the job is stopped (by a RESET instruction in the SJ monitor or by calls to the handlers in an FB environment).

Use the programmed request .EXIT in a background job to return control to the keyboard monitor by causing program termination. If Register 0 contains a zero upon execution of this request, a hard reset is performed, and the commands REENTER, CLOSE, and START are disabled. If Register 0 contains a non-zero value upon exit from your program, a soft reset is done, and these commands are not disabled. In a foreground job, an .EXIT programmed request stops the job but does not return control to the keyboard monitor. The job can be removed from memory by 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 adjust processing. In FORTRAN, channel information is stored in the job's impure area, and this information is not preserved across a .CHAIN request. Thus, close any channels in the first program, and reopen them in the program being chained to.

1.1.3.9 System Job Communications — System job support allows communications between any two jobs in the system. The background job, designated by the logical job name 'B', and the foreground job, designated by the logical job name 'F', can send and receive messages between each other by using the .RCVD and .SDAT programmed requests.

All jobs (that is, background, foreground, and system jobs) can communicate with each other by using the Message Handler (MQ). The MQ handler performs like an ordinary RT–11 device handler in the way it accepts and dispatches I/O requests from the queued I/O system. This permits .READ and .WRITE requests 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. Note that the Message Handler (MQ) can also be used under the distributed FB monitor; it does not require the system job special feature.

Care should be taken 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 System User's Guide*).

1.1.3.10 Extended Memory Functions — The RT–11 extended memory (XM) monitor permits MACRO programs to access extended memory by mapping their virtual addresses to physical locations in memory. This is done in conjunction with a hardware option called the Memory Management Unit that converts a 16-bit virtual address to an 18- or 22-bit physical address.

You access extended memory in a program through programmed requests. In accessing extended memory, you must first establish window and region definition blocks. Next, you must specify the amount of physical memory the program requires and describe the virtual addresses you plan to use. Do this by creating regions and windows. Then, associate virtual addresses with physical locations by mapping the windows to the regions. You can 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, you can manipulate the mapping of windows to regions that best meet your requirements.

There are four types of extended memory programmed requests:

1. Window requests
2. Region requests
3. Map requests
4. Status requests

The window and region requests have their own data structures. RT–11 provides the macro .WDBBK to create a window definition block and the macro .RDBBK to create 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.

Certain programmed requests are restricted when they are in an extended memory 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(decimal) 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 Software Support Manual.*

1.1.3.11 Interrupt Service Routines — The system macro library (SYSMAC.SML) contains some macros that are not programmed requests, but are used like programmed requests in interrupt service communication to the monitor. The first macro call in every interrupt routine is .INTEN, which causes the system to use the system stack for interrupt service and allows the monitor scheduler to make note of the interrupt. If device service is all the routine does, .INTEN is the only call it need make. If you need to issue one or more programmed requests, such as .READ or .WRITE from the interrupt service routine, you must issue the .SYNCH call. The .INTEN call described above 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 the .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 Software Support Manual* contains a detailed description of interrupt service routines.

1.1.3.12 Device Handlers — The system macro library (SYSMAC.SML) contains several macros that simplify the writing of a device handler. A device handler is divided into several sections. These sections are as follows:

- Preamble section
- Header section
- I/O initiation section
- Interrupt service section
- I/O completion section
- Termination section

The .DRDEF macro is used near the beginning of your device handler, and performs much of the work in the preamble section. The .DRBEG macro sets up the first five words in the header section, stores information in block 0 of the handler file, and creates some global symbols. The .DRAST macro sets up the interrupt entry point and the abort entry point in the interrupt service section, and lowers the processor priority. The .DRFIN macro generates the instructions for the jump back to the monitor at the end of the handler I/O completion routine. The .DREND macro generates handler termination code. 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 .DRSET macro sets up the option table for the SET command in block 0 of the device handler file. The .DRVTB macro sets up a table of vectors for devices that require more than one vector.

Each of the device handler macros is described in Chapter 2. The *RT–11 Software Support Manual* details the use of these macros in writing a device handler.

1.1.4 Compatibility With Previous RT–11 Versions

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

1.1.4.1 Version 1 Programmed Requests — Programmed requests provided with the first release of RT–11, such as .READ and .WRITE, were designed for a single-user, single-job environment. As such, they differ significantly from the programmed requests of the later versions. For Version 1 requests, arguments were pushed on the stack instead of being stored, as they are presently, in an argument block. The channel number was limited to the range 0 through 17(octal), while later versions can allocate an additional number of channels. Also, no arguments could be omitted in the macro call.

Programs written for use under Version 1 assemble and execute properly under Versions 3, 4, and 5 when the ..V1.. macro call is used. The ..V1.. macro call causes all Version 1 programmed requests to expand exactly as they did in Version 1. The ..V2.. macro call expands all requests in Version 2 format. However, it is to your advantage to convert Version 1 and Version 2 programs to the current format for programmed requests (see Section 2.7). Future versions of RT–11 may no longer support the older formats.

1.1.4.2 Version 2 Programmed Requests — The release of RT–11 Version 2 included new programmed requests and a different way of handling arguments. The new programmed requests reflected RT–11's ability to run a foreground job as well as a background job. They included requests to suspend/resume the foreground job and to share messages and data between the two jobs.

Arguments in Version 2 programmed requests were stored in an argument block instead of on the stack. Another difference in Version 2 was that arguments could be omitted from macro calls. If the Area argument — that is, the pointer to the argument block — was omitted, the macro assumed that R0 pointed to a valid argument block. If any of the optional arguments were not present, the macro placed a zero in the argument block for the corresponding argument. Version 1 programmed requests were modified to incorporate these changes, and the ..V1.. macro was provided to allow Version 1 programmed requests to execute under Version 2 without further modification.

Programs written for use under Version 2 assemble and execute properly under Versions 3 and 4 when the ..V2.. macro call is used. The ..V2.. macro call causes all programmed requests prior to Version 3 to expand in Version 2 format.

1.1.4.3 Version 3 Programmed Requests — The programmed requests for Version 3 provide means for user programs to access regions in extended memory and to use more than one terminal. The chief difference between Version 2 and Version 3 programmed requests is the way in which omitted arguments are handled. In Versions 3, 4, and 5, blank arguments in the macro calls do not cause zeros to be entered into the argument block, but leave the corresponding argument block entry for the missing argument untouched.

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

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

Programmed requests from previous versions were modified to incorporate this change, and the ..V2.. macro call was provided so that Version 2 pro-

grams could execute properly under Version 3 without further modification.

1.1.4.4 Version 4 Programmed Requests — Certain programmed requests have taken on additional functions to support the system job feature. These programmed requests and their additional functions follow:

Programmed Request	Added Function
.GTJB	Returns logical job name.
.CHCOPY	May specify logical job name.
.LOOKUP	Opens message channel to any job; issues .READ/C/W, .WRITE/C/W, and .WAIT requests to communicate between jobs.

1.1.4.5 Version 5 Programmed Requests — Version 5 added several new programmed requests and modified others. The requests added for Version 5 are .ABTIO, .FPROT, .PEEK, .POKE, .PVAL, and .SFDAT. Although the XM monitor was expanded to support 22-bit Q-bus addressing, from a program's point of view the extended memory programmed requests are unchanged from Version 4. Channel number 377 is now reserved for system use and is always unavailable to user programs. The programmed requests which have changed between Version 4 and Version 5 are summarized below:

Programmed Request	Added Function
.CSTAT	Available under SJ monitor.
.TWAIT	Available under SJ monitor.
.FETCH	Available under XM monitor.
.GTLIN	Optional argument to force terminal input.

1.1.5 Programmed Request Conversion

The previous sections describe the modified format of programmed requests that were developed after those of Version 1. This section describes the conversion process from the Version 1 format to Version 3.

1.1.5.1 Macro Calls Not Requiring Conversion — Version 1 macro calls that do not require any conversion are as follows:

.CSIGEN	.LOCK	.SRESET
.CSISPC	.PRINT	.TTINR (Note 2)
.DATE	.QSET	.TTOUTR (Note 2)
.DSTATUS	.RCTRLO	.TTYIN
.EXIT	.RELEAS	.TTYOUT
.FETCH	.SETTOP (Note 1)	.UNLOCK
.HRESET		

Note 1: Provided that location 50 is examined for the maximum value.

Note 2: Except in FB or XM systems.

1.1.5.2 Macro Calls That Can Be Converted — Version 1 macro calls that can be converted are as follows:

.CLOSE	.RENAME
.DELETE	.REOPEN
.ENTER	.SAVESTATUS
.LOOKUP	.WAIT
.READ	.WRITE

The general format of the ..V1.. system macro is

```
.PRGREQ Chan,Arg1,...,Argn
```

In this form, *Chan* is an integer between 0 and 17(octal) and is not a general assembler argument. The channel number is assembled into the EMT instruction itself. The arguments *Arg1* through *Argn* are either moved into R0 or pushed on the stack.

The ..V2.. equivalent of the above call is

```
.PRGREQ Area,Chan,Arg1,...,Argn
```

In this form, the *Chan* argument can be any legal assembler argument and can be in the range from 0 to 377(octal). *Area* points to an argument block where the arguments *Arg1* through *Argn* will be placed.

For example, consider a .READ programmed request in both forms:

Version 1:

```
.READ 5,#BUFF,#256.,BLOCK
```

Version 2:

```
.READ #AREA,#5,#BUFF,#256.,BLOCK

.
.
.

AREA:   .WORD 0 ;CHANNEL/FUNCTION CODE HERE
        .WORD 0 ;BLOCK NUMBER HERE
        .WORD 0 ;BUFFER ADDRESS HERE
        .WORD 0 ;WORD COUNT HERE
        .WORD 0 ;A 1 GOES HERE
```

Thus, the difference in the two macro calls is that Version 2 declares the channel number as a legal assembler argument and adds an Area argument.

Table 1–3 shows a complete list of conversions for the programmed requests that can be converted. Version 1 and Version 2 formats are given. In Versions 3 and later, this function is performed automatically. The arguments shown inside the square brackets ([ ]) are optional. Refer to the appropriate section in Chapter 2 for more details on each request.

Table 1–3: Programmed Request Conversions (Version 1 to Version 2)

Version	Programmed Request
V1:	.DELETE chan,dblk
V2:	.DELETE area,chan,dblk[,count]
V1:	.LOOKUP chan,dblk
V2:	.LOOKUP area,chan,dblk[,count]
V1:	.ENTER chan,dblk[,length]
V2:	.ENTER area,chan,dblk[,length[,count]]
V1:	.RENAME chan,dblk
V2:	.RENAME area,chan,dblk
V1:	.SAVESTAT chan,cblk
V2:	.SAVESTAT area,chan,cblk
V1:	.REOPEN chan,cblk
V2:	.REOPEN area,chan,cblk
V1:	.CLOSE chan
V2:	.CLOSE chan
V1:	.READ/.READW chan,buff,wcnt,blk
V2:	.READ/.READW area,chan,buff,wcnt,blk
V1:	.READC chan,buff,wcnt,crtn,blk
V2:	.READC area,chan,buff,wcnt,crtn,blk
V1:	.WRITE/.WRITW chan,buff,wcnt,blk
V2:	.WRITE/.WRITW area,chan,buff,wcnt,blk
V1:	.WRITC chan,buff,wcnt,crtn,blk
V2:	.WRITC area,chan,buff,wcnt,crtn,blk
V1:	.WAIT chan
V2:	.WAIT chan

Several important features of Version 3 calls to be kept in mind when using them are as follows:

1. Version 3 calls require the *area* argument, which points to the area where the other arguments will be (unless R0 already points to it and the first word is set up).
2. Enough memory space must be allocated to hold all the required arguments.
3. The *chan* argument must be a legal assembler argument, not just an integer between 0 and 17(octal).
4. Blank fields are permitted in the Version 3 calls. Any field not specified (left blank) is not modified in the argument block.

1.1.6 Programmed Request Summary

Many programmed requests operate only in a specific RT–11 environment, such as under a foreground/background monitor or when using a special feature such as multiterminal operation. Table 1–4 lists the programmed requests that can be used in all RT–11 environments, including multiterminal operation. Table 1–5 lists the additional programmed requests that can be used under the foreground/background monitor and extended memory monitor. 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 described.

Macros that are 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. The programmed requests followed by (MT) work only under the multiterminal feature of RT–11.

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

Name	EMT	Code	Purpose
.ABTIO	374	13	Aborts I/O in progress on the specified channel
.CDFN	375	15	Defines additional channels for I/O
.CHAIN	374	10	Chains to another program (in background job only)
.CLOSE	374	6	Closes the specified channel
.CMKT	375	23	Cancels an unexpired mark time request (special feature in single-job environment)
.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 (special feature)
.DATE	374	12	Moves the current date information into R0
.DELETE	375	0	Deletes the file from the specified device
.DRAST	—	—	Used with device handlers to create the asynchronous entry points to the handler
.DRBEG	—	—	Used with device handlers to create a five-word header, and .ASECT locations 52 through 60
.DRBOT	—	—	Used with system device handlers to set up the primary driver

(continued on next page)

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

Name	EMT	Code	Purpose
.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
.DRFIN	—	—	Used with device handlers to generate the code required to exit to the completion code in the resident monitor
.DRSET	—	—	Used with device handlers to create the list of SET options for a device
.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
.DSTATUS	342	—	Returns the status of a particular 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
.GTIM	375	21	Gets the time of day
.GTJB	375	20	Gets parameters of a job
.GTLIN	345	—	Accepts an input line from either an indirect command 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

(continued on next page)

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

Name	EMT	Code	Purpose
.MFPS	—	—	Reads the priority bits in the processor status word, but does not read the condition codes
.MRKT	375	22	Marks time, that is, sets an asynchronous routine to be entered after specified interval (special feature in single-job environment)
.MTATCH (MT)	375	37	Attaches a terminal for exclusive use by the requesting job
.MTDTCH (MT)	375	37	Detaches a terminal from one job and frees it for use by other jobs
.MTGET (MT)	375	37	Returns the status of a specified terminal to the user
.MTIN (MT)	375	37	Operates as a .TTYIN request for a multiterminal configuration
.MTOUT (MT)	375	37	Operates as a .TTYOUT request for a multiterminal configuration
.MTPRNT (MT)	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
.MTRCTO (MT)	375	37	Resets the CTRL/O flag for the designated terminal
.MTSET (MT)	375	37	Modifies terminal status in a multiterminal configuration
.MTSTAT (MT)	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 byte
.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, overriding any previous CTRL/O
.READ	375	10	Transfers data on the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue; no special action is taken upon completion of I/O

(continued on next page)

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

Name	EMT	Code	Purpose
.READC	375	10	Transfers data on the specified channel to a memory buffer and returns control to the user program when the transfer request is entered in the I/O queue; upon completion of the read, control transfers asynchronously to the completion routine specified in the .READC request
.READW	375	10	Transfers data via the specified channel to a memory buffer and returns control to the user program only after the transfer is complete
.RELEASE	343	—	Removes a device handler from memory
.RENAME	375	4	Changes the name of the indicated file to a new name; if this request is attempted when using magtape, the handler returns an invalid operation code
.REOPEN	375	6	Restores the parameters stored via a .SAVESTATUS request and reopens the channel for I/O
.SAVESTATUS	375	5	Saves the status parameters of an open file in user memory and frees the channel for 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
.SFPA	375	30	Sets user interrupt for floating-point processor exceptions
.SPFUN	375	32	Performs special functions on magtape, cassette, diskette, and some disk devices
.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 (special feature in all environments)
.TLOCK	374	7	Indicates if the USR is currently used by another job and performs exactly as a .LOCK request in a single-job environment
.TRPSET	375	3	Sets a user intercept for traps to monitor locations 4 and 10

(continued on next page)

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

Name	EMT	Code	Purpose
.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
.UNLOCK	347	—	Releases the USR after execution of a .LOCK and swaps in the user program, if required
..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	375	11	Transfers data on the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue; upon completion of the write, control transfers asynchronously to the completion routine specified in the .WRITC request
.WRITE	375	11	Transfers data on the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue; no special action is taken upon completion of the I/O
.WRITW	375	11	Transfers data on the specified channel to a device and returns control to the user program only after the transfer is complete

Table 1–5 lists the additional programmed requests that can be used only in a foreground/background and extended memory environment. The programmed requests followed by (XM) operate only in an extended memory environment.

Table 1–5: Foreground/Background and Extended Memory Programmed Requests

Name	EMT	Code	Purpose
.CHCOPY	375	13	Allows one job to access another job's channel
.CNTXSW	375	33	Requests that the indicated memory locations be part of FB or XM context switch process
.CRAW (XM)	375	36	Creates a window in virtual memory
.CRRG (XM)	375	36	Creates a region in extended memory
.DEVICE	375	14	Allows device interrupts in FB or XM to be disabled upon program termination
.ELAW (XM)	375	36	Eliminates an address window in virtual memory
.ELRG (XM)	375	36	Eliminates an allocated region in extended memory

(continued on next page)

Table 1–5: Foreground/Background and Extended Memory Programmed Requests (Cont.)

Name	EMT	Code	Purpose
.GMCX (XM)	375	36	Returns mapping status of a specified window
.MAP (XM)	375	36	Maps a virtual address window to extended memory
.MWAIT	374	11	Waits for messages to be processed
.PROTECT	375	31	Requests that specified vectors in the area from 0 to 476 be given exclusively to the current job
.RCVD .RCVDC .RCVDW	375	26	Receives data — allows a job to read messages or data sent by another job in an FB environment. The three modes correspond to the .READ, .READC, and .READW requests
.RDBBK (XM)	—	—	Reserves space in a program for a region definition block and sets up the region size and region status word
.RDBDF (XM)	—	—	Defines the offsets and bit names associated with a region definition block
.RSUM	374	2	Causes the mainline code of the job to be resumed after it was suspended by a .SPND request
.SDAT .SDATC .SDATW	375	25	Sends messages or data to the other job in an FB environment. The three modes correspond to the .WRITE, .WRITC, and .WRITW requests
.SPCPS	375	41	Used in a completion routine to change the flow of control of the mainline code (special feature)
.SPND	374	1	Causes the running job to be suspended
.UNMAP (XM)	375	36	Unmaps a virtual address memory window
.UNPROTECT	375	31	Cancels the .PROTECT vector protection request
.WDBBK (XM)	—	—	Reserves space in a program for a window definition block and sets up the associated data
.WDBDF (XM)	—	—	Defines the offsets and bit names associated with a window definition block

1.2 Using the System Subroutine Library

The system subroutine library is a collection of FORTRAN-callable routines that allow various RT–11 system features to be used by a FORTRAN programmer. There are no FORTRAN routines in SYSLIB to access extended memory under the extended memory (XM) monitor.

This collection of subroutines is placed in a system library called SYSLIB.OBJ. This library file also contains the overlay handlers, utility functions, a character string manipulation package, and two-word integer support routines. The linker uses this library to resolve undefined globals. It is resident on the system device (SY:).

You should be familiar with the *PDP–11 FORTRAN Language Reference Manual* and the *RT–11/RSTS/E FORTRAN IV User's Guide* before using the material in this chapter.

The system subroutine library provides the following capabilities:

1. Complete RT–11 I/O facilities, including synchronous, asynchronous, and event-driven modes of operation. FORTRAN subroutines can be activated upon completion of an input/output operation.
2. Timed scheduling of completion routines. This feature is standard in the FB and XM monitors, and is a special feature in the SJ monitor.
3. Facilities for communication between foreground and background jobs.
4. FORTRAN language interrupt service routines for user devices.
5. Complete timer support facilities, including timed suspension of execution in a FB or XM environment, conversion of different time formats, and time-of-day information. The timer support facilities can use either 50- or 60-cycle clocks.
6. All RT–11 auxiliary input/output functions, including the capabilities of opening, closing, renaming, and creating or deleting files on any device.
7. All monitor-level information functions, such as job partition parameters, device statistics, and input/output channel statistics.
8. Access to the RT–11 command string interpreter (CSI).
9. A character string manipulation package supporting variable-length character strings.
10. INTEGER*4 support routines that allow two-word integer computations.

NOTE

When variables are described or mentioned, and unless otherwise specified, INTEGER means INTEGER*2, (16-bit integer) and REAL means REAL*4 (single-precision floating point). Integer and real arguments to subprograms are indicated in this section as follows:

i = INTEGER*2 arguments
j = INTEGER*4 arguments
a = REAL*4 arguments
d = REAL*8 arguments

In general, the routines in SYSLIB were written for use with RT–11 V2 or later and FORTRAN IV V1B or later versions. The use of SYSLIB with prior versions of RT–11 or FORTRAN may lead to unpredictable results.

1.2.1 System Conventions

This section describes system conventions that must be followed for proper operation of calls to the system subroutine library. Certain restrictions that apply are described in Section 1.2.1.7.

1.2.1.1 Channel Numbers — A channel number is a logical identifier for a file used by FORTRAN. Thus, when you open a file on a particular device, you assign a channel number to that file. To refer to an open file, it is only necessary to refer to the appropriate channel number.

The FORTRAN system has 16(decimal) channels available for your use. The call IGETC assigns a channel to your program and notifies the FORTRAN I/O system, which also uses these channels, that the channel is in use. When there is no longer need for a channel, the program should close the channel with a CLOSEC, ICLOSE or a PURGE SYSLIB call. The channel should also be freed and returned to the FORTRAN I/O system with a IFREEC call.

Up to 254(decimal) channels can be activated with the ICDFN call. This function sets aside memory in the job area to accommodate status information for the extra channels. Use the ICDFN call during the initialization phase of your program. You can use all channels numbered higher than 15(decimal). The FORTRAN I/O system uses channels 0 through 15(decimal).

Channels must be allocated in the main program routine or its subprograms. Do not allocate channels in routines that are activated as the result of I/O completion events or ISCHED or ITIMER calls.

1.2.1.2 Completion Routines — Completion routines can be written in FORTRAN or assembly language, depending upon the function called.

A completion routine is a subprogram that executes asynchronously with a main program and is scheduled to run as soon as possible after the completion of an associated event, such as an I/O transfer or the passing of a specified time interval. All completion routines of the current job have higher priority than other parts of the job. Therefore, once a completion routine becomes runnable because of its associated event, it interrupts execution of the job and continues to execute until it relinquishes control.

Completion routines are handled differently in the SJ and the FB and XM monitors. In the SJ monitor, these routines are totally asynchronous and can interrupt one another. Unlike completion routines run under FB and XM monitors, which are serialized and run at priority 0, completion routines run under an SJ monitor are nested and can interrupt each other. In addition, they execute not at priority 0, but at the same priority as the device whose interrupt scheduled them. For example, the completion routine resulting from a .WRITC programmed request to device TT: runs at priority 4. Completion routines from timer requests run at the same priority as the system clock. This is particularly important on LSI–11 and PDP/03 systems that have only two interrupt levels, ON and OFF, because clock interrupts may be lost while lengthy completion routines execute. In the FB and XM monitors, completion routines do not interrupt each other but are queued and have to wait until the correct job is running. They are then scheduled on a first-in first-out basis.

Assembly language completion routines exit with an RTS PC instruction. FORTRAN completion routines exit by the execution of a RETURN or END

statement in the subroutine. All names of completion routines external to the routine being coded that are passed to scheduling calls must be specified in an EXTERNAL statement in the FORTRAN program unit issuing the call.

A completion routine written in FORTRAN can have a maximum of two arguments as follows:

Form: SUBROUTINE crtn [(iarg1,iarg2)]

where:

crtn	is the name of the completion routine
iarg1	is equivalent to R0 on entry to an assembly language completion routine
iarg2	is equivalent to R1 on entry to an assembly language completion routine

If an error occurs in a completion routine or in a subroutine at completion level, the error handler traces back through to the original interruption of the main program. Thus, the traceback is shown as though the completion routine were called from the main program. This lets you know where the main program was executing, so that when an error is fatal, it can be diagnosed and corrected.

Certain restrictions apply to completion routines that are activated by the following calls:

INTSET	ISDATF
IRCVDC	ISPFNC
IRCVDF	ISPFNF
IREADC	ITIMER
IREADF	IWRITC
ISCHED	IWRITF
ISDATC	MRKT

The restrictions that apply when using these calls are as follows:

- No channels can be allocated by calls to IGETC or freed by calls to IFREEC from a completion routine. Channels to be used by completion routines should be allocated and placed in a COMMON block for use by the routine.
- The completion routine cannot perform any call that requires the use of the USR, such as LOOKUP and IENTER. See Section 1.2.1.5 for a list of the SYSLIB functions that call the USR.
- Files that are used by the completion routine must be opened and closed by the main program. There are, however, no restrictions on the input or output operations that can be performed in the completion routine. If many files must be made available to the completion routine, they can be opened by the main program and saved for later use (without tying up RT–11 channels) by an ISAVES call. The completion routine can later

make them available by reattaching the file to a channel with an IRE-OPN call.

Even if the completion routine itself does not issue any programmed requests, but does perform I/O to a logical unit number through the OTS, that logical unit number must be opened from the main level. To accomplish this, either the first I/O access or an OPEN statement must be issued from main level. A completion routine may not call CLOSE to close a logical unit.

- FORTRAN subroutines are reusable but not reentrant. That is, a given subroutine can be used many times as a completion routine or as a routine in the main program, but a subroutine executing as main program code does not work properly if it is interrupted and then called again at the completion level. This restriction applies to all subroutines that can be invoked at the completion level while they are active in the main program.
- FORTRAN completion routines can be called only by SYSLIB functions that end in F. Conversely, MACRO completion routines cannot be called by SYSLIB functions that end in F. Refer to Section 1.1.3.5 for details of other restrictions on MACRO completion routines.
- Under the SJ monitor, only one completion function should be active at any time.

1.2.1.3 Device Blocks — A device block is a four-word block of Radix–50 information that specifies a physical device and a file name. In FORTRAN, you can use one of three different methods to set up this block as follows:

1. You can use the DIMENSION and DATA statements. For example,

```
DIMENSION IFILE (4)
DATA IFILE/3RSY ,3RFIL,3RE ,3RXYZ/
```

2. You can translate the available ASCII file description string into Radix–50 format, using the SYSLIB calls IRAD50, R50ASC, and RAD50. For example,

```
REAL*8 FSPEC
CALL IRAD50 (12,'SY FILE  XYZ', FSPEC)
```

3. You can use the SYSLIB call ICSI to call the Command String Interpreter (CSI) to accept and parse standard RT–11 command strings.

1.2.1.4 INTEGER*4 Support Functions — This section discusses the initialization of INTEGER*4 variables for the FORTRAN programmer. Section 1.2.6.3 describes the use of INTEGER*4 functions for use by the MACRO programmer.

When the DATA statement is used to initialize INTEGER*4 variables, it must specify both the low- and high-order parts. For example, the code

```
INTEGER*4 J
DATA J/3/
```

Initializes only the first word. The correct way to initialize an INTEGER*4 variable to a constant such as 3 is as follows:

```
INTEGER*4 J
INTEGER*2 I(2)
EQUIVALENCE (J,I)
DATA I/3,0/      !INITIALIZE J TO 3
```

If you are initializing an INTEGER*4 variable to a negative value such as –4, the high-order (second word) part must be the continuation of the two's complement of the low-order part. For example,

```
INTEGER*4 J
INTEGER*2 I(2)
EQUIVALENCE (J,I)
DATA I/-4,-1/    !INITIALIZE J TO -4
```

The following example is suitable for initializing INTEGER*4 arguments to subprograms:

```
INTEGER*2 J(2)
DATA J/3,0/      !LOW ORDER,HIGH ORDER
```

1.2.1.5 User Service Routine (USR) Requirements — User-written routines that interface to the FORTRAN Object Time System (OTS) must account for the location of the RT–11 User Service Routine (USR). The USR occupies 2K words. When your program calls a SYSLIB routine that requests a USR function (such as IENTER or LOOKUP), or when the USR is invoked by the FORTRAN OTS, the USR is swapped into memory if it is nonresident. The FORTRAN OTS is designed so that the USR can swap over it.

If you permit the USR to swap over certain kinds of data and code, you will obtain unpredictable results. In particular, you should restrict interrupt service routines and completion routines to locations outside the USR swapping area. To find the limits of this swapping area, examine the link map and, if necessary, change the order of object modules and libraries as specified to the Linker.

Subroutines that require the USR are as follows:

CLOSEC,ICLOSE
GETSTR (only if first I/O operation on logical unit)
ICDFN (single job only)
GTLIN
ICSI
IDELET
IDSTAT
IENTER
IFETCH
IQSET
IRENAM
ITLOCK (only if USR is not in use by another job)
LOCK (only if USR is in a swapping state)
LOOKUP
PUTSTR (only if first I/O operation on logical unit)

CONTROLLING USR SWAPPING

You can control USR swapping by using the KMON commands SET USR NOSWAP and SET USR SWAP. The SET USR NOSWAP command prevents swapping and freezes the USR in memory. The command SET USR SWAP reverses this, allowing the USR to swap under program control.

Alternatively, you can compile your FORTRAN main program with the /NOSWAP option if you are sure that there is space just below the foreground partition or RMON to make the USR permanent for the duration of your program. Use this option if your program does not need the 2K words of memory that the USR occupies. If the /NOSWAP option is not specified, the USR swaps over the 2K words of your program above the base address — that is, from location 1000(octal) to 11000(octal), which is the part of a FORTRAN program least likely to violate the USR restrictions.

To prevent USR swapping for part of the program execution time and allow the USR to swap out at other times, use the LOCK, UNLOCK, and ITLOCK calls.

The LOCK call locks the USR into main memory and attaches it to the requesting job. The UNLOCK call allows the USR to swap again and to be used by another job. The ITLOCK call is used to determine whether another job is already using the USR. If so, the ITLOCK call returns immediately with an error code. This allows the program to try for a lock, but to continue with other action if it fails. The LOCK and UNLOCK calls are used in a foreground program to prevent interference from the background during initialization and completion phases and to minimize the number of swaps.

STRATEGIES IN USR SWAPPING

If you decide to change the position of code or data to avoid the USR swapping area, or if you want to move the USR itself, you must consider the concept of PSECT (program section) ordering.

PSECTs contain code and data and are identified by names as segments of the object program. The attributes associated with each PSECT direct the Linker to combine several separately compiled FORTRAN program units, assembly language modules, and library routines into an executable program.

The order in which program sections are allocated in the executable program is the order that they are presented to the Linker. Applications that are sensitive to this ordering typically separate those sections containing read-only information (such as executable code and pure data) from impure sections containing variables.

The main program unit of a FORTRAN program (normally the first object module in sequence presented to LINK) declares PSECT ordering as shown in Table 1–6.

The USR can swap over pure code, but must not be loaded over constants or impure data that can be used as arguments to the USR. The ordering

Table 1–6: FORTRAN Program PSECT Ordering

Section Name	Attributes
OTS$I	RW,I,LCL,REL,CON
OTS$P	RW,D,GBL,REL,OVR
SYS$I	RW,I,LCL,REL,CON
USER$I	RW,I,LCL,REL,CON
$CODE	RW,I,LCL,REL,CON
OTS$O	RW,I,LCL,REL,CON
SYS$O	RW,I,LCL,REL,CON
$DATAP	RW,D,LCL,REL,CON
OTS$D	RW,D,LCL,REL,CON
OTS$S	RW,D,LCL,REL,CON
SYS$S	RW,D,LCL,REL,CON
$DATA	RW,D,LCL,REL,CON
USER$D	RW,D,LCL,REL,CON
.$$$$.	RW,D,GBL,REL,OVR
Other COMMON Blocks	RW,D,GBL,REL,OVR

shown in Table 1–6 collects all pure sections before impure data in memory. The USR can safely swap over sections OTSI, OTSP, SYS$I, USER$I, and $CODE. When a FORTRAN program is running, the USR will normally swap starting at the base of section OTS$I. Location 46 of the System Communication Area contains the address where the USR will swap. If location 46 is zero, the USR will swap at its default location, below RMON and any handlers.

See the *RT–11/RSTS/E FORTRAN IV User's Guide* for more information on program sections. The *RT–11 Software Support Manual* also contains information on USR swapping and PSECT ordering.

USR LOCKOUT AND TIMING

If one job is using the USR and another job requests it, the second job will become blocked until the first job releases the USR. The second job may be locked out for seconds or minutes at a time. Interrupt service and completion routines can run, but not the mainline code. The timing problems that arise as a result can be eliminated, or minimized, in one of the following four ways:

1. Do not use devices with slow directory operations, such as cassettes and magtapes.
2. Code real-time operations as completion and interrupt service routines in your foreground job so that a locked out mainline program does not impact real-time operations.
3. Separate USR and real-time operations.
4. Use the ITLOCK call and avoid SYSLIB calls that request the USR while the USR is owned by another job.

Typically, a real-time foreground job can be constructed of (1) an initialization phase that opens all required channels and begins a real-time operation, (2) a real-time phase that performs interrupt service and I/O operations, and (3) a completion phase that halts real-time activity and then

closes the channels. Maintaining this structure in the foreground allows the background task to do USR operations during the real-time phase without locking out the foreground. This also simplifies USR swapping since the USR can swap over the interrupt routines and I/O buffers as long as they are inactive.

1.2.1.6 Subroutines Requiring Additional Queue Elements — Certain subroutines require queue elements for their proper operation. These subroutines are as follows:

IRCVD/IRCVDC/IRCVDF/IRCVDW
IREAD/IREADC/IREADF/IREADW
ISCHED
ISDAT/ISDATC/ISDATF/ISDATW
ISLEEP
ISPFN/ISPFNC/ISPFNF/ISPFNW
ITIMER
ITWAIT
IUNTIL
IWRITC/IWRITE/IWRITF/IWRITW
MRKT
MWAIT

One queue element per job is automatically allocated. Issuing more than one request from the list requires extra queue elements. Additional queue elements can be allocated through a call to the IQSET function.

1.2.1.7 System Restriction — The following restrictions must be considered when coding a FORTRAN program that uses SYSLIB.

1. Programs using IPEEK, IPOKE, IPEEKB, IPOKEB, or ISPY to access system-specific addresses, such as FORTRAN, monitor, or hardware addresses, are not guaranteed to run under future releases or on different configurations. When using these functions, you should document their use precisely so that you can check your references against the current documentation. Also, these routines may act differently under the XM monitor. NOTE: IPEEK and IPOKE are *not* equivalent to the programmed requests .PEEK and .POKE.

2. Various functions in SYSLIB return values that are of type integer, real, and double precision. If you specify an implicit statement that changes the defaults for external function types, you must explicitly declare the type of those SYSLIB functions that return integer or real results. You must also be sure that the arguments to the SYSLIB routines are the correct type for the routine. Double-precision functions must always be declared to be type DOUBLE PRECISION (or REAL*8). Failure to observe this restriction leads to unpredictable results.

3. All names of completion routines external to the routine being coded that are passed to scheduling calls (such as ISCHED, ITIMER, and IREADC) must be specified in an EXTERNAL statement in the FORTRAN program issuing the call.

4. Certain arguments to SYSLIB calls must be located in such a manner as to prohibit the RT–11 User Service Routine (USR) from swapping over them at execution time. This kind of swapping can occur when the OTS$I section (which contains the all-pure code and data for the module) is less than 2K words in length. Swapping in this uncommon situation can be avoided either by typing the SET USR NOSWAP command to make the USR resident before starting the job, or by compiling the mainline routine with a /NOSWAP option. You can also use the linker /BOUNDARY option to make OTS$O start at word boundary 11000(octal). (This problem generally occurs only with small FORTRAN programs.)

   In FORTRAN IV, program sections (PSECTs) are used to collect code and data into appropriate areas of memory. If the RT–11 USR is needed and is not resident, it swaps over a FORTRAN program starting at the symbol OTS$I for 2K words of memory.

5. Certain restrictions apply when using completion or interrupt routines. See Section 1.2.1.2 for a description of these restrictions.

6. Unless explicitly stated, null arguments should not be used in calls to SYSLIB routines.

7. If several arguments to a call are listed as being optional, they must either be all present or all omitted.

1.2.2 Calling SYSLIB Subroutines

SYSLIB includes both function subprograms and callable subroutines, which are called in the same manner as user-written subroutines.

Function subprograms receive control by means of a function reference as follows:

```
i = function name ([arguments])
```

The returned function value may be an error code, or it may be information that is useful to the calling routine. See the description of the particular function for the meaning of the returned function value.

Call subroutines are invoked by means of a CALL statement as follows:

```
CALL subroutine name [(arguments)]
```

All subroutines in SYSLIB can be called as FUNCTION programs if a return value is desired, or as SUBROUTINE programs if no return value is desired. For example, the LOCK subroutine can be referenced as either

```
CALL LOCK
```

or

```
I = LOCK()
```

Some subroutines have two acceptable formats. For example, the subroutine CLOSEC can also be specified as ICLOSE because error codes are returned by the subroutine and require an integer return to be useful.

Quoted-string literals are useful as arguments of calls to routines in SYSLIB, notably the character string routines. These literals are allowed in subroutine and function calls (see Section 1.2.7.3).

1.2.3 FORTRAN/MACRO Interface

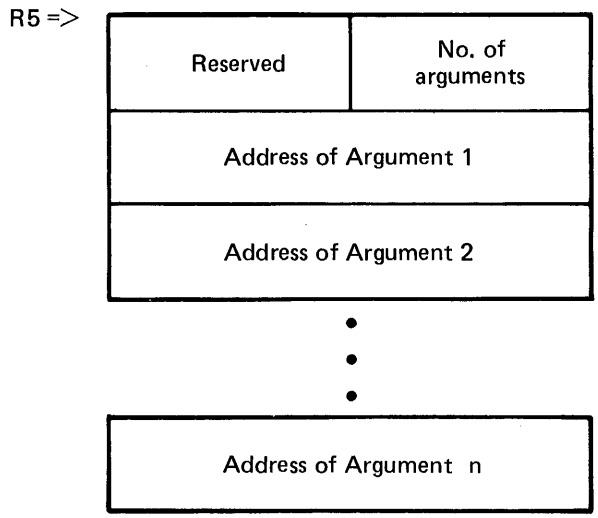
FORTRAN calling routines and subroutines follow a well-defined set of conventions regarding transfer of control, transfer of information, memory usage, and register usage. By adhering to these conventions a MACRO programmer can write FORTRAN-callable routines such as those in SYSLIB.

Control is transferred to a subroutine by

```
JSR PC,SUBR
```

When control passes to the subroutine SUBR, Register 5 (R5) points to an argument block that has the format shown in Figure 1–5.

Figure 1–5: Subroutine Argument Block



Null arguments in CALL statements must be entered with successive commas, for example, CALL SUBR (A,,B). The value –1 is stored in the argument block as the address of a null argument.

The lower byte of the first word of the argument block contains the number of arguments that are passed to the subroutine. The rest of the argument block contains the addresses of those arguments. The argument block is n+1 words long for n arguments.

The program counter is the linkage register. The subroutine obtains its arguments through R5. In FORTRAN, the calling program saves the registers, and the subroutine leaves the contents of the stack pointer intact before returning to the calling program. The RETURN statement of the subroutine is replaced by

```
RTS PC
```

The name of the subroutine must be declared global with the .GLOBL directive in the calling program or with the double colon (::) construction in the called program.

NOTE

You must make sure that the called program does not modify the argument block passed by the calling program to a subprogram.

1.2.3.1 Subroutine Register Usage — A subroutine that is called by a FORTRAN program does not have to preserve any registers. However, each push onto the stack must be matched by a pop off the stack before exiting from the routine.

User-written assembly language programs must preserve any pertinent registers before calling FORTRAN subprograms or SYSLIB routines. They must then restore registers after the subroutine returns.

Function subroutines return a single result in a register. Table 1–7 shows the register assignments for returning the different variable types.

Table 1–7: Return Value Conventions for Function Subroutines

Type	Result Placed In	
INTEGER*2 LOGICAL*1	R0	
INTEGER*4 LOGICAL*4	R0 R1	low-order result high-order result
REAL	R0 R1	high-order result (including sign and exponent) low-order result
DOUBLE PRECISION	R0 R1 R2 R3	highest-order result (including sign and exponent) next higher order next higher order lowest-order result
COMPLEX	R0 R1 R2 R3	high-order real result low-order real result high-order imaginary result low-order imaginary result

Note that floating-point results are returned in the general purpose registers and not in the FPU registers. Assembly language subprograms that use the FP11 Floating Point Unit may be required to save and restore the FPU status.

1.2.3.2 FORTRAN Programs Calling MACRO Subroutines — FORTRAN programs can call MACRO subroutines, but several rules must be followed. For example, the following program named INIARR is a MACRO subroutine that can be called from a FORTRAN program.

```
        .TITLE INIARR

        .GLOBL INIARR

;       FILENAME INIARR.MAC

INIARR: TST     (R5)+           ;SKIP ARGUMENT COUNT
        MOV     (R5)+,R2        ;PUT ADDRESS OF ARRAY INTO R2
        MOV     @(R5)+,R1       ;PUT IVAL IN R1
        MOV     @(R5)+,R0       ;AND COUNT INTO R0
        BLE     RETURN          ;QUIT IF COUNT IS NOT POSITIVE
1$:     MOV     R1,(R2)+        ;INITIALIZE ARRAY
        DEC     R0              ;DECREMENT COUNT
        BNE     1$              ;CONTINUE UNTIL ZERO
RETURN: RTS     PC
        .END
```

A FORTRAN program calls the preceding routine with

CALL INIARR (IAR,IVAL,N)

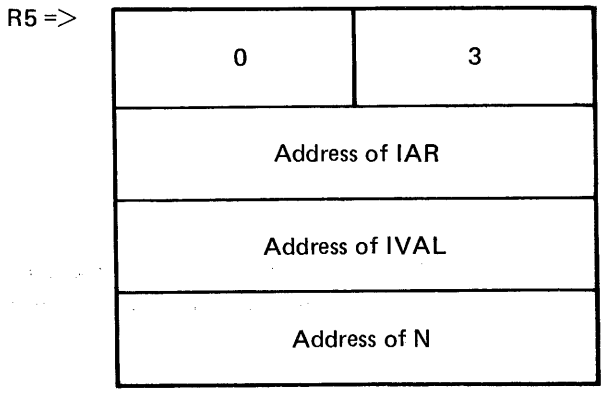
where:

INIARR	is the name of the subroutine
IAR	is the name of the array to initialize
IVAL	is the value the array is initialized to
N	is the number of elements to initialize

This program illustrates the rules that must be observed when calling a MACRO program. The name of the subroutine is made global by using the .GLOBL directive.

Register 5 (R5) is used to pass the arguments. Thus, in the program INIARR, the argument block would appear as shown in Figure 1–6.

Figure 1–6: Argument Block for Program INIARR



Registers R0 through R4 can be freely used since the calling program saves them. Once the arguments are retrieved, you can also use R5.

On completion, the subroutine returns to the calling program through an RTS PC. If your MACRO program pushes data on the stack, you must make sure that all data is popped off the stack before the RTS PC is executed.

The following FORTRAN program named DOFOR calls the subroutine INIARR.

```
        PROGRAM DOFOR
C
        INTEGER*2 ARRAY
        DIMENSION ARRAY(10)
        N=2
        DO 20 IVAL=1,10
        CALL INIARR (ARRAY,IVAL,N)
        WRITE (5,100) (ARRAY(I),I=1,N)
20      CONTINUE
100     FORMAT (I3)
        STOP
        END
```

After you compile and link both programs, run the program by typing

```
.RUN DOFOR (RET)
```

The initialized array will be output to the terminal as follows:

```
 1
 1
 2
 2
 3
 3
 4
 4
 5
 5
 6
 6
 7
 7
 8
 8
 9
 9
 10
 10
STOP --
```

1.2.3.3 MACRO Routines Calling FORTRAN Programs — If you want to call FORTRAN subroutines from a MACRO program, create a dummy main program such as

```
PROGRAM FORINT
CALL CALMAC
STOP
END
```

where CALMAC is the name of a MACRO program that can call FORTRAN or MACRO routines.

Creating a dummy program causes the FORTRAN main program to perform the initialization necessary for FORTRAN subroutines.

The following MACRO program named CALMAC calls a FORTRAN subroutine named MAXMIN.

```
        .TITLE CALMAC
        .GLOBL MAXMIN

CALMAC::
        MOV     #ARGBLK,R5      ;POINT R5 TO ARGUMENT BLOCK
        JSR     PC,MAXMIN       ;CALL MAXMIN
        RTS PC
I:      .WORD 28.               ;VALUE OF FIRST ARGUMENT
J:      .WORD 76.               ;VALUE OF SECOND ARGUMENT
ARGBLK: .WORD   2               ;NUMBER OF ARGUMENTS
        .WORD   I               ;ADDRESS OF FIRST ARGUMENT
        .WORD   J               ;ADDRESS OF SECOND ARGUMENT
        .END
```

You must set up the argument block either on the stack or in a separate area in your MACRO program. You then point R5 to the top of the argument block prior to calling the FORTRAN subroutine with a JSR PC,MAXMIN. In the above program, the argument block is set up in an area of your program.

The following program named STAKEM performs the same operation as the program CALMAC, except that it places the arguments on the stack.

```
        .TITLE STAKEM
        .GLOBL MAXMIN,STAKEM
STAKEM: MOV     #J,-(SP)
        MOV     #I,-(SP)
        MOV     #2,-(SP)
        MOV     SP,R5
        JSR     PC,MAXMIN
        ADD     #6,SP
        RTS     PC
I:      .WORD 28.
J:      .WORD 76.
        .END
```

If the argument block is set up on the stack, be sure that you remove the arguments from the stack prior to the execution of the RTS PC. In general, before calling the FORTRAN subroutine, you must save all pertinent registers. You do not know which registers the FORTRAN subroutine is using. The stack pointer remains unchanged across the call.

The name of the FORTRAN subroutine that the MACRO program calls must be defined as a global. In the FORTRAN subroutine, execute normal FORTRAN statements and return to the MACRO program with a RETURN statement.

The following program is the FORTRAN subroutine MAXMIN.

```
        SUBROUTINE MAXMIN(IN1,IN2)
        INTEGER BIG,SMALL
        IF (IN1.LT.IN2) GO TO 10
        BIG=IN1
        SMALL=IN2
        TYPE 20,BIG
        TYPE 30,SMALL
        RETURN
10      BIG=IN2
        SMALL=IN1
        TYPE 20,BIG
        TYPE 30,SMALL
20      FORMAT (' THE BIGGER NUMBER IS ',I2)
30      FORMAT (' THE SMALLER NUMBER IS ',I2)
        RETURN
        END
```

After assembling and linking the programs, using either the program CALMAC or STAKEM, type

```
.RUN FORINT RET
```

The program executes as follows:

```
THE BIGGER NUMBER IS 76
THE SMALLER NUMBER IS 28
STOP --
```

1.2.4 FORTRAN Programs in a Foreground/Background Environment

FORTRAN programs can be run in a foreground/background environment, which permits efficient use of CPU execution time. (See Chapter 15 of *Introduction to RT–11* for a description of running in an FB environment.) The basic steps in running FORTRAN programs that use the FB monitor are described in this section.

Before running your foreground program, you must use the LOAD command to load the device handlers required by the foreground job. The device handlers are placed in memory between RMON and the USR and KMON, which causes USR and KMON to move down in memory.

Next, you use the FRUN command to load your foreground program in memory between the device handlers and the USR, which causes the USR and KMON to move further down in memory. It is important that you allocate workspace when running a FORTRAN program in the foreground. You do this with the /BUFFER:n option of the FRUN command. Also make sure that any FORTRAN program you run in the foreground has adequate stack space. You can use one of the options supported by the linker (see the *RT–11 System Utilities Manual*).

The background area must be at least 4K words long to accommodate the USR and KMON. Until you run a background job with the RUN command, KMON is the background job.

When the USR is required, a 2K-word area must be set up in each job for the swapping to occur correctly — that is, there must be space for 2K words in the background area and 2K words in the foreground area. USR swapping is explained in Section 1.2.1.5.

1.2.4.1 Calculating Workspace for a FORTRAN Foreground Program — Additional workspace must be allocated in memory when running a FORTRAN program in the foreground of a foreground/background environment. For a foreground job, the space is allocated by the /BUFFER:n option of the FRUN command. (A background job uses whatever space is available between its high limit and the system's low limit.) When you allocate additional workspace in memory to run a FORTRAN program in the foreground, calculate the space required by using the following formula:

$$n = [1/2[504+(35*N)+(R-136)+A*512]]$$

where:

n = number of decimal words

A = the maximum number of files open at any one time. If double buffering is used, A should be multiplied by 2

N = the maximum number of simultaneously open channels (logical unit numbers); the default is 6

R = maximum formatted record length; the default is 136 characters

This formula must be modified for certain SYSLIB functions.

The IQSET function requires the formula to include additional space for queue elements (*qcount*) as follows:

$$n = [1/2[504+(35*N)+(R-136)+A*512]]+[10*qcount]$$

The ICDFN function requires the formula to include additional space for the integer number of channels (*num*) as follows:

$$n = [1/2[504+(35*N)+(R-136)+A*512]]+[6*num]$$

The INTSET function requires the formula to include additional space for the number of INTSET calls issued in the program as follows:

$$n = [1/2[504+(35*N)+(R-136)+A*512]]+[25*INTSET]$$

Any calls, including INTSET, that invoke completion routines must include 64(decimal) words plus the number of words needed to allocate the second record buffer (default is 68[decimal] words). The length of the record buffer is controlled by the /RECORD option to the FORTRAN compiler. If the /RECORD option is not used, the allocation in the formula must be 136(decimal) bytes, or the length that was set at FORTRAN installation time. This modifies the formula as follows:

$$n = [1/2[504+(35*N)+(R-136)+A*512]]+[64+R/2]$$

If the /BUFFER option does not allocate enough space in the foreground on the initial call to a completion routine, the following message appears:

```
?Err 0 Non-FORTRAN error call
```

This message also appears if there is not enough free memory for the background job or if a completion routine in the single-job monitor is activated during another completion routine. In the latter case, the job aborts; you should use the FB monitor to run multiple active completion routines.

1.2.4.2 Running a FORTRAN Program in a Foreground/Background Environment — This section briefly describes the procedure for running two FORTRAN programs, one in the background and one in the foreground.

The background program named BACK is as follows:

```
        PROGRAM BACKGROUND
        IMPLICIT INTEGER(O)
        CALL IPOKE("44,"10000.OR.IPEEK("44))
100     CALL PRINT('HELLO FROM THE BACKGROUND')
        ICHAR=ITTINR()
        OCHAR=ITTOUR(ICHAR)
        GO TO 100
        END
```

This program prints the message "HELLO FROM THE BACKGROUND" and will print the message each time you input a character at the terminal.

The foreground program named FORE is as follows:

```
        PROGRAM FOREGROUND
        IMPLICIT INTEGER(O)
        CALL IPOKE("44,"10000.OR.IPEEK("44))
100     CALL PRINT('HELLO FROM THE FOREGROUND')
        ICHAR=ITTINR()
        OCHAR=ITTOUR(ICHAR)
        GO TO 100
        END
```

After compiling both programs, link them. Link the foreground program using the LINK command with the /FOREGROUND option. This option produces a relocatable load module with a .REL file type. For example,

```
.LINK/FOREGROUND FORE (RET)
```

Then you can assign the device that will be used for the output of the foreground program. You must also load into memory the peripheral device handlers needed by the foreground program.

The command FRUN loads and starts execution of a .REL program as the foreground job. If the command

```
.FRUN FORE (RET)
```

is typed at this point, the error message

```
?Err 62 FORTRAN start fail
```

will be displayed. This message indicates that additional workspace allocation is required and that the /BUFFER option must be used. (Refer to the previous section for the formula to calculate the additional space needed.) Thus, the command would be typed as follows:

```
.FRUN FORE/BUFFER:760 RET
```

Execution of this command results in the following output at the terminal:

```
F>
HELLO FROM THE FOREGROUND

B>
.
```

The system first identifies the message as foreground output. Then the foreground job executes and outputs its message. The background monitor next prints the characters B> and a period, indicating that control has returned to monitor command mode. Command input remains directed to the background job. By typing

```
.RUN BACK RET
```

the message from the background job will be displayed

```
HELLO FROM THE BACKGROUND
```

Each time a character is input to the terminal, say an "L", the message will be repeated.

```
LHELLO FROM THE BACKGROUND
```

Use the CTRL/F command to direct terminal input to the foreground job. The system prints F> to remind you that you are now directing input to the foreground job. When you type a character, such as "Y", the foreground job message will be displayed.

```
F>
YHELLO FROM THE FOREGROUND
```

Type a CTRL/B to return to the background job or a CTRL/C to return to monitor command mode. If you are returning to a background environment, you should unload the foreground job and any handlers to reclaim memory space for background use.

1.2.5 Linking with FORLIB

Normally, the default system library file (SYSLIB.OBJ) also includes the overlay handlers and the appropriate FORTRAN run-time system routines.

To add FORLIB.OBJ modules to the default library SYSLIB.OBJ, use the following command:

```
.LIBRARY/INSERT/REMOVE SYSLIB FORLIB RET
Global?  $OVRH RET
Global?  RET
```

1.2.6 SYSLIB Services Not Provided by Programmed Requests

SYSLIB provides many services that are not provided by programmed requests. Such services are as follows:

- Time conversion and date access
- Program suspension
- Two-word integer support (INTEGER*4)
- Radix–50 conversion
- Character string manipulation

1.2.6.1 Time Conversion and Date Access — Several calls allow you to perform time conversions and access the system date.

You use the CVTTIM call to convert a two-word internal format time to hours, minutes, seconds, and ticks. The JTIME call converts a time given in hours, minutes, seconds, and ticks into the internal two-word time format.

If you need to print out the time, the TIMASC call converts the time returned by the .GTIM programmed request into an eight-character ASCII string; the TIME call returns the current time of day as an eight-character ASCII string.

The current system date can be accessed by your program with a DATE call. The date is returned as a string value. IDATE performs similarly, but returns an integer value. DATE and IDATE are part of FORLIB.OBJ.

1.2.6.2 Program Suspension — You suspend execution of a running program for a specified number of ticks with the ITWAIT call. You use the ISLEEP call to suspend a running program for a specified number of hours, minutes, seconds, and ticks. The IUNTIL call allows you to suspend job execution until a specific time of day, which is given to the routine in hours, minutes, seconds, and ticks. You can use this function to periodically collect data and to stop processing between acquisitions.

1.2.6.3 Two-Word Integer Support (INTEGER*4) — You can make calls to SYSLIB to manipulate a 32-bit integer that uses two words of storage. The first word contains the low-order part of the value and the second word contains the sign and the high-order part of the value. The range of numbers that is represented is –2(31) to 2(31)–1. This format differs from the two-word internal time format that stores the high-order part of the value in the first word and the low-order part in the second word. Table 1–8 shows the calls that you can use to convert from one format to another.

Calls are also available for you to perform arithmetic operations on INTEGER*4 values, move a value to a variable, and convert a two-word internal time format to and from an INTEGER*4 value.

Table 1–8: SYSLIB Conversion Calls

From	To	Call
INTEGER*2 (16-bit integer)	INTEGER*4	JICVT
INTEGER*4 (32-bit integer)	INTEGER*2	IJCVT
INTEGER*4	REAL*4	AJFLT//IAJFLT
INTEGER*4	REAL*8	DJFLT/IDJFLT
REAL*4 (2-word floating point)	INTEGER*2	JAFIX
REAL*8 (4-word floating point)	INTEGER*4	JDFIX

1.2.6.4 Radix–50 Conversion — You can convert ASCII characters to or from Radix–50.

IRAD50 converts a specified number of characters of Radix–50 and returns the number of characters converted as a function result. RAD50 encodes RT–11 file descriptors in Radix–50 notation. R50ASC converts a specified number of Radix–50 characters to ASCII.

1.2.6.5 Character String Operations — SYSLIB provides character string functions that perform string operations such as concatenation, comparison, copying, replacing, and computing the number of characters in a string. For example, the following program will concatenate two character strings.

```
        .TITLE GETTOO
        .GLOBL CONCAT
        .MCALL .PRINT,.EXIT

START:  MOV     #ARGBLK,R5
        JSR     PC,CONCAT
        .PRINT  #STRCON
        .EXIT
ARGBLK: .WORD 3
        .WORD STRNG1
        .WORD STRNG2
        .WORD STRCON
STRNG1: .ASCIZ /RESEARCH AND/
STRNG2: .ASCIZ / DEVELOPMENT/
STRCON: .BLKB 31
        .EVEN
        .END START
```

Running this program results in the concatenation of string 1 and string 2, and the output at the terminal is

```
RESEARCH AND DEVELOPMENT
```

The following section describes character string functions in detail.

1.2.7 Character String Functions

The SYSLIB character string functions and routines provide variable-length string support for RT–11 FORTRAN and for MACRO programs. SYSLIB calls perform the following character string operations:

Call	Operation
GETSTR	Reads character strings from a specified FORTRAN logical unit
PUTSTR	Writes character strings to a specified FORTRAN logical unit
CONCAT	Concatenates variable-length strings
INDEX	Returns the position of one string in another
INSERT	Inserts one string into another
LEN	Returns the length of a string
REPEAT	Repeats a character string
SCOMP	Compares two strings
SCOPY	Copies a character string
STRPAD	Pads a string with blanks on the right
SUBSTR	Copies a substring from a string
TRANSL	Performs character modification
TRIM	Removes trailing blanks
VERIFY	Verifies the presence of characters in a string

Strings are stored in LOGICAL*1 arrays that you define and dimension. These arrays store strings in ASCII format as one character per array element plus a zero element to indicate the current end of the string.

The length of a string can vary at execution time from zero characters to one less than the size of the array that stores the string. The maximum size of any string is 32767 characters. Strings can contain any of the seven-bit ASCII characters except null(0), since the null character is used to mark the end of the string. The inclusion of a terminating zero byte constitutes an "ASCIZ" format, which is the format set up by a MACRO assembler directive .ASCIZ. This directive automatically sets up strings with a terminating zero byte. Bit 7 of each character must be cleared. Therefore, the valid characters are those whose decimal representations range from 1 to 127, inclusive.

The ASCII code used in this string package is the same as that employed by FORTRAN for A-type FORMAT items, ENCODE/DECODE strings, and object-time format strings. Whenever quoted strings are used as arguments in the CALL statement, ASCIZ strings are generated for these routines by the FORTRAN compiler. Note that a null string (a string containing no characters) can be represented in FORTRAN by a variable or constant of any type that contains the value zero, or by a LOGICAL variable or constant with the .FALSE. value.

In many routines, it is difficult to predict the length of the string produced. To prevent a string from overflowing the array that contains it, you can

specify an optional integer argument to the subroutine. This argument, called *len*, limits the length of an output string to the value specified for *len* plus one (for the null terminator), so that the array receiving the result must be at least *len* plus one elements in size.

NOTE

If the string is larger than the array, other data may be destroyed and cause unpredictable results.

When *len* is specified, you can also include the optional argument called *err*. *Err* is a logical variable that should be initialized by the FORTRAN program to the .FALSE. value. If a string function is given the arguments *len* and *err*, and *len* is actually used to limit the length of the string result, then *err* is set to the .TRUE. value. If *len* is not used to truncate the string, *err* is unchanged — that is, it retains a .FALSE. value.

The argument *len* can appear alone. However, *len* must appear if *err* is specified. The *err* argument should be used for GETSTR and PUTSTR.

Several routines use the concept of character position. Each character in a string is assigned a position number. The first character in a string is in position one. Each subsequent character has a position number one greater than the character that precedes it.

1.2.7.1 Allocating Character String Variables — A one-dimensional LOGICAL*1 array can contain a single string whose length can vary from zero characters to one fewer than the dimensioned length of the array. For example,

```
LOGICAL*1 A(45)          !ALLOCATE SPACE FOR STRING VARIABLE A
```

allows array A to be used as a string variable that can contain a string of 44 or fewer characters. Similarly, a two-dimensional LOGICAL*1 array can be used to contain a one-dimensional array of strings. Each string in the array can have a length up to one less than the first dimension of the LOGICAL*1 array. There can be as many strings as the number specified for the second dimension of the LOGICAL*1 array. For example,

```
LOGICAL*1 W(21,10)       !ALLOCATE AN ARRAY OF STRINGS
```

creates string array W that has ten string elements, each of which can contain up to 20 characters. String I in array W is referenced in subroutine or function calls as W(1,I).

The following example allocates a two-dimensional string array.

```
LOGICAL*1 T(14,5,7)      !ALLOCATE A 5 BY 7 ARRAY OF 13-CHARACTER
                         !STRINGS
```

Each string in array T may vary in length to a maximum of 13 characters. String I,J of the array can be referenced as T(1,I,J). Note that T is the same as T(1,1,1). This dimensioning process can create string arrays of up to six dimensions (represented by LOGICAL*1 arrays of up to seven dimensions).

1.2.7.2 Passing Strings to Subprograms — There are three ways to pass strings to subprograms.

1. LOGICAL*1 arrays that contain strings can be placed in a COMMON block and referenced by any or all routines with a similar common declaration. However, when you place a LOGICAL*1 array in a common block, make sure that the array is even in length, that odd-length arrays are paired to result in an overall even length, or that the strings are together as the last elements in the COMMON block. Otherwise, all succeeding variables in the COMMON block may be assigned odd addresses.

   A LOGICAL*1 array has an odd length only if the product of its dimensions is odd. For example,

   ```
   LOGICAL*1 B(10,7)      !(10*7) = 70; EVEN LENGTH
   LOGICAL*1 H (21)       !21 IS AN ODD LENGTH
   ```

   These might be handled as follows:

   ```
   COMMON A1,A2,A3(10),H(21) !PLACE ODD-SIZED ARRAY AT END
   ```

   or

   ```
   COMMON A1,A2,H(21),H1(7),A3(10) !PAIR ODD-SIZE ARRAYS H AND H1
   ```

   These restrictions apply only to LOGICAL*1 variables and arrays.

2. A single string can be passed by using its array name as an argument. For example,

   ```
   LOGICAL*1 A(21) !STRING VARIABLE A, 20 CHARACTERS MAXIMUM
   CALL SUBR(A)
   ```

   passes string A to subroutine SUBR.

3. If the calling program has declared a multidimensional array, and only one string of that array is to be passed to a subroutine, then the subroutine call should specify the first element of the string to be passed (this requires that the first dimension of the array equals the maximum length of each string).

   For example,

   ```
      LOGICAL*1 NAMES (81,20) !20 NAMES, 80 CHARACTERS EACH
      LOGICAL*1 ERR
      .
      .
      .
      DO 10 NAMNUM=1,20 !GET ALL 20 NAMES
   10 CALL GETSTR (5,NAMES(1,NAMNUM),80,ERR) !FROM TT
   ```

If the maximum length of a string argument is unknown in a subroutine or function, or if the routine is used to handle many different lengths, the dummy argument in the routine should be declared as a LOGICAL*1 array with a dimension of one, such as LOGICAL*1 ARG(1). In this case, the string routines correctly determine the length of ARG whenever it is used,

but it is not possible to determine the maximum size of any string that can be stored in ARG. If a multidimensional array of strings is passed to a routine, it must be declared in the called program with the same dimensions that were specified in the calling program.

NOTE

The length argument specified in many of the character string functions refers to the maximum length of the string excluding the necessary null byte terminator. The length of the LOGICAL*1 array to receive the string must be at least one greater than the length argument.

1.2.7.3 Using Quoted-String Literals — You can use quoted strings as arguments to any of the string routines that are invoked as functions or with the CALL statement. For example,

```
CALL SCOMP(NAME,'SMYTHE, R',M)
```

compares the string in the array NAME to the constant string SMYTHE, R and sets the value of the integer variable accordingly.

1.2.8 System Subroutine Summary

Table 1–9 lists the SYSLIB subroutines alphabetically within categories, the sections in which they are located, and a brief description of each subroutine. Those subroutines prefaced with an asterisk (*) are allowed only in a foreground/background environment, under either the FB or XM monitor. The SYSLIB subroutines do not support the XM monitor mapping programmed requests. Use FORTRAN virtual arrays to access extended memory.

Table 1–9: Summary of SYSLIB Subroutines

Name	Section	Description
File-Oriented Operations		
CLOSEC, ICLOSE	3.3	Closes the specified channel.
IDELET	3.22	Deletes a file from the specified device.
IENTER	3.25	Creates a new file for output.
IFPROT	3.27	Changes the file's protection.
IRENAM	3.46	Changes the name of the indicated file.
ISFDAT	3.53	Changes the file's creation date.
LOOKUP	3.79	Opens an existing file for input and/or output via the specified channel.

(continued on next page)

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
Data Transfer Operations		
IABTIO	3.12	Aborts I/O operations on a specified channel.
GTLIN	3.11	Transfers a line of input from the console terminal or indirect file (if active) to the user program.
*IRCVD *IRCVDC *IRCVDF *IRCVDW	3.44	Receives data. Allows a job to read messages or data sent by another job in an FB environment. The four modes correspond to the IREAD, IREADC, IREADF, and IREADW modes.
IREAD	3.45	Transfers data from a file to a memory buffer and returns control to the user program when the request is entered in the I/O queue. No special action is taken upon completion of I/O.
IREADC	3.45	Transfers data from a file to a memory buffer and returns control to the user program when the request is entered in the I/O queue. Upon completion of the read, control transfers to the assembly language routine specified in the IREADC function call.
IREADF	3.45	Transfers data from a file to a memory buffer and returns control to the user program when the request is entered in the I/O queue. Upon completion of the read, control transfers to the FORTRAN subroutine specified in the IREADF function call.
IREADW	3.45	Transfers data from a file to a memory buffer and returns control to the program only after the transfer is complete.
*ISDAT *ISDATC *ISDATF *ISDATW	3.51	Allows the user to send messages or data to the other job in an FB environment. The four functions correspond to the IWRITE, IWRITC, IWRITF, and IWRITW modes.
ITTINR	3.59	Gets one character from the console keyboard.
ITTOUR	3.60	Transfers one character to the console terminal.
IWAIT	3.64	Waits for completion of all I/O on a specified channel (commonly used with the IREAD and IWRITE functions).
IWRITC	3.65	Transfers data to a file and returns control to the user program when the request is entered in the I/O queue. Upon completion of the write, control transfers to the assembly language routine specified in the IWRITC function call.
IWRITE	3.65	Transfers data to a file and returns control to the user program when the request is entered in the I/O queue. No special action is taken upon completion of the I/O.
IWRITF	3.65	Transfers data to a file and returns control to the user program when the request is entered in the I/O queue. Upon completion of the write, control transfers to the FORTRAN subroutine specified in the IWRITF function call.
IWRITW	3.65	Transfers data to a file and returns control to the user program only after the transfer is complete.

* FB and XM monitors only.

(continued on next page)

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
†MTATCH	3.81	Attaches a particular terminal in a multiterminal environment.
†MTDTCH	3.82	Detaches a particular terminal in a multiterminal environment.
†MTGET	3.83	Provides information about a particular terminal in a multiterminal system.
†MTIN	3.84	Transfers characters from a specific terminal to the user program in a multiterminal system.
†MTOUT	3.85	Transfers characters to a specific terminal in a multiterminal system.
†MTPRNT	3.86	Prints a message to a specific terminal in a multiterminal system.
†MTRCTO	3.87	Enables output to terminal by canceling the effect of a previously typed CTRL/O.
†MTSET	3.88	Sets terminal and line characteristics in a multiterminal system.
†MTSTAT	3.89	Returns multiterminal system status.
*MWAIT	3.90	Waits for messages to be processed.
PRINT	3.91	Outputs an ASCII string to the console terminal.
Channel-Oriented Operations		
ICDFN	3.16	Defines additional I/O channels.
*ICHCPY	3.17	Allows access to files currently open in another job's environment.
ICSTAT	3.21	Returns the status of a specified channel.
IFREEC	3.28	Returns the specified RT–11 channel to the available pool of channels for the FORTRAN I/O system.
IGETC	3.29	Allocates an RT–11 channel and informs the FORTRAN I/O system of its use.
ILUN	3.33	Returns the RT–11 channel number with which a FORTRAN logical unit is associated.
IREOPN	3.47	Restores the parameters stored via an ISAVES function and reopens the channel for I/O.
ISAVES	3.48	Stores five words of channel status information into a user-specified array and deactivates the channel.
PURGE	3.92	Deactivates a channel.
Device and File Specifications		
IASIGN	3.15	Sets information in the FORTRAN logical unit table.
ICSI	3.20	Calls the RT–11 CSI in special mode to decode file specifications and options.

† With multiterminal support only.

(continued on next page)

* FB and XM monitors only.

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
Timer Support Operations		
CVTTIM	3.5	Converts a two-word internal format time to hours, minutes, seconds, and ticks.
GTIM	3.9	Gets time of day.
ICMKT	3.19	Cancels an unexpired ISCHED, ITIMER, or MRKT request (valid under FB and XM, and for SJ monitors with timer support, a SYSGEN option).
ISCHED	3.49	Schedules the specified FORTRAN subroutine to be entered at the specified time of day as an asynchronous completion routine (valid under FB and XM, and for SJ monitors with timer support, a special feature).
ISDTTM	3.52	Changes the system date and time.
‡ISLEEP	3.54	Suspends main program execution of the running job for a specified amount of time; completion routines continue to run.
ITIMER	3.57	Schedules the specified FORTRAN subroutine to be entered as an asynchronous completion routine when the time interval specified has elapsed (valid under FB and XM, and for SJ monitors with timer support, a special feature).
‡ITWAIT	3.61	Suspends the running job for a specified amount of time; completion routines continue to run.
‡IUNTIL	3.62	Suspends the main program execution of the running job until a specified time of day; completion routines continue to run.
JTIME	3.76	Converts hours, minutes, seconds, and ticks into two-word internal format time.
MRKT	3.80	Schedules an assembly language routine to be activated as an asynchronous completion routine after a specified interval (valid under FB and XM, and for SJ monitors with timer support, a special feature).
SECNDS	3.103	Returns the current system time in seconds past midnight minus the value of a specified argument.
TIMASC	3.108	Converts a specified two-word internal format time into an eight-character ASCII string.
TIME	3.109	Returns the current system time of day as an eight-character ASCII string.
RT–11 Services		
CHAIN	3.2	Chains to another program (from the background job only).
*DEVICE	3.6	Specifies actions to be taken on normal or abnormal program termination, such as turning off interrupt enable on user-programmed devices.
GTJB,IGTJB	3.10	Returns the parameters of the specified job.
IDSTAT	3.24	Returns the status of the speciffied device.

‡ SYSGEN option in SJ monitor.
* FB and XM monitors only.

(continued on next page)

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
IFETCH	3.26	Loads a device handler into memory.
IQSET	3.42	Expands the size of the RT–11 monitor queue from the free space managed by the FORTRAN system.
ISPFN ISPFCN ISPFNF ISPFNW	3.55	Issues special function requests to various handlers, such as magtape. The four modes correspond to the IWRITE, IWRITC, IWRITF, and IWRITW modes.
*ITLOCK	3.58	Indicates whether the USR is currently in use by another job and performs a LOCK if the USR is available.
LOCK	3.78	Makes the RT–11 monitor User Service Routine (USR) permanently resident until an UNLOCK function is executed. If necessary, a portion of the user's program is swapped out to make room for the USR.
RCHAIN	3.96	Allows a program to access variables passed across a chain.
RCTRLO	3.97	Enables output to the terminal by canceling the effect of a previously typed CTRL/O.
*RESUME	3.99	Causes the main program execution of a job to resume at the point it was suspended by a SUSPND function call.
SCCA	3.100	Intercepts a CTRL/C command initiated at the console terminal.
SETCMD	3.104	Passes command lines to the keyboard monitor for execution after the program exits.
*SUSPND	3.107	Suspends main program execution of the running job; completion routines continue to execute.
UNLOCK	3.112	Releases the USR if a LOCK was performed; the user program is swapped in if required.
INTEGER*4 Support Functions		
AJFLT	3.1	Converts a specified INTEGER*4 value to REAL*4 and returns the result as the function value.
DJFLT	3.7	Converts a specified INTEGER*4 value to REAL*8 and returns the result as the function value.
IAJFLT	3.14	Converts a specified INTEGER*4 value to REAL*4 and stores the result.
IDJFLT	3.23	Converts a specified INTEGER*4 value to REAL*8 and stores the result.
IJCVT	3.32	Converts a specified INTEGER*4 value to INTEGER*2.
JADD	3.66	Computes the sum of two INTEGER*4 values.
JAFIX	3.67	Converts a REAL*4 value to INTEGER*4.
JCMP	3.68	Compares two INTEGER*4 values and returns an INTEGER*2 value that reflects the signed comparison result.
JDFIX	3.69	Converts a REAL*8 value to INTEGER*4.

* FB and XM monitors only.

(continued on next page)

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
JDIV	3.70	Computes the quotient and remainder of two INTEGER*4 values.
JICVT	3.71	Converts an INTEGER*2 value to INTEGER*4.
JJCVT	3.72	Converts the two-word internal time format to INTEGER*4 format, and vice versa.
JMOV	3.73	Assigns an INTEGER*4 value to a variable.
JMUL	3.74	Computes the product of two INTEGER*4 values.
JSUB	3.75	Computes the difference between two INTEGER*4 values.
Character String Functions		
CONCAT	3.4	Concatenates two variable-length strings.
GETSTR	3.8	Reads a character string from a specified FORTRAN logical unit.
INDEX	3.34	Returns the location in one string of the first occurrence of another string
INSERT	3.35	Replaces a portion of one string with another string.
ISCOMP	3.50	Compares two character strings.
IVERIF	3.63	Indicates whether characters in one string appear in another.
LEN	3.77	Returns the number of characters in a specified string.
PUTSTR	3.93	Writes a variable-length character string on a specified FORTRAN logical unit.
REPEAT	3.98	Concatenates a specified string with itself to provide an indicated number of copies and stores the resultant string.
SCOMP	3.101	Compares two character strings.
SCOPY	3.102	Copies a character string from one array to another.
STRPAD	3.105	Pads a variable-length string on the right with blanks to create a new string of a specified length.
SUBSTR	3.106	Copies a substring from a specified string.
TRANSL	3.110	Replaces one string with another after performing character modification.
TRIM	3.111	Removes trailing blanks from a character string.
VERIFY	3.113	Indicates whether characters in one string appear in another.

(continued on next page)

Table 1–9: Summary of SYSLIB Subroutines (Cont.)

Name	Section	Description
Radix–50 Conversion Operations		
IRAD50	3.43	Converts characters in ASCII format to Radix–50, returning the number of characters converted.
R50ASC	3.94	Converts characters in Radix–50 format to ASCII.
RAD50	3.95	Converts six ASCII characters, returning a REAL*4 result that is the two-word Radix–50 value.
Miscellaneous Services		
IADDR	3.13	Obtains the memory address of a specified entity.
IGETSP	3.30	Returns the address and size (in words) of free space obtained from the FORTRAN system.
INTSET	3.36	Establishes a specified FORTRAN subroutine as an interrupt service routine with a specified priority.
IPEEK	3.37	Returns the value of a word located at a specified absolute memory address.
IPEEKB	3.38	Returns the value of a byte located at a specified byte address.
IPOKE	3.39	Stores an integer value in an absolute memory location.
IPOKEB	3.40	Stores an integer value in a specified byte location.
IPUT	3.41	Changes the value of the word located at an offset specified from the beginning of the RT–11 monitor.
ISPY	3.56	Returns the integer value of the word located at a specified offset from the beginning of the RT–11 resident monitor.

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	a pointer to the EMT argument block for those requests that require a block.
blk	a block number specifying the relative block in a file or device where an I/O transfer is to begin.
buf	a buffer address specifying a memory location into which or from which an I/O transfer will be performed; this address has to be word-aligned — that is, located at an even address and not a byte or odd address.
cblk	the address of the five-word block where channel status information is stored.
chan	a channel number in the range 0–376(octal).
chrcnt	a character count in the range 1–255(decimal).
code	a flag used to indicate whether the code is to be set in an EMT 375 programmed request.
crtn	the entry point of a completion routine.
dblk	a four-word Radix–50 descriptor block that specifies the physical device, file name, and file type to be operated upon (see Section 1.1.2.6).
func	a numerical code indicating the function to be performed.
jobblk	a pointer to a three-word ASCII system job name.
jobdev	a 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	a number, the value of which depends on the request.

seqnum — a file number.

For cassette operation, a value of 0 is assumed if this argument is blank.

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 — the logical unit number of a particular terminal in a multiterminal system.

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

Many programmed requests are qualified as special features. These requests are enabled only if you performed a system generation process, that is, they are not available in a distributed monitor.

2.1 .ABTIO

The .ABTIO programmed request allows a running job to stop 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.

This request cannot be issued from a completion routine.

Macro Call: .ABTIO chan

where:

chan — is the channel number for which to abort I/O

Request Format:

R0 = | 13 | chan |

Errors:

none

Example:

```
          .TITLE  ABTIO.MAC

;This is an example of the .ABTIO request. The .ABTIO request
;is useful for immediately 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, .SCCA

START:    .SCCA       #AREA,#CTCWRD          ;Inhibit CTRL/C
          .ENTER      #AREA,#1,#FILNAM       ;Open channel 1 as output file
IOLOOP:     .
            .                                ;Perform I/O to the file...
            .
```

```
        TST     CTCWRD          ;Has CTRL/C been typed?
        BPL     IOLOOP          ;No, continue file I/O
        .ABTIO  #1              ;Yes, stop I/O on channel 1
          .
          .                     ;Continue other processing
          .

AREA:   .BLKW   4               ;EMT argument block
CTCWRD: .WORD   0               ;Terminal status word

        .END
```

2.2 .CDFN

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

The space for the new channels is taken from within 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.

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 20(decimal) channels (while 16 original channels are defined) creates 20 new I/O channels; the space for the original 16 is unused, but the contents of the old channel set are copied to the new channel set.

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

If an XM 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(octal). 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:

R0 → area:

15	0
addr	
num	

Errors:

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

Example:

```
          .TITLE  CDFN.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

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

AREA:     .BLKW       3                       ;EMT Argument Block
CHANL:    .BLKW       5*32.                   ;Space for new channels

BADCD:    .ASCIZ      /? .CDFN Failed ?/      ;Failure message
GOODCD:   .ASCIZ      /.CDFN Successful/      ;Success message

          .END        START
```

2.3 .CHAIN

The .CHAIN request allows a background program to 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 area in low memory from locations 500–507 contains the device name and file name (in Radix–50) to be chained to. The area from locations 510–777 is used to pass information between the chained programs.

Macro Call: .CHAIN

Request Format:

R0 =

10	0

Notes:

1. 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 a .CHAIN to or from a virtual job, locations 500–777 are not preserved.

2. I/O channels are left open across a .CHAIN for use by the new program. However, new I/O channels opened with a .CDFN request are not available in this way. Since the monitor reverts to the original 16 channels during a .CHAIN, programs that leave files open across a .CHAIN should not use .CDFN. Furthermore, 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.

3. 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. Again, locations 500–777 are not preserved in a .CHAIN to or from a virtual job.

   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, the .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(octal); if caution is not observed, the stack can destroy chain data before it can be used.

Example:

```
        .TITLE  CHAIN.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,.TTYIN,.PRINT

START:  MOV         #500,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                           ;
        .PRINT      #PROMT              ;Ask for the data to be passed
```

```
LOOP:   .TTYIN                          ;Now get a "command" line
        MOVB    R0,(R1)+                ;to pass to the chained program
        CMPB    R0,#12                  ;in locations 510 and up.
        BNE     LOOP                    ;Loop until line feed.
        CLRB    @R1                     ;Put in a null byte as a terminator.
        .CHAIN                          ;Chain to the next program.
CHPTR:  .RAD50  /DK/                    ;RAD50 File spec...
        .RAD50  /CTEST /
        .RAD50  /SAV/
PROMT:  .ASCII  /Enter data to be passed to CTEST > /<200>
        .END    START


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

        .TITLE  CTEST.MAC

        .MCALL  .PRINT,.EXIT

        JSW = 44                        ;Location of JSW
        CHAIN$ = 400                    ;CHAIN bit in JSW

CTEST:  BIT     #CHAIN$,@#JSW           ;Were we chained to?
        BEQ     1$                      ;Branch if not
        .PRINT  #CHAIND                 ;Say we were...
        MOV     #510,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  /CTEST was chained to - and here's the data passed.../
NOCHN:  .ASCIZ  /CTEST was not chained to/
        .END    CTEST
```

2.4 .CHCOPY (FB and XM Only)

The .CHCOPY request opens a channel for input, logically connecting it to a file that is currently open by another job for either input or output. 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 (including diskette) or DECtape. However, no errors are detected by the system if another device is used. (To close a channel following use of .CHCOPY, use either the .CLOSE or .PURGE request.)

Macro Call: .CHCOPY area,chan,ochan [,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 (see the *RT–11 System Utilities Manual*)

Request Format:

R0 → area:

13	chan
ochan	
jobblk	

Notes:

1. If the other job's channel was opened with .ENTER in order to create a file, the copier's channel indicates a file that extends to the highest block that the creator of the file had written at the time the .CHCOPY was executed.
2. A channel open on a non-file-structured device should not be copied, because intermixture of buffer requests can result.
3. A program can write to a file (that is being created by the other job) on a copied channel just as it could if it were the creator. When the copier's channel is closed, however, no directory update takes place.
4. Foreground and background jobs may optionally 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:

```
;+
; .CHCOPY - This is an example in the use of the .CHCOPY request.
; 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.
;-
          .TITLE  CHCOPF.MAC
;+
; 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 QUFILE/          ;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     /?Enter Error?/      ;Error message text
FEXIT:    .ASCIZ     /FG Job exiting/     ;Exit message
          .END       STARTF

          .TITLE  CHCOPB.MAC
;+
; 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        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
          .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
          .EXIT                           ;then exit.
AREA:     .BLKW      5                    ;EMT argument blk
MSG:      .BLKW      3                    ;Message buffer
BUFF:     .BLKW      256.                 ;File buffer
BEXIT:    .ASCIZ     /Channel-Record copy successful/
NOJOB:    .ASCIZ     /?No FG Job?/        ;Error messages...
NOCH:     .ASCIZ     /?FG channel not open?/
RDERR:    .ASCIZ     /?Read Error?/
          .END       STARTB
```

2.5 .CLOSE

The .CLOSE request terminates activity on the specified channel and frees it for use in another operation. The handler for the associated device must be in memory if the file was created with a .ENTER programmed request.

Macro Call: .CLOSE chan

Request Format:

R0 = | 6 | chan |

A .CLOSE request specifying a channel that is not open is ignored.

A file opened with .LOOKUP does not require any directory operations when a .CLOSE is issued, and the USR does not have to be in memory for such a .CLOSE. The 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.

A .CLOSE is required on any channel opened with .ENTER if the associated file is to become permanent.

NOTE

Do not close channel 17(octal) 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.

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

.CLOSE does not return any other errors unless the .SERR request has been issued. If the device handler for the operation is not in memory, and the .CLOSE request requires updating of the device directory, a fatal monitor error is generated.

Example:

Refer to the examples for the .CSISPC and .WRITW requests, which show typical uses for .CLOSE.

2.6 .CMKT (FB and XM. SJ Monitor Special Feature)

The .CMKT request causes one or more outstanding mark time requests to be canceled (see the .MRKT programmed request). The .CMKT request is a special feature in the SJ monitor, and is selected with the timer support 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:

1. Canceling a mark time request frees the associated queue element.
2. 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.
3. 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.

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.

2.7 .CNTXSW (FB and XM Only)

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 when jobs are switched. Refer to the *RT–11 Software Support Manual* for further details.

The system always saves the parameters it needs to uniquely identify and execute a job. These parameters include all registers and the following locations:

34,36	Vector for TRAP instruction
40–52	System Communication Area

If an .SFPA request has been executed with a non-zero address, all floating-point registers and the floating-point status are also saved.

It is possible that both jobs want to share the use of a particular location not included in normal context switch operations. For example, if a pro-

gram 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:

a. in the range 2–476
b. in the user job area
c. in the I/O page (addresses 160000–177776)

Request Format:

R0 → area:

33	0
addr	

Errors:

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

Example:

```
          .TITLE  CNTXSW.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 20
; and 22 (IOT vectors) and certain necessary EAE registers be context
; switched. This allows both jobs to use IOT and the EAE simultaneously
; yet independently.
;-
          .MCALL      .CNTXSW,.PRINT,.EXIT

START:    .CNTXSW     #AREA,#SWLIST          ;Issue the .CNTXSW request
          BCC         1$                     ;Branch if successful
          .PRINT      #ADDERR                ;Address error (should not occur)
          .EXIT                              ;Exit the program
1$:       .PRINT      #CNTOK                 ;Acknowledge success with a message
          .EXIT                              ;then exit the program

SWLIST:   .WORD       20                     ;Addresses to include in context switch
          .WORD       22                     ;IOT & EAE vectors...
          .WORD       177302                 ;EAE registers...
          .WORD       177304                 ;
          .WORD       177310                 ;
          .WORD       0                      ;List terminator !!!

AREA:     .BLKW       2                      ;EMT argument block

ADDERR:   .ASCIZ      /? .CNTXSW Addressing Error ?/
CNTOK:    .ASCIZ      /.CNTXSW Successful/

          .END        START
```

2.8 .CRAW (XM Only)

The .CRAW request 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 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.VNM set if one or more windows were unmapped to create and map this window
WS.ELW set if one or more windows were eliminated

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. You should eliminate a window first (.ELAW), or redefine your virtual address space into fewer windows.

If the WS.MAP bit was 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.

Example:

```
          .TITLE  XMCOPY
;+
; 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
          JSW      =   44                  ;JSW location
          J.VIRT   =   2000                ;Virtual Job bit in JSW
          ERRBYT   =   52                  ;Error byte location
          APR      =   2                   ;PAR/PDR for 1st window
          APR1     =   4                   ;    "   "  2nd   "
          BUF      =   WDB+W.NBAS          ;Virtual addr of 1st buffer
          BUF1     =   WDB1+W.NBAS         ;   "    "   "  2nd   "
          CORSIZ   =   4096.               ;Size of buffer in words
          PAGSIZ   =   CORSIZ/256.         ;Page size in blocks
          WRNID    =   WDB+W.NRID          ;Region ID addr of 1st region
          WRNID1   =   WDB1+W.NRID         ;   "    "   "   "  2nd   "

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

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

START::   .CSIGEN  #ENDCRE,#DEFLT,#0       ;Get filespecs, handlers, open files
          BCS      START                   ;Branch if error
```

```
        INCB    ERRNO                   ;ERR = 1x
        .CRRG   #CAREA,#RDB             ;Create a region
        BCC     10$                     ;Branch if successful
        JMP     ERROR                   ;Report error (JMP due to range!)
10$:    MOV     RDB,WRNID               ;Move region id to Window Def Blk
        INCB    ERRNO                   ;ERR = 2x
        .CRAW   #CAREA,#WDB             ;Create window...
        BCC     20$                     ;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$:    CLR     R1                      ;R1 = RT11 Block # for I/O
        MOV     #CORSIZ,R2              ;R2 = # of words to read
        INCB    ERRNO                   ;ERR = 4x
READ:   .READW  #RAREA,#3,BUF,R2,R1     ;Try to read 4K worth of blocks
        BCC     WRITE                   ;Branch if no error
        TSTB    @#ERRBYT                ;EOF?
        BEQ     PASS2                   ;Branch if yes
        JMP     ERROR                   ;Must be hard error, report it
WRITE:  MOV     R0,R2                   ;R2 = size of buffer just read
        .WRITW  #RAREA,#0,BUF,R2,R1     ;Write out the buffer
        BCC     ADDIT                   ;Branch if no error
        INCB    ERRNO                   ;ERR = 5x
        JMP     ERROR                   ;Report error
ADDIT:  ADD     #PAGSIZ,R1              ;Adjust block #
        BR      READ                    ;Then go get another buffer
PASS2:  INCB    ERRNO                   ;ERR = 6x
        .CRRG   #CAREA,#RDB1            ;Create a region
        BCC     35$                     ;Branch if no error
        JMP     ERROR                   ;Report error
35$:    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     40$                     ;Branch if no error
        TSTB    @#ERRBYT                ;EOF?
        BEQ     ENDIT                   ;Branch if yes
        JMP     ERROR                   ;Report hard error
40$:    MOV     R0,R2                   ;R2 = size of buffer read
        .READW  #RAREA,#0,BUF,R2,R1     ;Try to get same size from output file
        BCC     50$                     ;Branch if no error
        INCB    ERRNO                   ;ERR = 9x
        JMP     ERROR                   ;Report error
50$:    MOV     BUF,R4                  ;Get output buffer address
        MOV     BUF1,R3                 ;Get input buffer address
70$:    CMP     (R4)+,(R3)+             ;Verify that data is the same
        BNE     ERRDAT                  ;It's not, report error
        DEC     R2                      ;Are we finished?
        BNE     70$                     ;Branch if we aren't
        ADD     #PAGSIZ,R1              ;Adjust block # for page size
        BR      GETBLK                  ;Go get another buffer pair

ENDIT:  .PRINT  #ENDPRG                 ;Announce we're finished
XCLOS:  .CLOSE  #0                      ;Close output file
        .UNMAP  #CAREA,#WDB             ;Explicitly unmap 1st window
        .ELAW   #CAREA,#WDB             ;Explicitly eliminate 1st window
        .ELRG   #CAREA,#RDB             ;Eliminate 1st region
        .ELRG   #CAREA,#RDB1            ;Unmap,eliminate 2nd window & region
        .EXIT                           ;Exit program
```

```
ERROR:    MOVB     @#ERRBYT,R0              ;Make error byte code 2nd digit
          ADD      #'0,R0                   ;of error code...
          MOVB     R0,ERRNO+1               ;Put it in error message
          .PRINT   #ERR                     ;Print it...
          BR       XCLOS                    ;Go close output file
ERRDAT:   .PRINT   #ERRBUF                  ;Report verify failed...
          BR       XCLOS                    ;Go close output file

RDB:      .RDBBK   CORSIZ/32.               ;.RDDBK defines Region 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!)
CAREA:    .BLKW    2                        ;EMT argument blocks
RAREA:    .BLKW    6
DEFLT:    .WORD    0,0,0,0                  ;No default extensions
ENDPRG:   .ASCIZ   / * End of XM Example Program */
ERR:      .ASCII   /?XM Request or I-O Error # /
ERRNO:    .ASCIZ   /00/
ERRBUF:   .ASCIZ   /?Data Verification Error?/
ENDCRE    = .                              ;For CSIGEN - XM handlers loaded !

          .END     START
```

2.9 .CRRG (XM Only)

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

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.

Example:

Refer to example for the .CRAW request.

2.10 .CSIGEN

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.

The .CSIGEN request accepts a command string of the form *dev:output-filespec=dev:input-filespec/options*, and 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.

When called in general mode, the CSI closes channels 0 through 10(octal).

.CSIGEN loads all necessary handlers and opens the files as specified. The area specified for the device handlers must be large enough to hold all the necessary handlers simultaneously. If the device handlers exceed the area available, 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(octal). A null specification causes the associated channel to remain inactive. For example, the following string

```
*,LP:=F1,F2
```

causes channel 0 to be inactive since the first specification is null. Channel 1 is associated with the line printer, and channel 2 is inactive. Channels 3 and 4 are associated with two files on DK:, while channels 5 through 10 are inactive. 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.

Macro Call: .CSIGEN devspc,defext[,cstrng][,linbuf]

where:

devspc is the address of the memory area where the device handlers (if any) are to be loaded

defext is the address of a four-word block that contains the Radix–50 default file types. These file types are used when a file is specified without a file type (see Note 1)

cstrng is the address of the ASCIZ command string or a 0 if input is to come from the console terminal. (In an FB or XM 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 RET LF (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.

Note:

1. 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 block can be used to set up default file types for a macro assembler:

```
DEFEXT: .RAD50   "MAC"
        .RAD50   "OBJ"
        .RAD50   "LST"
        .WORD    0
```

   In the command string:

```
*DT0:ALPHA,DT1:BETA=DT2:INPUT
```

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

```
*DT0:OUTPUT=
*DT2:INPUT
```

   In other words, the equal sign is not necessary if only input files are specified.

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

3. 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   CSIGEN.MAC
;+
; .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

          ERRBYT=52                          ;Error Byte Location

START:    .CSIGEN  #DSPACE,#DEXT             ;Get string from terminal
          MOV      R0,BUFF                   ;R0 has first free location
          CLR      INBLK                     ;Input block #
          MOV      #LIST,R5                  ;EMT Argument list
READ:     .READW   R5,#3,BUFF,#256.,INBLK    ;Read 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$:       .PRINT                             ;Print the message
          CLR      R0                        ;Clear R0 for hard exit
          .EXIT                              ;Exit the program
2$:       .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
          .SRESET                            ;Release handler(s) from memory
          .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   /? Input error ?/
```

```
WTERR:    .ASCIZ   /? Output error ?/
          .EVEN

DSPACE    =.                              ;Handler(s) can be loaded starting here

          .END     START
```

2.10.1 Passing Option Information

In both general and special modes of the CSI, options and their associated values are returned on the stack. A CSI option is a slash (/) followed by any character. The CSI does not restrict the option to printing characters, although 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 command string

```
*DK:FOO/A,DT4:FILE.OBJ/A:100
```

has two A options. The first is associated with the input file DK:FOO. The second is associated with the input file DT4:FILE.OBJ and has a value of 100(octal). The format of the stack output of the CSI for options is as follows:

Word #	Value	Meaning
1 (top of stack)	N	Number of options found in command string. If N = 0, no options were found.
2	Option 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=DT3:INPUT/X:SY:20
```

on return, the stack is:

Stack Pointer →	4	Three options appeared (X option has two values and is treated as two options).
	101530	Last option = X; with file 3, has a value.
	20	Value of option X = 20(octal).
	101530	Next option = X; with file 3, has a value.
	075250	Next value of option X = RAD50 code for SY.
	505	Next option = E; associated with file 1, no value.
	100102	Option = B; associated with file 0 and has a value of 24.
	24	(octal).

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

```
*FILE[8,]/LP:/SY:FILE2[20,]=PC:/DT1:IN1/B/DT2:IN2/M:7
```

Assume also that the default file type block is:

```
DEFEXT:   /RAD50    'MAC'     ;INPUT FILE TYPE
          /RAD50    'OP1'     ;FIRST OUTPUT FILE TYPE
          /RAD50    'OP2'     ;SECOND OUTPUT FILE TYPE
          /RAD50    'OP3'     ;THIRD OUTPUT FILE TYPE
```

The results of the above CSI call are as follows:

1. An eight-block file named FILE.OP1 is entered on channel 0 on device DK:; channel 1 is open for output to the device LP:; a 20-block file named FILE2.OP3 is entered on the system device on channel 2.
2. Channel 3 is open for input from device PC:; channel 4 is open for input from a file IN1.MAC on device DT1:; channel 5 is open for input from IN2.MAC on device DT2:.
3. The stack contains options and values as follows:

Contents	Explanation
2	Two options found in string.
102515	Second option is M, associated with channel 5; has a value.
7	Numeric value is 7(octal).
2102	Option is B, associated with channel 4; has no value.

If the CSI were called in special mode, the stack would be the same as for the general mode call, and the descriptor table would contain:

```
OUTSPC:   15270     ;/RAD50   'DK'
          23364     ;/RAD50   'FIL'
          17500     ;/RAD50   'E'
          60137     ;/RAD50   'OP1'
             10     ;LENGTH OF 8 BLOCKS (DECIMAL)
          46600     ;/RAD50   'LP'
```

```
    0      ;NO NAME OR LENGTH SPECIFIED
    0
    0
    0
75250      ;.RAD50    'SY'
23364      ;.RAD50    'FIL'
22100      ;.RAD50    'E2'
60141      ;.RAD50    'OP3'
   24      ;LENGTH OF 20 (DECIMAL)
62170      ;.RAD50    'PC'
    0      ;NO NAME SPECIFIED
    0
    0
16077      ;.RAD50    'DT1'
35217      ;.RAD50    'IN1'
    0      ;.RAD50    '   '
50553      ;.RAD50    'MAC'
16100      ;.RAD50    'DT2'
35220      ;.RAD50    'IN2'
    0      ;.RAD50    '   '
50553      ;.RAD50    'MAC'
    0
    .
    .
    .
    0      (12 more zero words are returned)
```

Keyboard error messages that can occur 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	Output file specified already exists and is protected.

Notes:

1. In many cases, your program does not need to process options in CSI calls. However, you could inadvertently enter options at the console. In this case, it is wise to 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 word to be pushed onto the stack. This word indicates the number of options to follow.
2. Under an FB 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.

2.11 .CSISPC

The .CSISPC request calls the Command String Interpreter in special mode to parse the command string and return file descriptors and options to the program. In this mode, the CSI does not perform any .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 foreground/background 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 RET LF (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 RET LF (octal 15 and 12)

Notes:

1. 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.
2. 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:

```
*DT3:LIST.MAC[15]=PC:
```

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

16101	Radix–50 for DT3
46173	Radix–50 for LIS
76400	Radix–50 for T
50553	Radix–50 for MAC
00017	Octal value of size request

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

62170	Radix–50 for PC
0	No file name
0	specified
0	No file type given

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

Errors:

Errors are the same as in general mode except that invalid device specifications are checked only for output file specifications with null file names. Since .LOOKUP and .ENTER requests are not done, the valid error codes are:

Code	Explanation
0	Invalid command line.
1	Invalid device.

Example:

```
          .TITLE  CSISPC.MAC
;+
; .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 output 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.
;-

          .MCALL      .DSTATUS,.PRINT,.EXIT,.FETCH,.CSISPC,.DELETE

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,#OUTSP            ;Check on the output device
                                              ;(CSISPC catches illegal devices!)
          TST         STAT+4                  ;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
          .EXIT                               ;then exit program
2$:       .DELETE     #AREA,#0,#INSPEC        ;Now delete the file
          BCC         3$                      ;Branch if successful
          .PRINT      #NOFIL                  ;Print error message
          BR          START                   ;Then try again
3$:       .PRINT      #FILDEL                 ;Acknowledge successful deletion
          .EXIT                               ;then exit program

AREA:     .BLKW       2                       ;EMT Argument block
STAT:     .BLKW       4                       ;Block for status
DEFEXT:   .WORD       0,0,0,0                 ;No default extensions
FEFAIL:   .ASCIZ      /?.FETCH Failed?/       ;Fetch failed message
NOFIL:    .ASCIZ      /?File Not Found?/      ;File not found
FILDEL:   .ASCIZ      /!File Deleted!/        ;Delete acknowledgment
          .EVEN                               ;Fix boundary

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
```

2.12 .CSTAT

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 correspond to the following six points of information:

1. Channel status word (see the *RT–11 Software Support Manual* for details)
2. Starting block number of file (0 if sequential-access device, or if channel was opened with a non-file-structured .LOOKUP or .ENTER)
3. Length of file (0 if non-file-structured device, or if channel was opened with a non-file-structured .LOOKUP or .ENTER)
4. Highest relative block written since file was opened (no information if non-file-structured device). This word is maintained by the .WRITE/.WRITC/.WRITW requests
5. Unit number of device with which this channel is associated
6. Radix–50 of the device name with which the channel is associated (this is a physical device name, unaffected by any user name assignment in effect).

Errors:

Code	Explanation
0	The channel is not open.

Example:

```
          .TITLE  CSTAT.MAC
;+
; .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.
;-
          .MCALL      .CSTAT,.CSIGEN,.PRINT,.EXIT

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,#0,#ADDR          ;Get the status
          BCS         NOCHAN                  ;Channel 0 not open
          MOV         #ADDR+10,R5             ;Point to unit #
```

```
        MOV     (R5)+,R0            ;Unit # to R0
        ADD     (PC)+,R0            ;Make it RAD50
        .RAD50  / 0/
        ADD     (R5),R0             ;Get device name
        MOV     R0,DEVNAM           ;'DEVNAM' has RAD50 device name
        .EXIT                       ;Exit the program

NOCHAN: .PRINT  #MSG                ;Print error message
        .EXIT                       ;then exit program

MSG:    .ASCIZ  /?No Output 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=.                            ;Start CSI tables here...

        .END    START
```

2.13 .CTIMIO (Device Handler Only)

The .CTIMIO macro cancels the device time-out request in the handler interrupt service section. It is used when an interrupt occurs to disable the completion routine (see .TIMIO).

If the time interval has already elapsed and the device has, therefore, timed out, the .CTIMIO request fails. 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 was already queued.

The device time-out feature must have been selected during the system generation process.

Macro Call: .CTIMIO tbk

where:

tbk is the address of the seven-word timer block shown in Table 2–1

Table 2–1: 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	user	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	user	Sequence number of timer request. The valid range of sequence numbers is from 177000 to 177377.
12	monitor	–1
14	user	Address of the completion routine to execute if timeout occurs. The monitor zeroes this word when it calls the completion routine, indicating that the timer block is available for reuse.

The .CTIMIO macro expands as follows:

```
.CTIMIO   tbk

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

Example:

Refer to the example for the .TIMIO request.

2.14 .DATE

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

```
BIT:   15 14  13 ... 10  9 ... 5  4 ... 0
        0  0  MONTH      DAY      YEAR
```

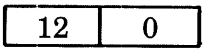
The year value in bits 4–0 is the actual year minus 1972. The day in bits 9 to 5 is a number from 1 to the length of the month. The month in bits 13 to 10 is a number from 1 to 12.

NOTE

RT–11 support of month and year rollover is a system generation special feature; otherwise, the keyboard monitor DATE command must be issued to change the month and year.

Macro Call: .DATE

Request Format:

R0 = 

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 may be assembled separately and linked with
; user written programs
;
; INPUT:              none
;
; OUTPUT:             R0 = MONTH (1-12)
;                     R1 = DAY   (1-31)
;                     R2 = YEAR  (Last two digits)
;
```

```
; ERRORS:             R0 = 0 if no date entered
;-

          .MCALL      .DATE

DATE::    .DATE                           ;Get date in R0 via .DATE request
          MOV         R0,R2               ;Copy R0
          BEQ         1$                  ;If zero, no date was entered
          BIC         #^C37,R2            ;Clear all but year bits
          ADD         #72.,R2             ;Make it current year
          MOV         R0,R1               ;Copy date word again
          ASL         R1                  ;Get day bits
          ASL         R1                  ;on a byte boundary...
          ASL         R1                  ;
          SWAB        R1                  ;Put day bits in low order byte
          BIC         #^C37,R1            ;Clear all but day bits
          SWAB        R0                  ;Put month bits in low byte
          ASR         R0                  ;Right adjust
          ASR         R0                  ;month bits...
          BIC         #^C17,R0            ;Clear all but month bits
1$:       RETURN                          ;Return to calling program

          .END
```

2.15 .DELETE

The .DELETE request deletes a named file from an indicated device. The .DELETE request is invalid for magtapes. The .SERR programmed request can be used to allow the program to process any errors.

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(octal)
dblk	is the address of a four-word Radix–50 descriptor of the file to be deleted
seqnum	file number for cassette operations: if this argument is blank, a value of 0 is assumed

Request Format:

R0 → area:

0	chan
dblk	
seqnum	

Notes:

The channel specified in the .DELETE request must not be open when the request is made, or an error will occur. The file is deleted from the device, and an empty (UNUSED) entry of the same size is put in its place. A .DELETE issued to a non-file-structured device is ignored. .DELETE requires that the handler to be used be in memory at the time the request is made. When the .DELETE is complete, the specified channel is left inactive.

Errors:

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

Example:

```
          .TITLE  DELETE.MAC
;+
; .DELETE - This is an example in the use of the .DELETE 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 output 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.
;-

          .MCALL      .DSTATUS,.PRINT,.EXIT,.FETCH,.CSISPC,.DELETE

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,#OUTSP          ;Check on the output device
                                            ;(CSISPC catches illegal devices!)
          TST         STAT+4                ;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
          .EXIT                             ;then exit program
2$:       .DELETE     #AREA,#0,#INSPEC      ;Now delete the file
          BCC         3$                    ;Branch if successful
          .PRINT      #NOFIL                ;Print error message
          BR          START                 ;Then try again
3$:       .PRINT      #FILDEL               ;Acknowledge successful deletion
          .EXIT                             ;then exit program

AREA:     .BLKW       2                     ;EMT Argument block
STAT:     .BLKW       4                     ;Block for status
DEFEXT:   .WORD       0,0,0,0               ;No default extensions

FEFAIL:   .ASCIZ      /?.FETCH Failed?/     ;Fetch failed message
NOFIL:    .ASCIZ      /?File Not Found?/    ;File not found
FILDEL:   .ASCIZ      /!File Deleted!/      ;Delete acknowledgment
          .EVEN                             ;Fix boundary

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
```

2.16 .DEVICE (FB and XM Only)

This request allows your program to load device registers with any necessary values when the program is terminated. You set up the list of addresses with the specified values. Upon issuing an .EXIT request or a CTRL/C from the terminal, this list is picked up by the system and the designated addresses are loaded with the corresponding values. This function is primarily designed to allow your program to turn off a device's interrupt enable bit when the program servicing the device terminates.

Successive calls to .DEVICE are allowed when you need to link requested tables. When the job is terminated for any reason, the list is scanned once. At that point, the monitor disables the feature until another .DEVICE call is executed. Thus, background programs that are reenterable should include .DEVICE as a part of the reenter code.

The .DEVICE request is ignored when it is issued by a virtual job running under the XM monitor.

Macro Call: .DEVICE area,addr[,link]

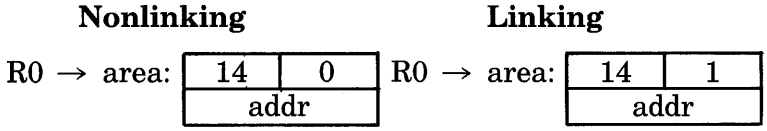
where:

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

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

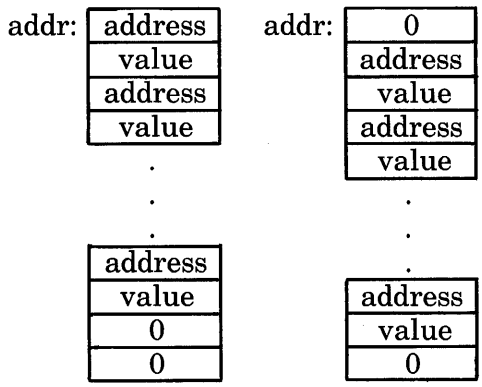
link — is an optional argument that, if present, specifies linking of tables on successive calls to .DEVICE. If the argument is omitted, the list referenced in the previous .DEVICE request is replaced by the new list. The argument must be supplied to cause linking of lists; however, linked and unlinked list types cannot be mixed

Request Format:



NOTE

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



Errors:

None.

Example:

```
          .TITLE  DEVICE.MAC

;+
; .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
          .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                     ;Fill it with '0'
          .WORD       0                     ;List terminator
BUFFR:                                      ;Data to send over DL11
          .REPT       8.                    ;8 lines of 32 characters...
          .ASCIZ      /Hello DL11... Are You There ??/
          .ENDR
NOVEC:    .ASCIZ      /?Vector already protected?/ ;Error message text

          .END        START
```

2.17 .DRAST (Device Handler Only)

The .DRAST macro sets up the interrupt and abort entry points, lowers the processor priority, and references a global symbol $INPTR, which contains a pointer to the $INTEN routine in the resident monitor. This pointer is filled in by the bootstrap (for a system device) or at .FETCH 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 an RTS PC instruction at the abort entry point, which is the word immediately preceding the interrupt entry point

Example:

```
        .TITLE      SP.MAC

;+
; SP.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 SP').
;-

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

.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 TIM$IT,    TIM$IT=0

.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)

        .QELDF                          ;Use .QELDF to define Q-Elem offsets
                                        ;Among others & of interest to us are:
;Q.BLKN = 4                             ;Offset to Block # (SPCQE => Q.BLKN)
;Q$CSW = -2                             ;Offset from Q.BLKN to CSW pointer
;Q$BUFF = 4                             ;  "      "     "   "  User buffer ptr
;Q$WCNT = 6                             ;  "      "     "   "  Word count

        .DRBEG      SP,SP$VEC,SPSIZ,SPSTS ;Begin driver code with .DRBEG
                                        ;MACRO expansion is...
;       .WORD       <SPEND-SPSTRT>      ;Size of driver (handler)
;       .WORD       0                   ;Size of device
;       .WORD       20000               ;Device status (Write only)
;       .WORD       ERL$G+<MMG$T*2>+<TIM$IT*4>;Default options
;SPSTRT::                               ;Beginning of driver
;       .WORD       SPVEC+4             ;Interrupt vector
;       .WORD       SPINT-.,^O340       ;Offset to Int svc rtne & Priority
;SPCQE:: .WORD      0                   ;Queue element pointers
;SPLQE:: .WORD      0                   ;(Point to 3rd word in element!)
        MOV         SPCQE,R4            ;R4 => Current Q-Element
        ASL         Q$WCNT(R4)          ;Make word count byte count
        BCC         SPERR               ;A read from a write/only device?
        BEQ         SPDUN               ;Zero word count...Just exit
SPRET:  BIS         #100,@#SP$CSR       ;Enable DL-11 interrupt
        RETURN                          ;Return to monitor

; INTERRUPT SERVICE ROUTINE

        .DRAST      SP,SP$PRI           ;Use .DRAST to define Int Svc Sect.
                                        ;MACRO expansion...
;       RTS         PC                  ;Abort Entry Point
;SPINT:: JSR        R5,@$INPTR          ;Do a .INTEN to alert monitor
;       .WORD       ^C<SP$PRI*^O40>&^O340 ;and drop processor priority
        MOV         SPCQE,R4            ;R4 => Q-Element
        TST         @#SP$CSR            ;Error?
        BMI         SPRET               ;Yes...'hang' until ready
        BIC         #100,@#SP$CSR       ;Disable interrupts
        .FORK       SPFORK              ;Continue at FORK level
SPNXT:  TSTB        @#SP$CSR            ;Is device ready?
        BPL         SPRET               ;No...go wait 'till it is
        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          SPNXT               ;Try to output another character

SPERR:  BIS         #IOERR,@Q$CSW(R4)   ;Set error bit in CSW
SPDUN:  .DRFIN      SP                  ;Use .DRFIN to return to Monitor
                                        ;MACRO expansion...
```

```
;         MOV         PC,R4                ;Calculate PIC addr of current
;         ADD         #SPCQE-.,R4          ;queue element pointer
;         MOV         @#54,R5              ;Put addr of base of RMON in R5
;         JMP         @^O270(R5)           ;Jump to handler completion in monitor

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

          .DREND      SP                   ;Use .DREND to end code
                                           ;MACRO expansion...
;$INPTR::.WORD        0                    ;Addr of .INTEN code in RMON
;$FKPTR::.WORD        0                    ;Addr of .FORK processor in RMON
;SPEND == .                                ;End of driver

          .END
```

2.18 .DRBEG (Device Handler Only)

The .DRBEG macro sets up the information in block 0 and the first five words of the handler. This macro also generates the appropriate global symbols for your handler. Before you use .DRBEG, invoke .DRDEF to define xxCSR, xxVEC, xxDSIZ, and xxSTS (see Section 2.20).

Macro Call: .DRBEG name

where:

name is a two-character device name

Example:

Refer to the example for .DRAST.

2.19 .DRBOT (Device Handler Only)

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 (see Section 2.21) 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]

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

The .DRBOT macro puts a pointer to the start of the primary driver into location 62 of the handler file. It puts the length (in bytes) of the primary driver into location 64. 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 is called before the .DREND macro that you issue. The code for the primary driver is placed between the .DRBOT and .DREND calls.

Example:

Refer to the *RT–11 Software Support Manual* for an example showing the use of .DRBOT.

2.20 .DRDEF (Device Handler Only)

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

Macro Call: .DRDEF name,code,stat,size,csr,vec

where:

name is the two-character device name

code is the numeric code that is the device identifier value for the device

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

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

vec is the default value for the device's vector

The .DRDEF macro performs the following operations:

1. A .MCALL is done for the following macros: .DRAST; .DRBEG; .DRBOT; .DREND; .DRFIN; .DRSET; .DRVTB; .FORK; .QELDF.
2. If the system generation conditionals TIMIT, MMGT, 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.
3. The .QELDF macro is invoked to define symbolic offsets within a queue element.
4. The symbols listed above are defined for the device status bits.

5. The following symbols are defined:

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

6. The symbol xxDSIZ is set to the value specified in size.
7. The symbol xx$COD is set to the specified device identifier code.
8. The symbol xxSTS is set to the value of the device identifier code plus the status bits.
9. If the symbol xx$CSR is not defined, it is set to the default csr value.
10. If the symbol xx$VEC is not defined, it is set to the default vector value.
11. The symbols xx$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:

Refer to the *RT–11 Software Support Manual* for an example showing the use of .DRDEF.

2.21 .DREND (Device Handler Only)

The .DREND macro generates the termination table for the termination section of the device handler.

Macro Call: .DREND name

where:

name is the two-character device name

The generation of the termination table, dependent upon certain conditions, is as follows:

Label	Addresses	
$RLPTR:	.WORD 0	($RELOC)
$MPPTR:	.WORD 0	($MPPHY)
$GTBYT:	.WORD 0	($GETBYT)
$PTBYT:	.WORD 0	($PUTBYT)
$PTWRD:	.WORD 0	($PUTWRD)
$ELPTR:	.WORD 0	($ERLOG)
$TIMIT:	.WORD 0	($TIMIO)
$INPTR:	.WORD 0	($INTEN)
$FKPTR:	.WORD 0	($FORK)

The generation of the labels depends upon the special features chosen during the system generation process. All the pointers in the termination section are initialized when the handler is loaded into memory with the .FETCH request. If the device handler is a system device, the pointers are initialized at boot time with the addresses shown in the address column.

The addresses are located within the monitor. The first five addresses are the locations of subroutines in the resident monitor that are available to device handlers in an extended memory environment. 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 by a .FETCH or LOAD command.

Example:

Refer to the example for .DRAST.

2.22 .DRFIN (Device Handler 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

Example:

Refer to the example for .DRAST.

2.23 .DRSET (Device Handler 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 appear one after another.

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 Register 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 routine must follow the option table in block 0 and must not go above address 776

mode — is an optional argument to indicate the type of SET parameter. 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

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. Thus 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.

Example:

Refer to the *RT–11 Software Support Manual* for an example of .DRSET.

2.24 .DRVTB (Device Handler 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 PSW are set to 7 even if you omit this argument

Example:

Refer to the *RT–11 Software Support Manual* for an example of .DRVTB.

2.25 .DSTATUS

This .DSTATUS request obtains information about a particular device.

Macro Call: .DSTATUS 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

.DSTATUS 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 number that identifies the device in the system. The values are currently defined in octal as follows:

0 = RK05 Disk
1 = TC11 DECtape
2 = Error Logger
3 = Line Printer
4 = Console Terminal or Batch Handler
5 = RL01/RL02 Disk
6 = RX02 Diskette
7 = PC11 High-speed Paper Tape Reader and Punch
10 = Reserved (V2 PP handler)
11 = TU10 Magtape
12 = RF11 Disk
13 = TA11 Cassette
14 = Card Reader (CR11,CM11)
15 = Reserved
16 = RJS03/RJS04 Fixed-head Disk
17 = Reserved
20 = TJU16 Magtape
21 = RP02/RP03 Disk
22 = RX01 Diskette
23 = RK06/RK07 Disk
24 = Reserved
25 = Null Handler
26–30 = Reserved (DECnet)
31–33 = Reserved (CTS–300,LQ,LR,LS)
34 = TU58 DECtape II
35 = TS11 Magtape
36 = PDT–11/130
37 = PDT–11/150
40 = Reserved
41 = Serial Line Printer Handler (LS)
42 = Message Queue Handler (MQ)
43 = DRV11–J Interface (MRRT)
44 = Down-line Load Handler (XT) (MRRT–11 only)

45 = Reserved
46 = Logical Disk Handler
47 = KT–11 VM Handler
50 = MSCP Class Disk Handler
51 = Single-line Editor

Bit 8: 1 = Handler can access variable-sized volumes and supports .SPFUN 373
0 = All volumes used by this device are the same size

Bit 9: 1 = Enter handler at abort entry whenever program terminates for any reason
0 = Do not enter at abort entry point unless conditions for bit 11 are satisfied

Bit 10: 1 = Handler accepts .SPFUN requests (for example, MT, CT, DX)
0 = No .SPFUN requests accepted

Bit 11: 1 = Enter handler abort entry every time a job is aborted
0 = Handler abort entry taken only if there is an active queue element belonging to aborted job

Bit 12: 1 = Non RT–11 directory-structured device (magtape, cassette)

Bit 13: 1 = Write-only device (line printer, serial line printer)

Bit 14: 1 = Read-only device (card reader, paper tape reader)

Bit 15: 1 = Random-access device (disk, DECtape)
0 = Sequential-access device (line printer, paper tape, card reader, magtape, cassette, terminal)

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: zero implies that it must be fetched before it can be used. The address returned is the load address of the handler +6.

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.

The device name can be a user-assigned name. .DSTATUS 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.

Errors:

Code	Explanation
0	Device not found in tables.

Example:

```
          .TITLE  DSTAT.MAC
;+
; .DSTATUS - This is an example in the use of the .DSTATUS 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 output 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.
;-

          .MCALL      .DSTATUS,.PRINT,.EXIT,.FETCH,.CSISPC,.DELETE

START:    .CSISPC     #OUTSP,#DEFEXT          ;Use .CSISPC to get output spec
          .DSTAT      #STAT,#OUTSP            ;Check on the output device
                                              ;(CSISPC catches illegal devices!)
          TST         STAT+4                  ;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
          .EXIT                               ;then exit program
2$:       .DELETE     #AREA,#0,#INSPEC        ;Now delete the file
          BCC         3$                      ;Branch if successful
          .PRINT      #NOFIL                  ;Print error message
          BR          START                   ;Then try again
3$:       .PRINT      #FILDEL                 ;Acknowledge successful deletion
          .EXIT                               ;then exit program

AREA:     .BLKW       2                       ;EMT Argument block
STAT:     .BLKW       4                       ;Block for status
DEFEXT:   .WORD       0,0,0,0                 ;No default extensions

FEFAIL:   .ASCIZ      /?.FETCH Failed?/       ;Fetch failed message
NOFIL:    .ASCIZ      /?File Not Found?/      ;File not found
FILDEL:   .ASCIZ      /!File Deleted!/        ;Delete acknowledgment
          .EVEN                               ;Fix boundary

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
```

2.26 .ELAW (XM Only)

The .ELAW request eliminates a virtual address window. An implied unmapping of the window occurs when its definition block is eliminated.

Macro Call: .ELAW area[,addr]

where:

area is the address of a two-word 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.

Example:

Refer to the example for the .CRAW request.

2.27 .ELRG (XM Only)

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.

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.

Example:

Refer to the example for the .CRAW request.

2.28 .ENTER

The .ENTER request allocates space on the specified device and creates a tentative entry in the directory for the named 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(octal)

dblk — is the address of a four-word Radix–50 descriptor of the file to be operated upon

len — is the file size specification. If the argument is omitted, it is not set to 0 in area. An argument of #0 must be specified to accomplish this. If an argument is left blank, the corresponding location in area is assumed to be set

The value of this argument determines the file length allocation as follows:

- 0 — either half the largest empty entry or the entire second-largest empty entry, whichever is larger. (A maximum size for nonspecific .ENTER requests can be patched in the monitor by changing resident monitor offset 314; refer to the example for .PVAL)
- m — a file of m blocks. The size, *m*, can exceed the maximum mentioned above
- –1 — the largest empty entry on the device

seqnum — is a file number for magtape or cassette. Programming for specific devices such as magtape or cassettes is discussed in detail in Chapter 10 of the *RT–11 Software Support Manual*. For cassette operation, if this argument is blank, a value of 0 is assumed

For magtape, *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)
- n — position magtape at file sequence number n if n is greater than zero and the file name is not null
- –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. The magtape is now positioned correctly. A new logical end-of-tape is implied

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.

The file created with an .ENTER request is not a permanent file until a .CLOSE request is given on that channel. Thus, 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 referencing the appropriate block number. 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.

Each job 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 an .ENTER request is made, the device handler must be in memory. Thus, a .FETCH should normally be executed before an .ENTER can be done.

Notes:

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 DECtape and diskette) 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.

Errors:

Code	Explanation
0	Channel is in use.
1	In a fixed-length request, no space greater than or equal to *m* was found; or the device or the directory was found to be full.
3	A file by that name already exists and is protected. A new file was not opened.

Example:

```
          .TITLE  ENTER.MAC
;+
; .ENTER - This is an example in the use of the .ENTER request.
; The example makes a copy of the file 'TECO.SAV' on device DK:
;-
```

```
        .MCALL      .LOOKUP,.ENTER,.WRITW,.READW,.CLOSE
        .MCALL      .PRINT,.EXIT
        ERRBYT=52
START:  .LOOKUP     #AREA,#0,#TECO          ;Lookup file TECO.SAV
        BCS         5$                      ;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         6$                      ;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        @#ERRBYT                ;Was error EOF?
        BEQ         3$                      ;Branch if yes
        MOV         #RERR,R0                ;Hard read error message to R0
        BR          7$                      ;Branch to print message
2$:     .WRITW      #AREA,#1,#BUFFR,#256.,BLK ;Write a block
        INC         BLK                     ;Bump block # (doesn't affect C bit)
        BCC         1$                      ;Branch if write was ok
        MOV         #WERR,R0                ;R0 => Write error message
        BR          7$                      ;Branch to print message
3$:     .CLOSE      #1                      ;Make new file permanent
        MOV         #DONE,R0                ;R0 => Done message
        BR          7$                      ;Branch to print message
5$:     MOV         #NOFIL,R0               ;R0 => File not found message
        BR          7$                      ;Branch to print it
6$:     MOV         #NOENT,R0               ;R0 => Enter Failed message
7$:     .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
TECO:   .RAD50      /DK/                    ;File descriptors...
        .RAD50      /TECO/
        .RAD50      /SAV/
TFILE:  .RAD50      /DK/
        .RAD50      /OLDTEC/
        .RAD50      /SAV/
NOFIL:  .ASCIZ      /?File not found?/      ;Message text...
NOENT:  .ASCIZ      /?.ENTER Failed?/
WERR:   .ASCIZ      /?Write Error?/
RERR:   .ASCIZ      /?Read Error?/
DONE:   .ASCIZ      /TECO Copy Complete/
        .END        START
```

2.29 .EXIT

The .EXIT request causes the user program to terminate. When used from a background job under the FB monitor or XM monitor, or in SJ, .EXIT causes KMON to run in the background area. All outstanding mark time requests are canceled. Any I/O requests and/or completion routines pending for that job are allowed to complete. If part of the background job resides where KMON and USR are to be read and SET EXIT SWAP is in effect, the user job is written onto the system swap blocks (the file SWAP.SYS). KMON and USR are then loaded and control goes to KMON in the background area. If SET EXIT NOSWAP is in effect, the user program is simply 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.

The .EXIT request allows a user program to pass command lines to KMON in the chain information area (locations 500–777octal) for execution after the job exits. This is performed under the following conditions:

1. The word (not byte) location 510 must contain the total number of bytes of command lines to be passed to KMON.

2. The command lines are stored beginning at location 512. The lines must be .ASCIZ strings with no embedded carriage return or line feed. For example:

```
      .=510
      .WORD B-A
A:    .ASCIZ /COPY A.MAC B.MAC/
      .ASCIZ /DELETE A.MAC/
B:
```

3. The user program must set bit 5 or bit 11 in the Job Status Word immediately 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:

1. If bit 11 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.
2. If bit 5 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.
3. 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 must make sure that the data being passed to KMON is not destroyed during the .EXIT request. Extreme care should be exercised so that the user stack does not overwrite this data area. If the user passes command lines to KMON, the stack pointer should be reset to 1000(octal) or above before an exit is made.

Macro Call: .EXIT

Errors:

None.

Example:

```
          .TITLE  EXIT.MAC
;+
; .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

CHNIF$    = 4000                          ;Chain bit in JSW
JSW       = 44                            ;JSW location
START:    MOV       #510,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       #CHNIF$,@#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
          .ASCIZ    "DIRECT/FULL *.MAC"
CMDEND:
          .EVEN
          .END      START
```

2.30 .FETCH/.RELEAS

The .FETCH request loads device handlers into memory from the system device.

Macro Call: .FETCH addr,dnam

where:

addr is the address where the device handler is to be loaded

dnam is the pointer to the Radix–50 device name

The storage address for the device handler is passed on the stack. When the .FETCH is complete, R0 points to the first available location above the handler. If the handler is already in memory, R0 contains the same value that was initially specified in the argument *addr*. If the argument on the stack is less than 400(octal), it is assumed that a handler .RELEAS is being done. (.RELEAS does not dismiss a handler that was loaded from the KMON; an UNLOAD must be done.) After a .RELEAS, a .FETCH must be issued in order to use the device again.

Several requests require a device handler to be in memory for successful operation. These include:

.CLOSE	.READC	.READ	.SFDAT
.LOOKUP	.WRITC	.WRITE	.FPROT
.ENTER	.READW	.SPFUN	
.RENAME	.WRITW	.DELETE	

When running under the foreground/background monitor, handlers for the foreground program or a system job must be loaded with the LOAD command before execution.

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 in the system, or there is no handler for that device in the system.

Example:

```
          .TITLE  FETCH.MAC
;+
; .FETCH - This is an example in the use of the .FETCH 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 output 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.
;-

          .MCALL      .DSTATUS,.PRINT,.EXIT,.FETCH,.CSISPC,.DELETE

START:    .CSISPC     #OUTSP,#DEFEXT          ;Use .CSISPC to get output spec
          .DSTAT      #STAT,#OUTSP            ;Check on the output device
                                              ;(CSISPC catches illegal devices!)
          TST         STAT+4                  ;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
          .EXIT                               ;then exit program
2$:       .DELETE     #AREA,#0,#INSPEC        ;Now delete the file
          BCC         3$                      ;Branch if successful
          .PRINT      #NOFIL                  ;Print error message
          BR          START                   ;Then try again
3$:       .PRINT      #FILDEL                 ;Acknowledge successful deletion
          .EXIT                               ;then exit program

AREA:     .BLKW       2                       ;EMT Argument block
STAT:     .BLKW       4                       ;Block for status
DEFEXT:   .WORD       0,0,0,0                 ;No default extensions

FEFAIL:   .ASCIZ      /?.FETCH Failed?/       ;Fetch failed message
NOFIL:    .ASCIZ      /?File Not Found?/      ;File not found
FILDEL:   .ASCIZ      /!File Deleted!/        ;Delete acknowledgment
          .EVEN                               ;Fix boundary

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
```

The .RELEAS request notifies the monitor that a fetched device handler is no longer needed. The .RELEAS is ignored if the handler is (1) the system device, (2) not currently resident, (3) resident because of a LOAD command

to the keyboard monitor. .RELEAS from the foreground or system job under the FB monitor or the XM monitor is always ignored, since the foreground job in a FB environment or extended memory environment can only use handlers that have been loaded by the LOAD command.

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 DECtape handler (DT) is loaded into memory,
;used, then released. If the system device is DECtape, the handler is
;always resident, and .FETCH will return HSPACE in R0.

        .MCALL  .FETCH,.RELEAS,.EXIT,.PRINT

START:  .FETCH  #HSPACE,#DTNAME  ;Load DT handler
        BCS     FERR             ;Not available

; Use handler

        .RELEAS #DTNAME          ;Mark DT no longer in
                                 ;memory
        BR      START
FERR:   .PRINT  #NODT            ;DT not available
DTNAME: .RAD50  /DT /            ;Name for DT handler
NODT:   .ASCIZ  /?DT HANDLER NOT AVAILABLE/
        .EVEN
HSPACE:                          ;Beginning of handler
                                 ;area

        .END    START
```

2.31 .FORK (Device Handler and Interrupt Service Routine Only)

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:

```
.FORK fkblk
JSR %5,@$FKPTR
.WORD fkblk-.
```

The .FORK call must be preceded by an .INTEN call, and the address of a four-word block must be supplied with the request. Your program must not have left any information on the stack between the .INTEN and the .FORK call. The contents of registers R4 and R5 are preserved through the call, and on return registers R0 through R3 are available for use.

If you are using a .FORK call from a device handler, it is assumed that you are also using the other macros (.QELDF, .DRBEG, .DRAST, .DRFIN, and .DREND) provided for handlers.

The .DREND macro allocates a word for $FKPTR. This word is filled in at bootstrap time for a system device or at LOAD or .FETCH time for a non-system device.

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:

```
          SYSPTR=54                       ;Address containing base
                                          ;address of RMON
          FORK=402                        ;Monitor offset containing
                                          ;offset to fork processor
          .GVAL     #AREA,#FORK           ;Return offset in R0
          ADD       @#SYSPTR,R0           ;Add RMON base address
          MOVE      R0,$FKPTR             ;Save as address of the
                                          ;fork processor
           .
           .
           .
AREA:     .BLKW 2
$FKPTR:   .WORD 0
```

Once the pointer is set up, use the macro in the usual way as follows:

```
.FORK   fkblk
```

This method permits you to preserve both R4 and R5 across the fork.

The .FORK request is linked into a queue and serviced on a first-in first-out basis. On return to the driver 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 Software Support Manual* gives additional information on the .FORK request.

Notes:

For use within a user interrupt service routine, monitor fixed offset 402 (FORK) contains the offset from the start of the resident monitor to the

.FORK request processor. A .FORK can be done by computing the address of the .FORK request processor and using a subroutine instruction. (Under the XM monitor, only privileged jobs can contain user interrupt service routines.) For example:

```
        SYSPTR=54                       ;Address containing base
                                        ;address of RMON
        FORK=402                        ;Monitor offset containing
                                        ;offset to fork processor
        .GVAL     #AREA, #FORK          ;Return offset in R0
        ADD       @#SYSPTR, R0          ;Add RMON base address
        MOV       R0, R4
        JSR       R5, (R4)              ;Call fork process
        .WORD     BLOCK-.
          .
          .
          .
AREA:   .BLKW 2
BLOCK:  .BLKW 4
```

This method destroys the contents of R4.

Example:

Refer to the example following the description of .DRAST.

2.32 .FPROT

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. For example, a .LOOKUP of a protected file followed by a .WRITE to the file is permitted.

Protection is enabled by setting bit 15 of a file's directory entry status word.

Macro Call: .FPROT area, chan, dblk, prot

where:

area is the address of a four-word EMT argument block

chan is a channel number in the range 0–376(octal)

dblk is the address of a four-word block containing the filespec in Radix–50 of the file

prot = #1 — 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 in use
1	File not found
2	Invalid operation
3	Invalid value for PROT

Example:

```
;.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

START:  .CSISPC  #OUTSP, #DEFEXT        ;Use CSI to get input filespec
        .DSTAT   #STAT,#INSPEC          ;Check the device
        TST      STAT+4                 ;to see if the hanndler is resident
        BNE      1$                     ;Branch if it is
        .FETCH   #INSPEC                ;Otherwise, load that handler
        BCC      1$                     ;ok
        .PRINT   #LOFAIL                ;Otherwise, print load error message
        BR       START                  ;and try again
1$:     .FPROT   #EMTBLK, #0, #INSPEC, #1  ;Mark file as protected
        BCC      2$                     ;and branch if okay
        .PRINT   #PRFAIL                ;Otherwise, print protect error message
        BR       START                  ;and try again
2$:     .SFDAT   #EMTBLK, #0, #INSPEC, #0  ;Finally, set current date
                                        ;A date of 0 means "use current system date"
        BCC      10$                    ;Branch if everything is okay
        .PRINT   #SDFAIL                ;Otherwise, print date error message
        BR       START                  ;and try again
10$:    .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   /Error in .LOAD request/
PRFAIL: .ASCIZ   /Error in .FPROT request/
SDFAIL: .ASCIZ   /Error in .SFDAT request/
        .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
```

2.33 .GMCX (XM Only)

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.

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:

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

Otherwise, these locations are zeroed.

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	

Errors:

Code	Explanation
3	An illegal window identifier was specified.

Example:

Refer to the example for the .CRAW request.

2.34 .GTIM

.GTIM allows user programs to access 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 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 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 SJ monitor, the time of day is not reset unless SJ timer support was selected during the system generation process. The month is not automatically updated in either monitor. (Proper month and year rollover is a special feature that you enable through the system generation process.)

The default clock rate is 60 cycles, that is, 60 ticks per second. Consult the *RT–11 System Generation Guide* if conversion to a 50-cycle rate is necessary.

Because day rollover is done only through a .GTIM request, make sure that your program receives the correct time and day by issuing a .GTIM request before using the .DATE request. Nearly all RT–11 system utility programs issue a .GTIM request to make sure that rollover occurs daily. If you do not use a system utility program regularly, issue a .GTIM request at least once during a 24-hour period.

NOTE

There are also several SYSLIB routines that perform time conversion (see Chapter 3). They are CVTTIM, TIMASC, TIME, and SECNDS.

Errors:

None.

Example:

```
          .TITLE  GTIM.MAC

;+
; .GTIM - This is an example in the use of the .GTIM request.
; This example is a subroutine that can be assembled separately
; and linked with a user program.
;
; 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 '$DIVTK' & '$DIVGO'
;-

          .GLOBL      $DIVTK,$DIVGO
          .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:  .WORD   0,0             ;Ticks since midnight returned here

        .END
```

2.35 .GTJB

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. The values returned are:

Word 1 Job Number = priority level *2 (background job is 0; system jobs are 2, 4, 6, 10, 12, 14; and foreground job is 16 in system job monitors; background job is 0 and foreground job is 2 in FB and XM monitors; job number is 0 in a SJ monitor)

2 High-memory limit of job partition (highest location available to a job in low memory if the job executes a privileged .SETTOP #–2 request)

3 Low-memory limit of job partition (first location)

4 Pointer to I/O channel space

5 Address of job's impure area in FB and XM monitors

6 Low byte: unit number of job's console terminal (used only with multiterminal option; 0 when multiterminal feature is not used)
High byte: reserved for future use

7 Virtual high limit for a job created with the linker /V option (XM only; 0 when not running under the XM monitor or if /V option is not used)

8–9 Reserved for future use

10–12 ASCII logical job name (system job monitors only; contains zeroes for non-system jobs in FB and XM, not defined in SJ)

jobblk is a pointer to a three-word ASCII logical job name for which data is being requested

Word 4 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 when the status of a job is requested, it is interpreted as a job number. If the *jobblk* argument is 'ME', or equals –1, information about the current job is returned. If the *jobblk* argument is 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 F/B 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	0
addr	
jobblk	

Errors:

Code	Explanation
0	No such job currently running.

Example:

See the program GTJB.MAC in the example listing.

```
        .TITLE GTJB.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                     ;Fixed offset to SYSGEN word
        SYSJOB= 40000                   ;System Job option bit

START:  MOV     #2,     R1              ;Assume FG Job number = 2
        .GVAL   #LIST,  #SYSGEN         ;Get SYSGEN option word
        BIT     #SYSJOB, R0             ;System Job monitor?
        BEQ     1$                      ;Branch if not
        MOV     #16,    R1              ;If so, FG Job number = 16
1$:     .GTJB   #LIST, #JOBARG, R1      ;Find out if FG loaded
        BCS     2$                      ;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.

LIST:   .BLKW   3                       ;EMT Argument block
```

```
JOBARG:  .BLKW    12.                    ;Job parameters passed back here

FGLOAD:  .ASCIZ   /!FG Loaded!/          ;FG loaded message
NOFG:    .ASCIZ  /?No FG Job?/           ;No FG message

         .END     START
```

2.36 .GTLIN

The .GTLIN request collects a line of input from either the console terminal or an indirect command file, if one is active. This request is similar to .CSIGEN and .CSISPC in that it requires the USR, but no format checking is done on the input line. Because the .GTLIN command is implemented in the USR, the CSI will generate an error message if you attempt to input more than 80 characters to a .GTLIN request. Normally, .GTLIN collects a line of input from the console terminal and returns it in the buffer specified by you. However, if there is an indirect command file active, .GTLIN collects the line of input from the indirect command file just as though it were coming from the terminal.

When bit 3 of the Job Status Word is set and your program encounters a CTRL/C in an indirect command file, the .GTLIN request collects subsequent lines from the terminal. Note that 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; this causes the program to abort. Further input will come from the indirect command file, if there are any more lines in it. When bit 14 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 command 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

In an F/B environment, .GTLIN performs a temporary implicit unlock while the line is being read from the console.

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 forces .GTLIN to take its input from the terminal rather than from an indirect file

NOTE

The only requests that can take their input from an indirect command file are .CSIGEN, .CSISPC, and .GTLIN. The .TTYIN and .TTINR requests cannot get characters from an indirect command file. 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 command file. However, the response to the linker's *Transfer Symbol?* query would normally be collected through a .GTLIN, so that the LINK command could be invoked and the start address specified from an indirect file. Also, if there is no active indirect command file, .GTLIN simply collects an input line from the console terminal by using .TTYIN requests.

Errors:

None.

Example:

```
        .TITLE  GTLIN.MAC

;+
; .GTLIN - This is an example in the use of the .GTLIN request.
; The example merely accepts input from the console terminal and
; echoes it back.
;-
```

```
        .MCALL      .GTLIN,.PRINT,.EXIT

START:  .GTLIN      #BUFF,#PROMT        ;Get a line of input from keyboard
        TSTB        BUFF                ;Nothing entered?
        BEQ         1$                  ;Branch if nothing entered
        .PRINT      #BUFF               ;Echo the input back
        CLRB        BUFF                ;Clear first char of buffer
        BR          START               ;Go back for more
1$:     .EXIT                           ;Exit program on null input
BUFF:   .BLKW       41.                 ;80 character buffer (ASCIZ for .PRINT)
PROMT:  .ASCII      /Enter something>/<200>
        .END        START
```

2.37 .GVAL/.PVAL

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 and .PVAL must be used in an XM environment to read or change any fixed offset, and should be used with other RT–11 monitors for compatibility with XM and possible future releases of RT–11.

Chapter 3 of the *RT–11 Software Support Manual* contains a table of the monitor's fixed offset locations.

Macro Calls: .GVAL area, offset

.PVAL area, offset, value

where:

area — is the address of a two- or 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

Request Format for .GVAL:

R0 → area:

34	0
offset	

Request format for .PVAL:

R0 → area:

34	2
offset	
value	

Errors:

Code	Explanation
0	The offset requested is beyond the limits of the resident monitor.

Example:

```
          .TITLE  .GVAL.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

          CONFIG= 300     ;Offset in monitor of configuration word
          FJOB$=  200     ;Bit in config word is on if FG active

START:    .GVAL   #AREA, #CONFIG    ;Get monitor CONFIG word in R0
          BIT     #FJOB$,R0         ;See if FG Active bit is on
          BEQ     1$                ;Branch if not
          .PRINT  #FGACT            ;Announce FG is active
          .EXIT                     ;then exit program
1$:       .PRINT  #NOFG             ;Announce there's no FG Job
          .EXIT                     ;then exit program

AREA:     .BLKW   2                 ;EMT argument block
FGACT:    .ASCIZ  /! FG is active !/     ;FG active message
NOFG:     .ASCIZ  /? 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, .EXIT

          MAXBLK= 314               ;Monitor offset of default file size

START:    .PVAL   #EMTBLK,#MAXBLK,#NEWSIZ ;Change default file size to 100. blocks
          MOV     R0,     OLDSIZ    ;Save the old default
          .EXIT                     ;We'll just exit now, but presumably
                                    ;in a real program we'd do more
                                    ;processing, perhaps creating files
                                    ;with the new default size we just
                                    ;set, then before exiting we'd restore
                                    ;the old default size.

EMTBLK:   .BLKW   3                 ;EMT argument block
NEWSIZ:   .WORD   100.
OLDSIZ:   .WORD   0                 ;Old default size is saved here

          .END    START
```

2.38 .HERR/.SERR

.HERR and .SERR are complementary requests used to govern monitor behavior for serious error conditions. During program execution, certain error conditions can arise that cause the executing program to be aborted (see Table 2–2).

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. It is your responsibility to 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:

Table 2–2 contains a list of the errors that are 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 2–2: Soft Error Codes (.SERR)

Code	Explanation
–1	Called USR from completion routine.
–2	No device handler; this operation needs one.
–3	Error doing directory I/O.
–4	.FETCH error. Either an I/O error occurred while the handler was being used, or an attempt was made to load the handler over USR or RMON.
–5	Error reading an overlay.
–6	No more room for files in the directory.
–7	Invalid address (FB only); tried to perform a monitor operation outside the job partition.
–10	Invalid channel number; number is greater than actual number of channels that exist.
–11	Invalid EMT; an invalid function code has been decoded.
–12	Reserved for monitor internal use.
–13	Reserved for monitor internal use.
–14	Invalid directory.
–15	Unloaded XM handler.
–16	Reserved for monitor internal use.
–17	Reserved for monitor internal use.
–20	Reserved for monitor internal use.
–21	Reserved for monitor internal use.
–22	Reserved for monitor internal use.

Example:

```
          .TITLE  HERR.MAC
;+
; .HERR / .SERR - This is an example in the use of the .HERR & .SERR
; requests. Normally 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
          .CSISPC     #OUTSP,#DEFEXT          ;Use .CSISPC to get filespec
          .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:   NEG         R0                      ;Make error # positive
          DEC         R0                      ;Adjust by one
          ASL         R0                      ;Multiply by 2 to make an index
          MOV         TBL(R0),R0              ;Put message address into R0
          .PRINT                              ;and print it.
          BR          START                   ;Go try some more errors

TBL:      M1                                  ;Table of Error Message Addresses
          M1
          M2
          M3
          M4
          M5
          M6
          M7
          M10
          M11
          M12
          M13
          M14
          M15
          M16
          M17
          M20
          M21
          M22
                                              ;Error Messages...
M2:       .ASCIZ      /?Invalid Device -or- No Handler?/
M3:       .ASCIZ      /?Directory I-O Error?/
M7:       .ASCIZ      /?Address Check Error?/
M10:      .ASCIZ      /?Invalid Channel?/
M11:      .ASCIZ      /?Invalid EMT?/
M12:      .ASCIZ      /?Trap to 4?/
M13:      .ASCIZ      /?Trap to 10?/
M14:      .ASCIZ      /?Invalid directory?/
M17:      .ASCIZ      /?Memory error?/
M1:                                           ;Not possible in this program
M4:                                           ;Not possible in this program
M5:                                           ;Not possible in this program
M6:                                           ;Not possible in this program
M15:                                          ;Not possible in this program
M16:                                          ;Not possible in this program
M20:                                          ;Not possible in this program
M21:                                          ;Not possible in this program
M22:      .ASCIZ      /?Not Possible?/        ;Not possible in this program
NOFIL:    .ASCIZ      /?File Not Found?/
```

```
LUPOK:    .ASCIZ    /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)

          .END      START
```

2.39 .HRESET

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.) In an SJ environment, a hardware RESET instruction is used to terminate I/O. In an FB or XM environment, only the I/O associated with the job that issued the .HRESET is affected by entering active handlers at the abort entry point of the handler. All other transfers continue.

Macro Call: .HRESET

Errors:

None.

Example:

Refer to the example for .SRESET for format.

2.40 .INTEN

.INTEN is used by interrupt service routines to:

1. Notify the monitor that an interrupt has occurred and to switch to system state.
2. Set the processor priority to the correct value.
3. Save the contents of R4 and R5 before returning to the Interrupt Service Routine. Any other registers must be saved by you.

.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 the device should be run. On return from .INTEN, the device interrupt can be serviced, at which point the interrupt routine returns with an RTS PC.

NOTE

An RTI instruction does not return correctly from an interrupt routine that specifies an .INTEN.

Macro Call: .INTEN prio[,pic]

where:

prio — is the processor priority at which to run the interrupt routine, normally the priority at which the device requests an interrupt

pic — is an optional argument that should be non-blank if the interrupt routine is written as a PIC (position-independent code) routine. Any interrupt routine written as a device handler must be a PIC routine and must specify this argument

Errors:

None.

Example:

```
          .TITLE  SL11.MAC

;+
; SL11.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,SL11    ;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

SL11::    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$:       RETURN                             ;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
```

2.41 .LOCK/.UNLOCK

.LOCK

The .LOCK request keeps the USR in memory to provide any of its services required by your program. 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.) A .LOCK request always causes the USR to be loaded in memory if it is not already in memory. The USR is not released until an .UNLOCK request is given. (Note that under an FB monitor, calling the CSI or using a .GTLIN request can also perform an implicit and temporary .UNLOCK.) A program that has many USR requests to make can .LOCK the USR in memory, make all the requests, and then .UNLOCK the USR.

In an FB environment, a .LOCK inhibits the other job from using the USR. Note that the .LOCK request reduces time spent in file handling by eliminating the swapping of the USR in and out of memory. .LOCK causes the USR to be read into memory or swapped into memory. After a .LOCK has been executed, an .UNLOCK request must be executed to release the USR from memory. The .LOCK/.UNLOCK requests are complementary and must be matched. That is, if three .LOCK requests are issued, at least three .UNLOCK requests must be done, otherwise the USR is not released. More .UNLOCK than .LOCK requests can be issued without error.

Macro Call: .LOCK

Notes:

1. It is vital that the .LOCK call not come from within the area into which the USR will be swapped. If this should occur, the return from the .LOCK request would not be to the user program, but to the USR itself, since the .LOCK function inhibits the user program from being re-read. Also, none of the executable code should be in the area or reference anything in the area that the USR will occupy while it is locked.

2. Once a .LOCK has been performed, it is not advisable for the program to destroy the area the USR is in, even if no further use of the USR is required, because this causes unpredictable results when an .UNLOCK is done.

3. If a foreground job performs a .LOCK request while the background job owns the USR, foreground execution is suspended until the USR is available. In this case, it is possible for the background to lock out the foreground (see the .TLOCK request).

Errors:

None.

Example:

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 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 (see Note 1).

Macro Call: .UNLOCK

Notes:

1. 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.
2. With two running jobs in an FB environment use .LOCK/.UNLOCK pairs only where absolutely necessary. When a job locks the USR, the other job cannot use it until it is unlocked, which can degrade performance in some cases.
3. In an FB environment, calling the CSI with input coming from the console terminal results in an implicit (though temporary) .UNLOCK.
4. Make sure that the .UNLOCK request is not in the area that the USR swaps into. Otherwise, the request can never be executed.

Errors:

None.

Example:

```
          .TITLE  LOCK.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

          SYSPTR=54                       ;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$:       .PRINT      #LMSG               ;Print Error Message
          .EXIT                           ;then exit program
1$:       .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
          .EXIT                           ;and exit program

AREA:     .BLKW       3                   ;EMT Argument Block
FILE1:    .RAD50      /DK/                ;A File we're sure to find
          .RAD50      /PIP   /
          .RAD50      /SAV/
FILE2:    .RAD50      /DK/                ;Another file we might find
          .RAD50      /TECO  /
          .RAD50      /SAV/
```

```
LMSG:   .ASCIZ   /?Error on .LOOKUP?/ ;Error message
F1FND:  .ASCIZ   /...Found PIP.SAV/
F2FND:  .ASCIZ   /...Found TECO.SAV/
        .EVEN
        .END     START
```

2.42 .LOOKUP

A .LOOKUP request can be used in two different ways. The first way is to use the request as a standard lookup, which occurs under the SJ, FB, and XM monitors. The second way is to use the request when the system job feature is implemented. Both ways are described in this section.

2.42.1 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
.SAVESTATUS
.SRESET
.HRESET
.PURGE
.CSIGEN (if the channel is in the range 0–10 octal)

Note that if the program is overlaid, channel 17(octal) 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. 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 for the length.

NOTE

Care should be exercised when doing a non-file-structured .LOOKUP on a file-structured device, since if your program writes data, corruption of the device directory can occur and effectively destroy the disk. (The RT–11 directory starts in absolute block 6.)

In particular, avoid doing a .LOOKUP or .ENTER with a file specification where the file value is missing. If the device type is not known in advance and is to be entered from the keyboard, include a dummy file name with the .LOOKUP or .ENTER, even when it is assumed that the device is always non-file structured.

Macro Call: .LOOKUP area,chan,dblk[,seqnum]

where:

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

chan — is a channel number in the range 0–377(octal)

dblk — is the address of a four-word Radix–50 descriptor of the file to be operated upon

seqnum — is a file number for magtape and cassette

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. This means 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*.

Request Format:

R0 → area:

1	chan
dblk	
seqnum	

Errors:

Code	Explanation
0	Channel already open.
1	File indicated was not found on the device.

Example:

```
        .TITLE  LOOKUP.MAC

;+
; .LOOKUP - This is an example in the use of the .LOOKUP request.
; This example determines whether or not the RT-11 Device Queue
; Workfile exists on device DK: and if so, prints its size in
; blocks on the console terminal.
;-
```

```
          .MCALL      .LOOKUP,.PRINT,.EXIT
                                dblk =4w.
START:    .LOOKUP     #AREA,#0,#QUSPEC        ;See if there's a DK:QUFILE.TMP
          BCC         1$                      ;Branch if there is
          .PRINT      #NOFIL                  ;Print 'File Not Found' message
          .EXIT                               ;then exit program
1$:       MOV         #SIZE,R1                ;R1 => where to put ASCII size
          CALL        CNV10                   ;Convert size (in R0) to ASCII
          .PRINT      #BUFF                   ;Print size of QUFILE.TMP on console
          .EXIT                               ;then exit program
CNV10:    MOV         R0,-(SP)                ;Subroutine to convert Binary # in R0
          CLR         R0                      ;to Decimal ASCII by repetitive
1$:       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
QUSPEC:   .RAD50      /DK QUFILE/
          .RAD50      /TMP/
BUFF:     .ASCII      /DK:QUFILE.TMP = /
SIZE:     .ASCIZ      /    Blocks/
NOFIL:    .ASCIZ      /?File Not Found DK:QUFILE.TMP ?/
          .EVEN

          .END        START
```

2.42.2 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/C/W–.WRITE/C/W–.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/
.ASCII /logical-job-name/

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 zero, the channel will be opened for .READ/C/W requests only and such requests 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 already open.
1	No such job.

Example:

```
          .TITLE  SJLOOK.MAC

;+
; .LOOKUP - This is an example in the use of the .LOOKUP request
; to open a message channel to a System Job, specifically, the
; RT-11 Device Queue Foreground program. NOTE: This example assumes
; it will be run under an FB Monitor generated with System Job
; Support and that QUEUE.REL has been successfully FRUN/SRUN !!!
;-

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

START:    .LOOKUP     #AREA,#0,#QMSG          ;Try to open a channel to QUEUE
          BCC         1$                      ;Branch if successful
          .PRINT      #NOJOB                  ;Error...Print error message
          .EXIT                               ;then exit program
1$:       .WRITW      #AREA,#0,#RMSG,#6       ;Send a meaningless message to QUEUE
          BCS         2$                      ;Branch if error
          .READW      #AREA,#0,#RMSG,#6       ;Wait for an acknowledgment message
          BCS         2$                      ;Branch if error
          .PRINT      #QRUN                   ;Announce QUEUE alive and well
          .EXIT                               ;Then exit
2$:       .PRINT      #MSGERR                 ;Print error message
          .EXIT                               ;Then exit

AREA:     .BLKW       5                       ;EMT Argument Block
QMSG:     .RAD50      /MQ/                    ;Job Descriptor Block for .LOOKUP
          .ASCIZ      /QUEUE/
          .WORD       0,0
RMSG:     .WORD       0                       ;Dummy message...
          .ASCII      /SJLOOK/
MSGERR:   .ASCIZ      /?Message Error?/       ;Error Messages, etc.
NOJOB:    .ASCIZ      /?QUEUE is not running?/
QRUN:     .ASCIZ      /! QUEUE is alive and running !/
          .EVEN
          .END        START
```

2.43 .MAP (XM Only)

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. If the window is already mapped to another region, an implicit unmapping operation is performed (see the .UNMAP programmed request).

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 region is larger than the window, or the window would extend beyond the bounds of the region.

Example:

Refer to example for the .CRAW request.

2.44 .MFPS/.MTPS

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.

In the XM monitor, .MFPS and .MTPS can be used only by privileged jobs and are not available for use by virtual jobs.

Macro Call: .MFPS addr

where:

addr is the address into which the processor status is to be stored; if *addr* is not present, the value is returned on the stack. Note that only the priority bits are significant

The .MTPS call is used to set the priority bits.

Macro Call: .MTPS addr

where:

addr is either the value or the address of the value (depending on addressing mode) to be placed in the PSW. If *addr* is not present, the processor status word is taken from the stack. Note that the high byte on the stack is set to 0 when *addr* is present. If *addr* is not present, 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

Note:

It is possible to perform MTPS and MFPS operations and access the condition codes by following this special technique:

1. In the beginning of your program, set up the IOT trap vector as follows:

```
        .ASECT                  ;SET UP IOT
. = 20
        .WORD   GETPS           ;IOT SERVICE ADDRESS IN 'MFPS' SUBROUTINE
        .WORD   340             ; PRIORITY 7
```

2. 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
```

3. To get the PSW or to set the PSW to a desired value, follow this sequence of instructions:

```
;+
; TO GET PS ...
;-

        JSR     PC,MFPS         ;GET PS
                                ;CONTINUE, PS IS ON STACK ...


;+
; TO PUT PS ...
;-

        MOV     NEWPS,-(SP)     ;PUT DESIRED PS ON STACK ...
        JSR     PC,MTPS         ;CALL MTSP
                                ;CONTINUE PROCESS W/ NEW PS ...
```

Errors:

None.

Example:

```
          .TITLE  MFPS

;+
; .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

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$:       NOP                             ;Program continues
          NOP                             ;
          .PRINT      #GOT1               ;Announce success
2$:       .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       #340                ;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      /?No more Queue Elements Available?/
GOT1:     .ASCIZ      /Element acquired...press any key to continue/

          .END        START
```

2.45 .MRKT (FB and XM; SJ Monitor Special Feature)

The .MRKT request schedules a completion routine to be entered after a specified time interval (measured in clock ticks) has elapsed. The .MRKT request is an optional feature in the SJ monitor, and is selected as a system generation option.

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). The user should allocate enough queue elements to handle at least as many mark time and I/O requests as are expected to be pending simultaneously.

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 not be within the range 177000–177777, which is 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  TREAD.MAC

;+
; .MRKT/.CMKT - This is an example in 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,.TTYOUT,.CMKT,.TWAIT,.QSET

          LF = 12                         ;Line Feed
          JSW = 44                        ;Job Status Word location
          TCBIT$ = 100                    ;Return C-bit bit in JSW
          TTSPC$ = 10000                  ;TTY Special Mode bit in JSW

START:    .QSET       #XQUE,#1            ;Need an extra Q-Elem for this
1$:       MOV         #PROMT,R0           ;Mainline - R0 => Prompt
          MOV         #BUFFR,R1           ;R1 => Input buffer
          CALL        TREAD$              ;Do a "timed read"
          BCS         2$                  ;C-bit set = Timed out
          .PRINT      #LINE               ;"Process" data...
          BR          1$                  ;Go back for more
2$:       .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$:   TST       R0                      ;See if we have to prompt
          BEQ       1$                      ;Branch if no...
          .PRINT                            ;Output prompt
1$:       CLR       TBYT                    ;Clear time-out flag
          .MRKT     #TAREA,#TIME,#TOUT,#1   ;Issue a .MRKT for 10 sec
          BIS       #TCBIT$,@#JSW           ;Set C-Bit bit in JSW (for F/B)
          CLRB      @R1                     ;Start with "empty" buffer
TTIN:     .TWAIT    #AREA                   ;Wait so we don't lock out BG
          .TTINR                            ;Look for a character
          BIT       #1,(PC)+                ;Timed out?
TBYT:     .WORD     0                       ;Time-out flag
          BNE       2$                      ;Branch if yes
          BCS       TTIN                    ;Branch if input not complete
          MOVB      R0,(R1)+                ;Xfer 1st character
          .CMKT     #TAREA,#0               ;Cancel .MRKT
2$:       BIS       #TTSPC$,@#JSW           ;Turn on TT: Special mode
3$:       .TTINR                            ;Flush TT: ring buffer
          MOVB      R0,(R1)+                ;putting characters in user buffer
          BCC       3$                      ;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
          RETURN                            ;Leave completion code
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    /Not in stock - Part # / ;Dummy response
BUFFR:    .BLKB     81.                     ;User input buffer
PROMT:    .ASCIZ    /Enter Part # >/<200>   ;Prompt
TIMOUT:   .ASCIZ    /Timed read expired!/   ;Too bad message
          .END      START
```

2.46 .MTATCH (Special Feature)

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. If .MTATCH request fails because the terminal is owned by another job, the job number of the owner is returned in R0.

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 special feature that you can select during the system generation process.)
- 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:

R0 → area:

37	5
addr	
0	unit

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 the XM monitor, the optional status word address is not in valid user virtual address space.

Example:

```
;+
; MTXAMP.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

MTXAMP:                                ;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$:     ADD      #2,R2                ;R2 -> Next AST word
         INC      R1                   ;Get next LUN
         CMP      R1,R4                ;Done?
         BLO      10$                  ;Nope, go attach another
```

```
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$:    ADD     #2,R2                       ;Point to next AST word
        INC     R1                          ;Get next LUN
        CMP     R1,R4                       ;Done them all?
        BLO     10$                         ;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:  .ASCII  <33>"H"<33>"J"              ;VT52 Home + Erase to EOS
        .ASCIZ  /Hello! Characters typed will be echoed/
NOMTTY: .ASCIZ  /?Not multiterminal system?/
UNEXP:  .ASCIZ  /?Unexpected error...program aborting?/

        .END    MTXAMP                      ;End of program
```

2.47 .MTDTCH (Special Feature)

The .MTDTCH request detaches a terminal from one job and makes it available for other jobs. When a terminal is detached, it is deactivated, and unsolicited interrupts are ignored. Input is disabled immediately, but any characters in the output buffer are printed. Attempts to detach a terminal attached by another 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	Invalid unit number, unit not attached.
2	Nonexistent logical unit number.
3	Invalid request; function code out of range.

Example:

```
          .MCALL  .MTDTCH,.MTPRNT,.MTATCH,.EXIT,.PRINT

START:
          .MTATCH #MTA,#0,#3          ;ATTACH TO LUN 3
          BCS     1$                  ;ATTACH ERROR
          .MTPRNT MTA,#MESS,#3        ;PRINT MESSAGE
          .MTDTCH #MTA,#3             ;DETACH LUN 3
          .EXIT
1$:       .PRINT  #ATTERR             ;ATTACH ERROR
                                      ;(PRINTED ON CONSOLE)
          .EXIT
ATTERR:   .ASCIZ/ATTACH ERROR/
MESS:     .ASCIZ/DETACHING TERMINAL/
          .EVEN
MTA:      .BLKW   3
          .END    START
```

2.48 .MTGET (Special Feature)

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 is returned in R0. You do not need to do an .MTATCH before using the .MTGET request.

Macro Call: .MTGET area,addr,unit

where:

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

addr — is the address of a four-word status block where the status information is returned

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. In any other error condition, the contents of R0 are undefined

Request Format:

R0 → area:

37	1
addr	
—	unit

The status block has the following structure:

addr→

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

Note that if an error occurs, and the error code is not 1 or 4, the status block will not have been modified.

NOTE

Use the Bit Set (BIS) and Bit Clear (BIC) instructions instead of Move (MOV) and Clear (CLR) instructions when setting terminal and line characteristics. This avoids changing other bits inadvertently.

The bit definitions for terminal configuration word 1 (M.TSTS) are as follows:

Value	Bit	Meaning
1	0	Terminal has hardware tab
2	1	Output RET/LF when carriage width exceeded
4	2	Terminal has hardware form feed
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)
100	6	Inhibit TT wait (similar to bit 6 in the Job Status Word)
200	7	Enable CTRL/S–CTRL/Q processing
7400	8–11	Line speed (baud rate) mask. Bits 8 through 11 indicate the terminal baud rate (DZ11 and DZV11 only). The values are as follows:

Octal Value of Line Speed Mask (M.TSTS bits 11–8)	Baud Rate
0000	50
0400	75
1000	110
1400	134.5
2000	150
2400	300

Octal Value of Line Speed Mask (M.TSTS bits 11–8)	Baud Rate
3000	600
3400	1200
4000	1800
4400	2000
5000	2400
5400	3600
6000	4800
6400	7200
7000	9600
7400	(unused)

10000	12	Character mode input (similar to bit 12 in the Job Status Word)
20000	13	Terminal is remote (Read-only bit)
40000	14	Lowercase to uppercase conversion disabled
100000	15	Use backspace for rubout (video type display)

The bit definitions for terminal configuration word 2 (M.TST2) are as follows:

Value	Bit	Meaning
3	0–1	Character length, which can be 5(00), 6(01), 7(10), or 8(11) bits (DZ only)
4	2	Unit stop, which sends one stop bit when clear, two stop bits when set (DZ only)
10	3	Parity enable (DZ only)
20	4	Odd parity when set; even parity when clear
140	5–6	Reserved
200	7	Read pass all
77400	8–14	Reserved
100000	15	Write pass all

The bit definitions for terminal status byte (M.TSTW) are as follows:

Value	Bit	Meaning
2000	10	Terminal is shared console
4000	11	Terminal has hung up
10000	12	Terminal interface is DZ11
40000	14	Double CTRL/C was struck (the .MTGET request resets this bit in the terminal control block if it is on)
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.

Example:

Refer to the example for the .MTATCH request.

2.49 .MTIN (Special Feature)

The .MTIN request reads characters from the keyboard buffer. 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 specified by you. The terminal must be attached. An updated user buffer address is returned in R0 if the request is successful. If bit 6 is set in the M.TSTS word (see the MTSET request), the .MTIN request returns immediately with the carry bit set (code 0) if there is no input available. Operation is similar for the system console if bit 6 is set in the JSW. If bit 12 in M.TSTS is clear, no line is available; if bit 12 is set, there are no characters in the buffer. If these conditions do not exist, the .MTIN request waits until input is available, and the job is suspended until input is available.

The meaning of bits 6 and 12 in the terminal configuration word (M.TSTS) for the programmed request .MTIN is as follows:

Bit 6	Bit 12	Meaning
0	0	Normal mode of input (echo provided); wait for line
1	0	Carry bit set: no line available
1	1	Carry bit set: no character available; no echo provided
0	1	No echo provided

If a multiple-character request was made and the number of characters requested are not available, the request can either wait for the characters to become available, or it can return with a partial transfer. If bit 6 of M.TSTS is clear, the request waits for more characters. If bit 6 is set, the request returns with a partial transfer. In the latter case, R0 contains the updated buffer address (pointing past the last character transferred), the C bit is set, and the error code is 0.

The .MTIN request has the following form:

Macro Call: .MTIN area,addr,unit[,chrcnt]

where:

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

addr is the byte address of the user buffer

unit is the logical unit number of the terminal input

chrcnt is a character count indicating the number of characters to transfer. The valid range is from 1 to 255(decimal). A character count of zero means one character

Request Format:

R0 → area:

37	2
addr	
chrcnt	unit

Errors:

Code	Explanation
0	No input available — bit 6 is set in the Job Status Word (for the system console) or in M.TSTS by the .MTSET request.
1	Invalid unit number, unit not attached.
2	Nonexistent logical unit number.
3	Invalid request; function code out of range.
5	In the XM monitor, the user buffer address is not in valid user virtual address space.

Example:

Refer to the example for the .MTATCH request.

2.50 .MTOUT (Special Feature)

The .MTOUT request transfers characters to the terminal output buffer. This request is the multiterminal form of the .TTYOUT request. The .MTOUT request moves one or more characters from the user's buffer to the output ring buffer of the terminal. The terminal must be attached. An updated user buffer address is returned in R0 if the request is successful. When there is no room in the output ring buffer, the carry bit is set and an error code of 0 is returned in byte 52 if bit 6 is set in M.TSTS. Otherwise, the job is suspended until room becomes available.

If a multiple-character request was made and there is not enough room in the output ring buffer to transfer the requested number of characters, the request can either wait for enough room to become available, or it can return with a partial transfer. If bit 6 in M.TSTS is clear, the request waits until it can complete the full transfer. If bit 6 is set, the request returns with a partial transfer. In the latter case, R0 contains the updated buffer address (pointing past the last character transferred), the C bit is set, and the error code is 0.

The meaning of bit 6 in the terminal configuration word (M.TSTS) for the .MTOUT request is as follows:

Bit 6	Meaning
0	Normal mode for output; wait for room in buffer
1	Carry bit set: no room in output ring buffer

Macro Call: .MTOUT area,addr,unit[,chrcnt]

where:

area	is the address of a three-word EMT argument block
addr	is the address of the caller's input buffer
unit	is the unit number of the terminal
chrcnt	is a character count indicating the number of characters to transfer. The valid range is from 1 to 255(decimal)

Request Format:

R0 → area:

37	3
addr	
chrcnt	unit

Errors:

Code	Explanation
0	No room in output buffer.
1	Invalid unit number, unit not attached.
2	Nonexistent logical unit number.
3	Invalid request; function code out of range.
5	In the XM monitor, the user buffer address is not in valid user virtual address space.

Example:

Refer to the example for the .MTATCH request.

2.51 .MTPRNT (Special Feature)

This .MTPRNT request allows one or more lines to be printed at the specified terminal in a multiterminal environment. It is equivalent to the .PRINT request (see .MTSET request for more details). The string to be printed must be terminated with a null byte or a 200 byte, similar to the string used with the .PRINT request as follows:

```
.ASCIZ /string/
```

or

```
.ASCII /string/<200>
```

The null byte causes a carriage return/line feed combination to be printed after the string. The 200 byte suppresses the carriage return/line feed combination and leaves the carriage positioned after the last character of the string. 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	7
addr	
—	unit

Errors:

Code	Explanation
1	Invalid unit number, unit not attached.
2	Nonexistent logical unit number.
3	Invalid request; function code out of range.
5	In the XM monitor, the character string address is not in valid user virtual address space.

Example:

Refer to the example for the .MTATCH request.

2.52 .MTPS

See .MFPS/.MTPS (Section 2.44).

2.53 .MTRCTO (Special Feature)

The .MTRCTO request resets the CTRL/O switch of the specified terminal and enables terminal output in a multiterminal environment. It is equivalent to 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	Invalid unit number, unit not attached.
2	Nonexistent logical unit number.
3	Invalid request; function code out of range.

Example:

Refer to the example for the .MTATCH request.

2.54 .MTSET (Special Feature)

This multiterminal request sets terminal and line characteristics. It also determines the input/output mode of the terminal service requests for the specified terminal.

Macro Call: .MTSET area,addr,unit

where:

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

addr is the address of a four-word status block containing the line and terminal status being requested

unit is the logical unit number associated with the line and terminal

Request Format:

R0 → area:

37	1
addr	
—	unit

When the program returns from the request, the status block contains the following information:

Byte Offset	Contents
0	Terminal configuration word 1 (The bit definitions are the same as those for the .MTGET request.)
2	Terminal configuration word 2 (The bit definitions are the same as those for the .MTGET request.)
4	Character requiring fillers
5	Number of fillers
6	Carriage width (byte)

NOTE

The .MTSET request sets all of the parameters listed above. The recommended procedure for using .MTSET is: (1) precede it by an .MTGET request; (2) use BIS and BIC instructions to set or clear bit fields (modify 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 will not have been modified.

Errors:

Code	Explanation
1	Invalid unit number, *lun* not attached.
2	Nonexistent logical unit number.
3	Invalid request, function code out of range.
5	In the XM monitor, the status block address is not in valid user virtual address space.

Example:

Refer to the example for the .MTATCH request.

2.55 .MTSTAT (Special Feature)

The .MTSTAT 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. The status block contains the following information:

Byte Offset	Contents
0	Offset from the base of the resident monitor to the first terminal control block (TCB)
2	Offset from the base of the resident monitor to the terminal control block of the console terminal for the program
4	The value (0–16 decimal) of the highest logical unit number (LUN) built into the system
6	The size of the terminal control block in bytes
10–17	Reserved

Request Format:

R0 → area:

37	10
addr	
0	

Errors:

Code	Explanation
3	Invalid request; function code out of range
5	In XM, the status block address is not in valid user address space.

Example:

Refer to the example for the .MTATCH request.

2.56 .MWAIT (FB and XM Only)

This request is similar to the .WAIT request. .MWAIT, however, 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. It should be used with the .RCVD or .SDAT modes of message handling, where no action is taken when a message is completed.

Macro call: .MWAIT

Request Format:

R0 =

11	0

Errors:

None.

Example:

```
;+
; .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     MWAITF.MAC
;+
; Foreground Program ...
;-

          .MCALL     .RCVD,.MWAIT,.PRINT,.EXIT

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    /Hi - FG alive and well and waiting for a message!/
FMSG:     .ASCIZ    /Hey BG - Got your message it reads:/
          .END      MWAITF

          .TITLE    MWAITB.MAC
;+
; Background Program - Send 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       1$                       ;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
1$:       PRINT     #NOFG                    ;No FG !
          .EXIT                              ;Exit program
AREA:     .BLKW     5                        ;EMT Argument Block
BUFF:     .BLKW     40.                      ;Up to 80 char message
PROMT:    .ASCII    /Enter message to be sent to FG Job/<15><12>/>/<200>
NOFG:     .ASCIZ    /?No FG?/
          .END      MWAITB
```

2.57 .PEEK/.POKE

The .PEEK programmed request returns in R0 the contents of a memory location; .POKE changes the contents of a memory location. The .POKE request also returns the old contents of the memory location in R0 to simplify the saving and restoring of a location. .PEEK and .POKE must be used in an XM environment to change memory locations that are not defined as monitor fixed offsets, and should be used with all RT–11 monitors for compatibility.

Although .PEEK and .POKE may seem very similar to .GVAL and .PVAL, respectively, they are different in the way they refer to locations. .GVAL and .PVAL access only monitor fixed offsets. All offsets used by .GVAL and .PVAL are calculated relative to the base of the resident monitor. Addresses used by .PEEK and .POKE, on the other hand, are simply memory addresses. Although .PEEK and .POKE can be used to access monitor fixed offsets, this requires that you find the base address of RMON, add the offset value, and use the resulting address as an argument to .PEEK or .POKE.

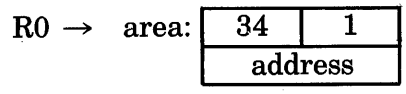
Macro Calls: .PEEK area,address

.POKE area, address,value

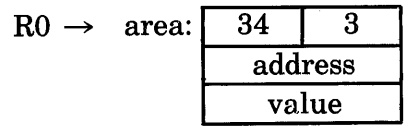
where:

area	is the address of a two- or three-word EMT argument block
address	is the address of the location to examine or change
value	is the new contents to place in the location

Request Format for .PEEK:



Request Format for .POKE:



Errors:

None.

Example:

```
;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, .EXIT

          RMON=   54
          MAXBLK= 314

START:    .PEEK   #EMTBLK,#RMON           ;Pick up base of RMON from loc. 54
          ADD     #MAXBLK, R0             ;Add fixed offset of default file size,
          MOV     R0,     R1
          .POKE   #EMTBLK, R1, #NEWSIZ    ;Set a new default file size, return old
          MOV     R0,     OLDSIZ          ;default file size in R0 and save the old size.
          .EXIT                           ;We'll just exit now, but presumably
                                          ;in a real program we'd do more
                                          ;processing, perhaps creating files
                                          ;with the new default size we just set, then
                                          ;before exiting we'd restore the old
                                          ;default size.
                                          ;EMT area
EMTBLK:   .BLKW   3                       ;EMT area
NEWSIZ:   .WORD   100.
OLDSIZ:   .WORD   0                       ;The old default size is saved here.

          .END    START
```

2.58 .POKE

Refer to .PEEK/.POKE (Section 2.57).

2.59 .PRINT

The .PRINT request causes output to be printed at the console terminal. The string to be printed can be terminated with either a null (0) byte or a 200 byte. If the null (ASCIZ) format is used, the output is automatically followed by a carriage return/line feed combination. If a 200 byte terminates the string, no carriage return/line feed combination is generated.

Control returns to the user program after all characters have been placed in the output buffer.

When a foreground job is running and the job that is producing output changes, a B> or F> appears. Any text following the message has been printed by the job indicated (foreground or background) until another B> or F> is printed.

When a system job prints a message to 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. Thus, for urgent messages, the .PRINT request should be used (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 carriage return/line feed combination, the console belongs to the background and .TTYOUTs from the foreground are not printed until a carriage return is typed to the background. A foreground job can force its message through by doing a .PRINT instead of the .TTYOUT.

Macro Call: .PRINT addr

where:

addr is the address of the string to be printed

Errors:

None.

Example:

```
          .TITLE  PRINT.MAC

;+
; .PRINT - This is an example in the use of the .PRINT request.
; The example merely accepts input from the console terminal and
; echoes it back.
;-

          .MCALL      .GTLIN,.PRINT,.EXIT

START:    .GTLIN      #BUFF,#PROMT        ;Get a line of input from keyboard
          TSTB        BUFF                ;Nothing entered?
          BEQ         1$                  ;Branch if nothing entered
          .PRINT      #BUFF               ;Echo the input back
          CLRB        BUFF                ;Clear first char of buffer
          BR          START               ;Go back for more
1$:       .EXIT                           ;Exit program on null input
BUFF:     .BLKW       41.                 ;80 character buffer (ASCIZ for .PRINT)
PROMT:    .ASCII      /Enter something/<15><12>/>/<200>
          .END        START
```

2.60 .PROTECT/.UNPROTECT (FB and XM Only)

.PROTECT
The .PROTECT request allows a job to obtain exclusive control of a vector (two words) in the region 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

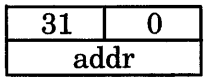
addr is the address of the word pair to be protected

NOTE

The argument *addr* must be a multiple of four, and must be less than or equal to 474(octal). The two words at *addr* and *addr*+2 are protected.

Request Format:

R0 → area:



Errors:

Code	Explanation
0	Protect failure; locations already in use.
1	Address (*addr*) is greater than 474 or is not a multiple of 4.

Example:

```
          .TITLE  PROTEC.MAC

;+
; .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

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...
        .ASCIZ  /Hello DL11... Are You There ??/
        .ENDR
NOVEC:  .ASCIZ  /?Vector already protected?/ ;Error message text

        .END    START
```

.UNPROTECT

The .UNPROTECT request is the complement of the .PROTECT request. It cancels any protected vectors in the 0 to 476 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(octal)

Request Format:

R0 → area:

31	1
addr	

Errors:

Code	Explanation
1	Address (*addr*) is greater than 474(octal) or is not a multiple of four.

Example:

Refer to the example for the .PROTECT request.

2.61 .PURGE

The .PURGE request deactivates a channel without performing a .HRESET, .SRESET, .SAVESTATUS, or .CLOSE request. .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 an inactive channel is ignored.

NOTE

Do not purge channel 17(octal) 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 freed

Request Format:

R0 =

3	chan

Errors:

None.

Example:

```
          .TITLE      PURGE.MAC

;+
; .PURGE - This is an example in the use of the PURGE request.
; This example merges 2-6 files into 1 file, making use of .SAVESTATUS
; and .REOPEN to read all input files on one channel. The .PURGE request
; is used to free the input channel after each transfer.
;-

          .MCALL      .CSIGEN,.SAVESTATUS,.REOPEN,.CLOSE,.EXIT
          .MCALL      .READW,.WRITW,.PRINT,.PURGE
          ERRBYT      = 52                    ;Error byte loc in SYSCOM

START:    .CSIGEN     #DSPACE,#DEFEXT         ;Get file spec,open files,load handlers
          MOV         #3,R4                   ;R4 = 1st input channel
          MOV         #AREA,R3                ;R3 => EMT Argument block
          MOV         #SAVBLK,R5              ;R5 => Channel savestatus blocks
1$:       .SAVEST     R3,R4,R5                ;Save channel status
          BCS         2$                      ;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         1$                      ;Branch if not
2$:       MOV         #SAVBLK,R5              ;R5 => to 1st saved channel status
          BEQ         7$                      ;Branch if no input files
4$:       .REOPEN     R3,#3,R5                ;Re-open input channel on Ch 3
          CLR         BLK                     ;Start reading with block 0
5$:       .READW      R3,#3,#BUFFR,#256.,BLK  ;Read a block
          BCC         6$                      ;Branch if no error
          TSTB        @#ERRBYT                ;Check if error = EOF
          BNE         8$                      ;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         4$                      ;Branch if yes
          .CLOSE      #0                      ;We're done...close output channel
          .PRINT      #DONE                   ;Announce merge complete
          .EXIT                               ;Exit program
6$:       .WRITW      R3,#0,#BUFFR,#256.,WBLK ;Write block just read
          INC         WBLK                    ;Bump to next output block
          INC         BLK                     ;same for input blk (doesn't affect C bit)
          BCC         5$                      ;Branch if no error on write
          MOV         #WERR,R0                ;Write error - R0 => message
          BR          9$                      ;merge...
7$:       MOVE        #NOINP,R0               ;R0 => No input files message
          BR          9$                      ;merge...
8$:       MOV         #RERR,R0                ;R0 => Read error message
9$:       .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      /?No input files?/      ;Error messages
WERR:     .ASCIZ      /?Write Error?/
RERR:     .ASCIZ      /?Read Error?/
DONE:     .ASCIZ      /I-O Transfer Completed/
          .EVEN
BUFFR:    .BLKW       256.                    ;I/O buffer
DSPACE    = .                                 ;Handlers start here...

          .END        START
```

2.62 .PVAL

See .GVAL/.PVAL (Section 2.37).

2.63 .QELDF (Device Handler Only)

The .QELDF macro symbolically defines queue element offsets for the specified set of system generation special features. The queue element offsets generated by this macro are as follows:

```
Q.LINK=0        (Link to next queue element)
Q.CSW=2.        (Pointer to channel status word)
Q.BLKN=4.       (Physical block number)
Q.FUNC=6.       (Special function code)
Q.JNUM=7.       (Job number)
Q.UNIT=7.       (Device unit number)
Q.BUFF=^O10     (User virtual memory buffer address)
Q.WCNT=^O12     (Word count)
Q.COMP=^O14     (Completion routine code)
```

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

```
Q$LINK=-4
Q$CSW=-2
Q$BLKN=0
Q$FUNC=2
Q$JNUM=3
Q$UNIT=3
Q$BUFF=4
Q$WCNT=6
Q$COMP=^O10
```

For SJ and FB systems:

```
Q.ELGH=^O16     (End of queue element; used to find length)
```

For XM systems:

```
Q.PAR=^O16      (PAR1 relocation bias)
Q$PAR=^O12
Q.ELGH=^O24     (End of queue element; used to find length)
```

Example:

Refer to the example following the description of .DRAST.

2.64 .QSET

The .QSET request allows additional entries to be made to the RT–11 I/O queue. A general rule to follow is that each program should contain one more queue element than the total number of I/O requests that will be active simultaneously on different channels. Timing and message requests such as .MRKT, .TWAIT, .SDAT/C, and .RCVD/C also require queue elements and must be considered when allocating queue elements for a program. 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.

Each time .QSET is called, a specified contiguous area of memory is divided into seven-word segments (10-word [decimal] segments for the XM monitor) and is added to the queue for that job. .QSET can be called as many times as required. The queue set up by multiple .QSET requests is a linked list. Thus, .QSET need not be called with strictly contiguous arguments. The space used for the new elements is allocated from your program space. Care must be taken so that the program in no way alters the elements once they are set up. The .SRESET and .HRESET requests discard all user-defined queue elements; therefore any 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.

Care should also be taken to allocate sufficient memory for the number of queue elements requested. The elements in the queue are altered asynchronously by the monitor; if enough space is not allocated, destructive references occur in an unexpected area of memory. 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 XM 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(octal).

NOTE

Programs that are to run in XM as well as SJ or FB environments should allocate 10(decimal) 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.

The following programmed requests require queue elements:

.TWAIT	.RCVDW
.MRKT	.WRITE
.READ	.WRITC
.READC	.WRITW
.READW	.SDAT
.RCVD	.SDATC
.RCVDC	.SDATW

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 SJ and FB monitor, each queue entry is seven words long; hence the space set aside for the queue should be *len**7 words. In the XM monitor, 10(decimal) words per queue element are required

On completion, R0 contains the address of the first word beyond the allocated queue elements.

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:

```
          .TITLE  QSET.MAC
;+
; .QSET - This is an example in the use of the .QSET request.
; The example illustrates a user implemented "Timed Read" to cancel an
; input request after a specified time interval.
;-
          .MCALL      .MRKT,.TTINR,.EXIT,.PRINT,.TTYOUT,.CMKT,.TWAIT,.QSET

          LF = 12                           ;Line Feed
          JSW = 44                          ;Job Status Word location
          TCBIT$ = 100
;Return C-bit bit in JSW
          TTSPC$ = 10000                    ;TTY Special Mode bit in JSW

START:    .QSET       #XQUE,#1              ;Need an extra Q-Elem for this
1$:       MOV         #PROMT,R0             ;Mainline - R0 => Prompt
          MOV         #BUFFR,R1             ;R1 => Input buffer
          CALL        TREAD$                ;Do a "timed read"
          BCS         2$                    ;C-bit set = Timed out
          .PRINT      #LINE                 ;"Process" data...
          BR          1$                    ;Go back for more
2$:       .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$:   TST         R0                    ;See if we have to prompt
          BEQ         1$                    ;Branch if no...
          .PRINT                            ;Output prompt
1$:       CLR         TBYT                  ;Clear time-out flag
          .MRKT       #TAREA,#TIME,#TOUT,#1 ;Issue a .MRKT for 10 sec
          BIS         #TCBIT$,@#JSW         ;Set C-Bit bit in JSW (for F/B)
          CLRB        @R1                   ;Start with "empty" buffer
TTIN:     .TWAIT      #AREA                 ;Wait so we don't lock out BG
          .TTINR                            ;Look for a character
          BIT         #1,(PC)+              ;Timed out?
TBYT:     .WORD       0                     ;Time-out flag
          BNE         2$                    ;Branch if yes
          BCS         TTIN                  ;Branch if input not complete
          MOVB        R0,(R1)+              ;Xfer 1st character
          .CMKT       #TAREA,#0             ;Cancel .MRKT
2$:       BIS         #TTSPC$,@#JSW         ;Turn on TT: Special mode
3$:       .TTINR                            ;Flush TT: ring buffer
          MOVB        R0,(R1)+              ;putting characters in user buffer
          BCC         3$                    ;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
          RETURN                            ;Leave completion code
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      /Not in stock - Part # / ;Dummy response
BUFFR:    .BLKB       81.                   ;User input buffer
PROMT:    .ASCIZ      /Enter Part # >/<200> ;Prompt
TIMOUT:   .ASCIZ      /Timed read expired!/ ;Too bad message
          .END        START
```

2.65 .RCTRLO

The .RCTRLO request makes sure that the console terminal is able to print by resetting the CTRL/O switch for the terminal. A CTRL/O typed while output is directed to the console terminal inhibits the output from printing until either another CTRL/O is struck or the program resets the CTRL/O switch. Therefore, a program with a message that must appear at the console should reset the CTRL/O switch.

A program should also 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:

None.

Example:

```
          .TITLE  RCTRLO.MAC
;+
; .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,.EXIT

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
```

2.66 .RCVD/.RCVDC/.RCVDW (FB and XM Only)

The .RCVD (receive data) request allows a job to read messages or data sent by another job in an FB environment.

There are three forms of the .RCVD request, and they 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 an FB monitor with the system job feature, .RCVD/C/W requests and .SDAT/C/W requests remain valid for sending messages between background and foreground jobs in addition to the general read and write capability available to all jobs.

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

This request is used to receive data and continue execution. The request is posted and the issuing job continues execution. When the job needs to have the transmitted message, an .MWAIT should be executed. This causes the job to be suspended until all .SDATx and .RCVDx requests for the job are complete.

Macro Call: .RCVD area,buf,wcnt

where:

area is the address of a five-word EMT argument block

buf is the address of the buffer into which the message length/message data is to be placed

wcnt is the number of words to be transferred

Request Format:

R0 → area:

26	0
reserved	
buf	
wcnt	
1	

Upon completion of the .RCVD, the first word of the message buffer contains the number of words transmitted. 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.

The word count is a variable number, and as such, the .SDAT/.RCVD combination can be used to transmit a few words or entire buffers. The .RCVD operation is only complete when a .SDAT is issued from the other job.

Programs using .RCVD/.SDAT must be carefully designed to either always transmit/receive data in a fixed format or to have the capability of handling

variable formats. 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 in the FB 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. 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

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

crtn is the address of a completion routine to be entered

As in the .RCVD request, word 0 of the buffer contains the number of words transmitted when the transfer is complete.

Request Format:

R0 → area:

26	0
reserved	
buf	
wcnt	
crtn	

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  RCVDC.MAC

;+
; .RCVDC - This is an example of the .RCVDC request. The example
; is a simulation of a mainline Foreground program which is currently
; suspended waiting for a message from the Background, but which needs
; to close a file (perhaps opened by a .ENTER ?) before aborting from
; CTRL-C action. A completion routine periodically inspects the CTRL-C
; status word and resumes the mainline if double CTRL-C is entered.
; NOTE: This example MUST be run as a FG job under an FB monitor.
;-

          .MCALL    .SCCA,.RCVDC,.EXIT,.PRINT,.MRKT
          .MCALL    .QSET,.SPND,.RSUM

START:    .QSET     #QELEM,#1                   ;Allocate another Q-Element
          .SCCA     #MAREA,#SCCA                ;Inhibit ^C^C action by monitor
1$:       CALL      CWATCH                      ;Start "watchdog" completion rtne
          .RCVDC    #MAREA,#MBUFF,#40.,#MESG    ;Look for a message
          ;         .                           ;No errors - there's always BG
          ;         .                           ;Other processing here...
          ;         .                           ;
          .PRINT    #SLEEP                      ;Announce we're going to suspend
          .SPND                                 ;Suspend to wait for message
          TST       SCCA                        ;We've been .RSUMed...^C^C hit??
          BNE       CLOSE                       ;Branch if yes
          ;         .                           ;otherwise assume message came in...
          ; <Process message here>
          ;         .
          BR        1$                          ;Loop...

CWATCH:   TST       SCCA                        ;Check if ^C^C entered...
          BEQ       MARK                        ;Branch if no
MESG:               .RSUM                       ;Yes...wake up the mainline
          RETURN                                ;then leave completion code
MARK:     .MRKT     #CAREA,#TIME,#CWATCH,#1     ;Schedule to run again in 10 sec.
          RETURN                                ;then leave completion code

CLOSE:    .PRINT    #ABORT                      ;Announce we're aborting
          ;         .                           ;Proceed with "orderly" abort
          ; <Output file(s) closed here>
          ;         .
          .EXIT                                 ;Exit the program

QELEM:    .BLKW     7                           ;Extra Q-Element
MBUFF:    .BLKW     41.                         ;Message buffer
MAREA:    .BLKW     5                           ;EMT Argument blocks
CAREA:    .BLKW     4                           ;
TIME:     .WORD     0,600.                      ;Time out in 10 seconds
SCCA:     .WORD     0                           ;^C^C Status word

ABORT:    .ASCIZ    /?! Abort Acknowledged...Closing Output File(s) !?/
SLEEP:    .ASCIZ    /! Mainline Suspending !/

          .END      START
```

.RCVDW

.RCVDW is used to receive data and wait. 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

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

Request Format:

R0 → area:

26	0
reserved	
buf	
wcnt	
0	

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 .SDATW request.

2.67 .RDBBK (XM Only)

The .RDBBK macro defines symbols for the region definition block and reserves space for it. The .RDBBK automatically invokes .RDBDF.

Macro Call: .RDBBK rgsiz

where:

rgsiz is the size of the dynamic region needed (expressed in 32-word units)

Example:

See Chapter 4 of the *RT–11 Software Support Manual* for an example that uses the .RDBBK macro and a detailed description of the extended memory feature.

2.68 .RDBDF (XM Only)

The .RDBDF macro defines the symbolic offset names for the region definition block and the names for the region status word bit patterns. In addition, this macro also defines the length of the region definition block by setting up the following symbol:

R.GLGH = 6

The .RDBDF macro does not reserve space for the region definition block.

Macro Call: .RDBDF

The .RDBDF macro expands as follows:

```
R.GID    = 0
R.GSIZ   = 2
R.GSTS   = 4
R.GLGH   = 6
RS.CRR   = 100000
RS.UNM   = 40000
RS.NAL   = 20000
```

2.69 .READ/.READC/.READW

Read operations for the three modes of RT–11 I/O are done using the .READ, .READC, and READW programmed requests.

In the case of .READ and .READC, additional queue elements should be allocated for buffered 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 (for example, paper tape). If the read is from a random-access device (disk or DECtape), R0 contains the actual number of words that will be read (.READ or .READC) or have been read (.READW). This number is less than the requested word count if an attempt is made to read past end-of-file, but a partial transfer 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(decimal) 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(decimal) 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

where:

area is the address of a five-word EMT argument block

chan is a channel number in the range 0–376(octal)

buf is the address of the buffer to receive the data read

wcnt is the number of words to be read

blk is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. Note that 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 uparrow (^) prompt (This is true for all .READ requests.)

Notes:

1. .READ and .READC requests instruct the monitor to do a read from the device by queuing a request for the device and immediately returning control to your program.
2. .READ and .READC requests execute as soon as all previous I/O requests to the device handlers have been completed. Note that a read from RK1: must wait for a previous read from RK0: to complete. This is a hardware restriction common to most disks because the controller looks at all I/O operations sequentially.
3. 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:

   a. Nonexistent device/unit
   b. Nonexistent block
   c. In general, errors that do not require data transfers but are controller errors or EOF errors

4. During the .READ and .READC requests, the monitor keeps track of errors in the channel status word. If an error occurs before the monitor can return to the caller, the error is reported on the return from the read request with the carry bit set and the error 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.

5. 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:

10	chan
blk	
buf	
wcnt	
1	

When the user program needs to access the data read on the specified channel, a .WAIT request should be issued as a check that the data has been read completely. If an error occurred during the transfer, the .WAIT request indicates the error.

Errors:

Code	Explanation
0	Attempt to read past end-of-file.
1	Hard error occurred on channel.
2	Channel is not open.

Example:

```
          .TITLE  READ.MAC
;+
; .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

          ERRBYT      = 52                    ;Error Byte location 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$:       .READ       R5,#3                   ;Read a block...
          BCS         G$                      ;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$:     TSTB    @#ERRBYT            ;Read error...EOF?
        BNE     5$                  ;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                   ;
BUFF:   .BLKW   256.                ;I/O buffer
DEFEXT: .WORD   0,0,0,0             ;No default extensions for CSIGEN
DONE:   .ASCIZ  /I-O Transfer Complete/ ;Messages...
MESSG:  .ASCIZ  <15><12>/< Simulating Mainline Processing >/
WERR:   .ASCIZ  /?Write Error?/
RERR:   .ASCIZ  /?Read Error?/
        .EVEN
DSPACE: = .                         ;Handlers may be loaded starting here

        .END    START
```

.READC

The .READC request transfers a specified number of words from the indicated channel to memory. Control returns to the user program immediately after the .READC is initiated. Attempting to read past end-of-file also causes an immediate return, in this case with the carry bit set and the error byte set to 0. Execution of the user program continues until the .READC is complete, then control passes to the routine specified in the request. When an RTS PC is executed in the completion routine, control returns to the user program.

Macro Call: .READC area,chan,buf,wcnt,crtn,blk

where:

area — is the address of a five-word EMT argument block

chan — is a channel number in the range 0–376(octal)

buf — is the address of the buffer to receive the data read

wcnt — is the number of words to be read

crtn — is the address of the user's completion routine. The address of the completion routine must be above 500(octal)

blk — is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. The user program normally updates *blk* before it is used again

When a completion routine is called, error or end-of-file information for a channel is not cleared. The next .WAIT or .READ/.READC on the channel (from either mainline code or a completion routine) produces an immediate return with the C bit set and the error code in byte 52. The completion routine will never be entered if the .READC request returns an error.

Request Format:

R0 → area:

10	chan
blk	
buf	
wcnt	
crtn	

When a .READC completion routine is entered, the following conditions are true:

1. R0 contains the contents of the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer; consequently, the data may not be reliable. The end-of-file bit, bit 13, may be set.
2. 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.
3. On a file-structured transfer, a shortened read is reported when the .READC request is returned, not when the completion routine is called.
4. Registers R0 and R1 can be used by the routine, but all other registers must be saved and restored. Data cannot be passed between the main program and completion routines in any register or on the stack.

Errors:

Code	Explanation
0	Attempt to read past end-of-file; no data was read.
1	Hard error occurred on channel.
2	Channel is not open.

Example:

```
        .TITLE    READC.MAC
;+
; .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,.CSIGEN,.EXIT,.WAIT,.SRESET

        ERRBYT    = 52              ;Error Byte location in SYSCOM
        .ENABL    LSB
START:  .CSIGEN   #DSPACE,#DEFEXT   ;Use CSIGEN to set handlers, files
        CALL      IOXFER            ;Start I/O
        .PRINT    #MESSG            ;Now simulate other mainline process
        MOV       #-1,R5            ;
```

```
1$:     DEC     R5                      ; (Kill some time)
        BNE     1$                      ;
        TSTB    EOF                     ;Did I/O complete?
        BEQ     1$                      ;No...do some more mainline work
        INCB    EOF                     ;Check for read/write error
        BEQ     WERR                    ;EOF = 0 = Write error
        BLT     RERR                    ;EOF < 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     3$                      ;Branch if not...
IOXFER: .READC  #AREA,#3,,,#4$          ;Queue up a read
        BCC     7$                      ;Branch if ok...
        TSTB    @#ERRBYT                ;Error - is it EOF?
        BEQ     6$                      ;Branch if yes
2$:     DECB    EOF                     ;User EOF Flag to indicate hard error
3$:     DECB    EOF                     ;EOF = -2 Read err / = -1 Write err
        RETURN                          ;Leave completion code
4$:     .WAIT   #3                      ;Compl rtne #2 - was read ok?

        BCS     2$                      ;Branch if not
        .WRITC  #AREA,#0,,,#WRDONE      ;Queue up a write...
        BCS     3$                      ;Branch if error
5$:     INC     BLOK                    ;Bump block # for next read
        RETURN                          ;Leave Completion code...
6$:     INCB    EOF                     ;Set EOF flag
7$:     RETURN                          ;then 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 rtne addr
BUFF:   .BLKW   256.                    ;I/O buffer
DEFEXT: .WORD   0,0,0,0                 ;No default extensions for CSIGEN
DONE:   .ASCIZ  /I-D Transfer Complete/ ;Messages...
MESSG:  .ASCIZ  /< Simulating Mainline Processing >/
WRERR:  .ASCIZ  /?Write Error?/
RDERR:  .ASCIZ  /?Read Error?/
EOF:    .BYTE   0                       ;EOF flag
        .EVEN
DSPACE  = .                             ;Handlers may be loaded starting here
        .END    START
```

.READW

The .READW request transfers a specified number of words from the indicated channel to memory. When the .READW is complete or an error is detected, control returns to the user program.

Macro Call: .READW area,chan,buf,wcnt,blk

where:

area is the address of a five-word EMT argument block

chan is a channel number in the range 0–376(octal)

buf is the address of the buffer to receive the data read

wcnt is the number of words to be read; each .READ request can transfer a maximum of 32K words

blk is the block number to be read. For a file-structured .LOOKUP, the block number is relative to the start of the file. For a non-file-structured .LOOKUP, the block number is the absolute block number on the device. The user program normally updates *blk* before it is used again

Request Format:

R0 → area:

10	chan
blk	
buf	
wcnt	
0	

If no error occurred, the data is in memory at the specified address. In an FB environment, the other job can be run while the issuing job is waiting for the I/O to complete.

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  READW.MAC
;+
; .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

          ERRBYT=52                           ;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
                                              ;blk!
          BCC         2$                      ;Branch if no errors
          TSTB        @#ERRBYT                ;Is error EOF?
          BEQ         3$                      ;Yes...
          MOV         #RERR,R0                ;R0 => Read Error Message
1$:       .PRINT                              ;Print the message
          BR          4$                      ;Exit program
```

```
2$:     .WRITW    R5,#0                ;Write the block just read
        INC       IOBLK                ;Bump block # (doesn't affect C bit)
        BCC       READ                 ;Branch if no error
        MOV       #WERR,R0             ;R0 => Write error message
        BR        1$                   ;Branch to output the message
3$:     .CLOSE    #0                   ;End-of-File...Close output channel
        .PRINT    #DONE                ;Announce successful copy
4$:     .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    /? Read error ?/
WERR:   .ASCIZ    /? Write error ?/
DONE:   .ASCIZ    /I-O Transfer Complete/
        .EVEN

DSPACE: =.                             ;Handler(s) can be loaded starting
                                       ;here

        .END      START
```

2.70 .RELEAS

See .FETCH/.RELEAS (Section 2.30).

2.71 .RENAME

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
- cchan — is an unused channel number in the range 0–376(octal)
- 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:

```
        .RENAME   #AREA,#7,#DBLK       ;USE CHANNEL 7
        BCS       RNMERR               ;NOT FOUND
        .
        .
        .
DBLK:   .RAD50    /DT3/
        .RAD50    /OLDFIL/
        .RAD50    /MAC/
        .RAD50    /DT3/
        .RAD50    /NEWFIL/
        .RAD50    /MAC/
```

The first string represents the file to be renamed and the device where it is stored. The second represents the new file name. If a file with the same name as the new file name specified already exists on the indicated device, it is deleted. The second occurrence of the device name DT3 is necessary for proper operation and should not be omitted. The specified channel is left inactive when the .RENAME is complete. .RENAME requires that the handler to be used be resident at the time the .RENAME request is made. If it is not, a monitor error occurs. Note that .RENAME is legal only on files on block-replaceable devices (disks and DECtape). In magtape operations, the handler returns an illegal operation code in byte 52 if a .RENAME request is attempted. 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 request or the PROTECT and UNPROTECT commands.

File dates can be changed using the .SFDAT request.

Errors:

Code	Explanation
0	Channel open.
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  RENAME.MAC

;+
; .RENAME - This is an example in the use of the .RENAME request.
; The example renames a file according to filespecs input thru the
; .CSISPC request.
;-

          .MCALL      .RENAME,.PRINT,.EXIT
          .MCALL      .CSISPC,.FETCH,.SRESET

          ERRBYT      = 52                    ;Error byte location

START:    .CSISPC     #FILESP,#DEFEXT         ;Use .CSISPC to get file specs
          .FETCH      #HANLOD,#FILESP         ;Get Handler from outspec
          BCS         2$                      ;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         1$                      ;Operation successful
          DECB        @#ERRBYT                ;Make error code -1,0 or +1
          BEQ         3$                      ;Branch if File-Not-Found
          MOV         #ILLOP,R0               ;Illegal operation-set up msg
          BR          5$                      ;Branch to report error
```

```
1$:     .SRESET                         ;Dismiss handlers
        .EXIT                           ;Exit program
2$:     MOV     #NOHAN,R0               ;Fetch failed-set up message
        BR      5$                      ;Branch to report error
3$:     MOV     #NOFIL,R0               ;File not found-setup message
5$:     .PRINT                          ;Print error message
        BR      1$                      ;Then exit via .SRESET
AREA:   .BLKW   5                       ;EMT Argument block
DEFEXT: .WORD   0,0,0,0                 ;No default extensions
NOFIL:  .ASCIZ  /?File not found?/      ;Error message text
ILLOP:  .ASCIZ  /?Illegal Operation?/
NOHAN:  .ASCIZ  /?.FETCH Failed?/
        .EVEN
FILESP: .BLKW   39.*2                   ;CSISPC Input Area
HANLOD  = .                             ;Handlers can load here...
        .END    START
```

2.72 .REOPEN

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(octal)

cblk — is the address of the five-word block where the channel status information was stored

Request Format:

R0 → area:

6	chan
cblk	

Errors:

Code	Explanation
0	The specified channel is in use. The .REOPEN has not been done.

Example:

Refer to the example for the .SAVESTATUS request.

2.73 .RSUM (FB and XM Only)

See .SPND/.RSUM (Section 2.84).

2.74 .SAVESTATUS

The .SAVESTATUS request stores five words of channel status information into a user-specified area of memory. These words contain all the information RT–11 requires to completely define a file. When a .SAVESTATUS is done, the data words are placed in memory, the specified channel is freed, and the file is closed. When the saved channel data is required, the .REOPEN request is used.

.SAVESTATUS can only be used if a file has been opened with .LOOKUP. If .ENTER was used, .SAVESTATUS is invalid and returns an error. Note that .SAVESTATUS is not valid for magtape or cassette files.

The .SAVESTATUS/.REOPEN requests are used together to open many files on a limited number of channels or to allow all .LOOKUPs to be done at once to avoid USR swapping.

While the .SAVESTATUS/.REOPEN combination is useful, care must be observed when using it. In particular, the following cases should be avoided:

1. If a .SAVESTATUS is performed and the same file is then deleted before it is reopened, it becomes available as an empty space that could be used by the .ENTER command. If this sequence occurs, the contents of the file supposedly saved changes.
2. Although the device handler for the required peripheral need not be in memory for execution of a .REOPEN, the handler must be in memory when a .READ or .WRITE is executed, or a fatal error is generated.

One of the more common uses of .SAVESTATUS and .REOPEN is to consolidate all directory access motion and code at one place in the program. All files necessary are opened and their status saved, then they are 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. The user should be aware of the consequences of locking in the USR in a foreground/background environment. If the background job locks in the USR when the foreground job requires it, the foreground job is delayed until the background job unlocks the USR.

Macro Call: .SAVESTATUS area,chan,cblk

where:

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

chan is a channel number in the range 0–376(octal)

cblk is the address of the five-word user memory block where the channel status information is to be stored

Request Format:

R0 → area:

5	chan
cblk	

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 of this file, or 0 if non-file-structured
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

Errors:

Code	Explanation
0	The channel specified is not currently associated with any files; that is, a previous .LOOKUP on the channel was never done.
1	The file was opened with an .ENTER request, or a .SAVESTATUS request was performed for a magtape or cassette file.

Example:

```
          .TITLE SAVEST.MAC
;+
; .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 2-6 files into 1 file,
;-reading all input files on one channel.

          .MCALL    .CSIGEN,.SAVESTATUS,.REOPEN,.CLOSE,.EXIT
          .MCALL    .READW,.WRITW,.PRINT,.PURGE

          ERRBYT    = 52                ;Error byte loc in SYSCOM

START:    .CSIGEN   #DSPACE,#DEFEXT     ;Get file specs,open files,load handlers
          MOV       #3,R4               ;R4 = 1st input channel
          MOV       #AREA,R3            ;R3 => EMT Argument block
          MOV       #SAVBLK,R5          ;R5 => Channel savestatus blocks
1$:       .SAVEST   R3,R4,R5            ;Save channel status
          BCS       2$                  ;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       1$                  ;Branch if not
2$:       MOV       #SAVBLK,R5          ;R5 => to 1st saved channel status
          BEQ       7$                  ;Branch if no input files
4$:       .REOPEN   R3,#3,R5            ;Re-open input channel on Ch 3
          CLR       BLK                 ;Start reading with block 0
5$:       .READW    R3,#3,#BUFFR,#256.,BLK ;Read a block
          BCC       6$                  ;Branch if no error
          TSTB      @#ERRBYT            ;Check if error = EOF
          BNE       8$                  ;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       4$                  ;Branch if yes
          .CLOSE    #0                  ;We're done...close output channel
          .PRINT    #DONE               ;Announce merge complete
          .EXIT                         ;Exit program
```

```
6$:     .WRITW   R3,#0,#BUFFR,#256.,WBLK ;Write block just read
        INC      WBLK                    ;Bump to next output block
        INC      BLK                     ;same ffor input blk (doesn't affect C bit)
        BCC      5$                      ;Branch if no error on write
        MOV      #WERR,R0                ;Write error - R0 => message
        BR       9$                      ;merge...
7$:     MOV      #NOINP,R0               ;R0 => No input files message
        BR       9$                      ;merge...
8$:     MOV      #RERR,R0                ;R0 => Read error msg
9$:     .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   /?No input files?/      ;Error messages
WERR:   .ASCIZ   /?Write Error?/
RERR:   .ASCIZ   /?Read Error?/
DONE:   .ASCIZ   /I-O Transfer Completed/
        .EVEN
BUFFR:  .BLKW    256.                    ;I/O buffer
DSPACE  = .                              ;Handlers start here...

        .END     START
```

2.75 .SCCA

The .SCCA request:

1. Inhibits a CTRL/C abort
2. Indicates when a double CTRL/C is initiated at the keyboard
3. Distinguishes between single and double CTRL/C commands

CTRL/C characters are placed in the input ring buffer and are treated as normal control characters without specific system functions. The request requires a terminal status word address that is used to report consecutive CTRL/C input sequences. Bit 15 of the status word is set when consecutive CTRL/C characters are detected. The program must clear the bit.

There are three cautions to observe when using .SCCA. First, the request can cause CTRL/C to appear in the terminal input stream, and therefore the program must provide a way to handle it. Second, the request makes it impossible to terminate program loops from the console, and therefore it should be used only in thoroughly tested, reliable programs. When .SCCA is in effect and the program enters an interminable loop, the system must be halted and re-bootstrapped. The keyboard monitor ABORT command is not inhibited by .SCCA, however, so foreground and system jobs can still be terminated in this manner. Third, a CTRL/C from indirect command files is not intercepted by the .SCCA request.

A .SCCA request with a status word address of zero disables the intercept and re-enables CTRL/C system action.

Macro Call: .SCCA area,addr

where:

area — is the address of a two-word parameter block

addr — is the address of a terminal status word (an address of 0 re-enables the CTRL/C command)

Request Format:

R0 → area:

35	0
addr	

Errors:

None.

Example:

```
          .TITLE  SCCA.MAC

;+
; .SCCA - This is an example in the use of the .SCCA request. The
; example is a simulation of a mainline Foreground program which is
; currently suspended waiting for a message from the Background, but which
; needs to close a file (perhaps opened by a .ENTER ?) before aborting
; from CTRL-C action. A completion routine periodically inspects the CTRL-C
; status word and resumes the mainline if double CTRL-C is entered.
; NOTE: This example MUST be run as a FG job under an FB monitor.
;-

          .MCALL      .SCCA,.RCVDC,.EXIT,.PRINT,.MRKT
          .MCALL      .QSET,.SPND,.RSUM

START:    .QSET       #QELEM,#1               ;Allocate another Q-Element
          .SCCA       #MAREA,#SCCA            ;Inhibit ^C^C action by monitor
1$:       CALL        CWATCH                  ;Start "watchdog" completion rtne
          .RCVDC      #MAREA,#MBUFF,#40.,#MESG ;Look for a message
          ;           .                       ;No errors - there's always BG
          ;           .                       ;Other processing here...
          ;           .                       ;
          .PRINT      #SLEEP                  ;Announce we're going to suspend
          .SPND                               ;Suspend to wait for message
          TST         SCCA                    ;We've been .RSUMed...^C^C hit??
          BNE         CLOSE                   ;Branch if yes
          ;           .                       ;otherwise assume message came in...
          ; <process message here>
          ;           .
          BR          1$                      ;Loop...

CWATCH:   TST         SCCA                    ;Check if ^C^C entered...
          BEQ         MARK                    ;Branch if no
MESG:     .RSUM                               ;Yes...wake up the mainline
          RETURN                              ;then leave completion code
MARK:     .MRKT       #CAREA,#TIME,#CWATCH,#1 ;Schedule to run again in 10 sec.
          RETURN                              ;then leave completion code

CLOSE:    .PRINT      #ABORT                  ;Announce we're aborting
          ;           .                       ;proceed with "orderly" abort
          ; <Output files(s) closed here>
          ;           .
          .EXIT                               ;Exit the program

QELEM:    .BLKW       7                       ;Extra Q-Element
MBUFF:    .BLKW       41.                     ;Message buffer
MAREA:    .BLKW       5                       ;EMT Argument blocks
CAREA:    .BLKW       4                       ;
TIME:     .WORD       0,600.                  ;Time out in 10 seconds
SCCA:     .WORD       0                       ;^C^C Status word

ABORT:    .ASCIZ      /?! Abort Acknowledged...Closing Output File(s) !?/
SLEEP:    .ASCIZ      /! Mainline Suspending !/

          .END        START
```

2.76 .SDAT/.SDATC/.SDATW (FB and XM Only)

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 the FB or XM 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 FB 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 .SDATW of zero words is treated as a .RCVDW of zero words. Thus, avoid a word count of zero at all times when using a .SDATW request.

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.

.SDAT
Macro Call: .SDAT area,buf,wcnt

where:

area is the address of a five-word EMT argument block

buf is the buffer address of the beginning of the message to be transferred

wcnt is the number of words to transfer

Request Format:

R0 → area:

25	0
unused	
buf	
wcnt	
1	

Errors:

Code	Explanation
0	No other job exists. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
;+
; .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.
;-

          .TITLE      SDATF.MAC
;+
; 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      /Hi - FG alive and well and waiting for a message!/
FMSG:     .ASCIZ      /Hey BG - Got your message it reads:/
          .END        MWAITF

          .TITLE      STARTB.MAC
;+
; 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:    .ASCII      /Enter message to be sent to FG Job/<15><12>/>/<200>
NOFG:     .ASCIZ      /?No FG?/
          .END        MWAITB
```

.SDATC

Macro Call: .SDATC area,buf,wcnt,crtn

where:

- area — is the address of a five-word EMT argument block
- buf — is the buffer address of the beginning of the message to be transferred
- wcnt — is the number of words to transfer
- crtn — is the address of the completion routine to be entered when the message has been transmitted

Request Format:

R0 → area:

25	0
unused	
buf	
wcnt	
crtn	

Errors:

Code	Explanation
0	No other job exists. (A job exists as long as it is loaded, whether or not it is active.)

Example:

See the example following .SDATW.

.SDATW
Macro Call: .SDATW area,buf,wcnt

where:

area is the address of a five-word EMT argument block

buf is the buffer address of the beginning of the message to be transferred

wcnt is the number of words to transfer

Request Format:

R0 → area:

25	0
unused	
buf	
wcnt	
0	

Errors:

Code	Explanation
0	No other job exists. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
;+
; .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.
;-
          .TITLE  SDATWF.MAC
;+
; 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 QUFILE/         ;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    /?Enter Error?/     ;Error message text
FEXIT:    .ASCIZ    /FG Job exiting/    ;Exit message
          .END      STARTF

          .TITLE  SDATWB.MAC
;+
; 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       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
          .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
          .EXIT                         ;then exit.
AREA:     .BLKW     5                   ;EMT argument blk
MSG:      .BLKW     3                   ;Message buffer
BUFF:     .BLKW     256.                ;File buffer
BEXIT:    .ASCIZ    /Channel-Record copy successful/
NOJOB:    .ASCIZ    /?No FG Job?/       ;Error messages...
NOCH:     .ASCIZ    /?FG channel not open?/
RDERR:    .ASCIZ    /?Read Error?/
          .END      STARTB
```

2.77 .SDTTM

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	

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 illegal 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, 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(decimal) 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 must be zero.

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 legal number of ticks for the system line frequency. For a 60 Hz clock, the high-order time may not be larger than 117(octal), and if it is equal to 117, the low-order time may not be equal to or larger than 15000(octal). For a 50 Hz clock, the high-order time may not be larger than 101(octal), and if it is equal to 101, the low-order time may not be equal to or larger than 165400(octal).

Changing the date and/or time has no effect on any outstanding mark time or timed wait requests.

Errors:

None.

Example:

```
          .TITLE  SDTTM.MAC

;+
; .SDTTM.MAC - 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,.PRINT,.EXIT,.GTIM
          .GLOBL      STD,DALITE

STD:      COM         HR                  ;Switch to STD time...
          NEG         HR+2                ;Make one hr in clock ticks
DALITE:   .GTIM       #AREA,#TIME         ;Get the current time
          CALL        JADD                ;Adjust +/- 1 hour
          .SDTTM      #AREA,#NEWDT        ;Set the new system time
          .GTIM       #AREA,#TIME         ;Force date rollover (if any)
          RETURN                          ;Return to caller

NEWDT:    .WORD       -1                  ;.SDTTM arguments - No new date
TIME:     .WORD       0,0                 ;New time
HR:       .WORD       3                   ;One hour in clock ticks (60 cycle
                                          ; clock!)
          .WORD       45700
AREA:     .WORD       0,0                 ;EMT Argument Block

JADD:                                     ;Double precision integer add
          MOV         #HR,R4              ;R4 => Low order of System time + 2
          MOV         #AREA,R3            ;R3 => Low order of One hour + 2
          MOV         #HR,R1              ;R1 => Low order of new time
          MOV         -(R4),R2            ;Put low order of 1st operand in R2
```

```
ADD         -(R3),R2            ;Add in low order of operand #2
MOV         -(R4),R5            ;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         R2,-(R1)            ;Store result where wanted
MOV         R5,-(R1)
RETURN                          ;Return to caller
.END
```

2.78 .SERR

See .HERR/.SERR (Section 2.38).

2.79 .SETTOP

The .SETTOP request specifies a new address as a program's upper limit. The monitor determines whether this address is legal and whether or not a memory swap is necessary when the USR is required. For instance, if the program specified an upper limit below the start address of USR (normally specified in offset 266 in the resident monitor), no swapping is necessary, as 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:

1. A .SETTOP is done to the high limit of the code in a program before buffers or work areas are allocated. If the program is aborted, minimal writing of the user program to the swap blocks (SWAP.SYS) occurs. However, the program is allowed to be restarted successfully.

2. A user command line is now read through .CSISPC or .GTLIN. An appropriate USR swap address is set in location 46. Successive .DSTATUS, .SETTOP, and .FETCH requests are performed to load necessary device handlers. This attempts to keep the USR resident as long as possible during the procedure.

3. 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.

4. If the process has a well-defined closing phase, another .SETTOP can be issued to cause the USR to become resident again to close files (the user should remember to set location 46 to zero if this is done, so that the USR again swaps in the normal area). On return from .SETTOP, both R0 and the word in location 50(octal) contain

the highest memory address allocated for use. If the job requested an address higher than the highest address legal for the requesting job, the address returned is the highest legal address for the job rather than the requested address.

When doing a final exit from a program, the monitor writes the program to the file SWAP.SYS and then reads in the KMON. A .SETTOP #0 at exit time prevents the monitor from swapping out the program 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.

Macro Call: .SETTOP addr

where:

addr is the address of the highest word of the area desired (the last word the program will modify, not the first word it leaves untouched)

Notes:

1. A program should never do a .SETTOP and assume that its new upper limit is the address it requested. It must always examine the returned contents of R0 or location 50 to determine its actual high address.
2. It is imperative that the value returned in R0 or location 50 be used as the absolute upper limit. If this value is exceeded, vital parts of the monitor can be destroyed.

Errors:

None.

Example:

```
          .TITLE  SETTOP.MAC
;+
; .SETTOP - This is an example in the use of the .SETTOP request. The
; 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

          SYSPTR=54                         ;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
          .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
        .EXIT                         ;and exit program

AREA:   .BLKW     3                   ;EMT Argument Block
FILE1:  .RAD50    /DK/                ;A File we're sure to find
        .RAD50    /PIP   /
        .RAD50    /SAV/
FILE2:  .RAD50    /DK/                ;Another file we might find
        .RAD50    /TECO  /
        .RAD50    /SAV/
LMSG:   .ASCIZ    /?Error on .LOOKUP?/ ;Error message
F1FND:  .ASCIZ    /...Found PIP.SAV/
F2FND:  .ASCIZ    /...Found TECO.SAV/
        .EVEN
        .END      START
```

2.79.1 .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 Chapter 11 in the *RT–11 System Utilities Manual* or Chapter 4 of the *RT–11 System User's Guide*). The *RT–11 Software Support 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

1. 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.

2. A .SETTOP that requests a job's upper limit above the program's virtual high limit will return the highest available address as follows:

   a. 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.

   b. 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

1. As in privileged jobs, a .SETTOP request can never get less than the virtual high limit of the job.

2. If a .SETTOP requests an upper limit greater than the virtual high limit, the following occurs:

   a. 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 LINK chapter of the *RT–11 System Utilities Manual* and the *RT–11 Software Support 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.

   b. 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.

2.80 .SFDAT

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.

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 0–376

dblk is the address of a four-word block containing a filespec in Radix–50

date is the address of the new date, in RT–11 format If this argument is #0, the system date is used; bits 14 and 15 are always set to 0, but no other check is made for an illegal date

Request Format:

R0 → area:

42	chan
dblk	
date	

Errors:

Code	Explanation
0	Channel in use
1	File not found
2	Invalid operation (device not file structured)

Example:

Refer to the example for the .FPROT request.

2.81 .SFPA (Special Feature)

The .SFPA request allows users with floating-point hardware to set trap addresses to be entered when a floating-point exception occurs. If no user trap address is specified and a floating-point (FP) exception occurs, a *?MON–F–FPU trap* occurs, and the job is aborted.

Macro Call: .SFPA area,addr

where:

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

addr is the address of the routine to be entered when an exception occurs

Request Format:

R0 → area:

30	0
addr	

Notes:

1. The user trap routine must save and restore any registers it uses. It exits with an RTI instruction.
2. 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.
3. In the FB environment, an address value of #1 indicates that the FP registers should be switched when a context switch occurs, but no user traps are enabled. This allows both jobs to use the FP unit. An address of #1 to the SJ monitor is equivalent to an address of #0.
4. 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.

5. If the FP11 is being used, the instruction STST –(SP) is executed by the monitor before entering the user's trap routine. Thus, the trap routine must pop the two status words off the stack before doing an RTI. The program can tell if FP hardware is available by examining the configuration word in the monitor.

Errors:

None.

Example:

```
          .TITLE  SFPA.MAC

;+
; .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

          SYSPTR      = 54                ;Loc of beginning of Monitor
          CONFIG      = 300               ;Offset to Monitor configuration wd
          FP11        = 100               ;FPU present bit
START:    ;           .                   ;Mainline program...
          ;           .
          ;           .
          .SFPA       #AREA,#FPTRAP       ;Set up FPU error trap
          ;           .
          ;           .                   ;continue mainline program
          ;           .
          .EXIT                           ;Exit program

FPTRAP:                                   ;FPU exception routine
          ;           .                   ;Handle exception...
          ;           .
          ;           .
CKFPU:    MOV         @#SYSPTR,R0         ;R0 => base of RMON
          BIT         #FP11,CONFIG(R0)    ;Check for FPU hdwe
          BEQ         1$                  ;Branch if none
          CMP         (SP)+,(SP)+         ;Must pop status regs off stack!
1$:       RTI                             ;Before returning from interrupt
          .END        START
```

2.82 .SPCPS (FB and XM SYSGEN Option)

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.

The .SPCPS request is a system generation feature and is available only in FB and XM. If a program issues this call under SJ or under a monitor not generated for the call, no action is taken and no error is returned.

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   SPCPS.MAC
          .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

          ERRBYT   = 52                        ;Error Byte location in SYSCOM

          .ENABL   LSB
START:    .CSIGEN  #DSPACE,#DEFEXT             ;Use CSIGEN to get handlers, files
          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     @#ERRBYT                    ;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)
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  /I-O Transfer Complete/ ;Messages...
MESSG:  .ASCIZ  /< Simulating Mainline Processing >/
WRERR:  .ASCIZ  /?Write Error?/
RDERR:  .ASCIZ  /?.Read Error?/
SPERR:  .ASCIZ  /?.SPCPS Error?/
        .EVEN
DSPACE  = .                             ;Handlers may be loaded starting here
        .END    START
```

2.83 .SPFUN

This request is used with certain device handlers to do device dependent functions, such as rewind and backspace. It can be used with diskettes and some disks to allow reading and writing of absolute sectors. This request can determine the size of a volume mounted in a particular device unit for RX02 diskettes, RK06/RK07 disks, RL01/RL02 disks, MSCP disks, and logical disks.

Macro Call: .SPFUN area,chan,func,buf,wcnt,blk[,crtn]

where:

area — is the address of a six-word EMT argument block

chan — is a channel number in the range 0 to 376(octal)

func — is the numerical code of the function to be performed; these codes must be negative (bytewise) i.e. bit 7 set

buf — is the buffer address; this parameter must be set to zero if no buffer is required

wcnt — is defined in terms of the device handler associated with the specified channel and in terms of the specified special function code

blk is also defined in terms of the device handler associated with the specified channel and in terms of the specified special function code

crtn is the entry point of a completion routine. If left blank, 0 is automatically inserted. This value is the same as for .READ, .READC, and .READW.

0 = wait I/O (.READW)
1 = real time (.READ)

Value >500 = completion routine

Request Format:

R0 → area:

32	chan
blk	
buf	
wcnt	
func	377
crtn	

The *chan, blk,* and *wcnt* arguments are the same as those defined for .READ/.WRITE requests. They are only required when doing a .WRITE with extended record gap to magnetic tape. If the *crtn* argument is left blank, the requested operation completes before control returns to the user program. 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.

The available functions and function codes for magtape and cassette are as follows:

Function	MM, MS, MT	CT
Forward to last file		377
Forward to last block		376
Forward to next file		375
Forward to next block		374
Rewind to load point	373	373
Write file gap		372
Write EOF	377	
Forward one block	376	
Backspace one block	375	
Write with extended file gap	374	
Off-line rewind	372	
Write	371	
Read	370	
Stream at 100 ips (MS only)	367	

The available functions and function codes for diskettes, RK06/RK07 disks, RL01 and RL02 disks, the logical disk handler, and MSCP disks are as follows:

Function	DX	DM	DY	DL	LD	DU
Read	377	377	377	377		
Write	376	376	376	376		
Write with deleted data mark	375		375			
Force a read by the handler of the bad block replacement table from block 1 of the disk		374		374		
Return device size	373	373	373	373	373	373
Read/write translation table					372	372
Direct MSCP access						371

To use the .SPFUN request, the handler must be in memory and a channel must be associated with a file via a .LOOKUP request.

A .SPFUN request to write absolute blocks on a diskette should not write anything in track 0 if you want to use DUP or the COPY/DEVICE command to back up the volume. DUP does not copy data in track 0. Also, you should be careful to specify a valid buffer address and word count. The monitor checks that the *buf* argument is in the job area, but it does not check *buf* + *wcnt*. If you use the .SPFUN request, and the device handler for that device does not support special functions or the particular .SPFUN code used, the call simply returns to the program without reporting an error.

For the RK06/07 handler (DM), special function codes 377 and 376 require the buffer size to be one word larger than necessary for the data. The first word of the buffer contains the error information returned as a result of the .SPFUN request. The data transferred as a result of the read or write request is found in the second and following words of the buffer. The error codes and information are as follows:

Code	Meaning
100000	The I/O operation is successful.
100200	A bad block was detected (BSE error).
100001	An ECC error is corrected.
100002	An error recovered on retry.
100004	An error recovered through an offset retry.
100010	An error recovered after recalibration.
1774xx	An error did not recover.

Other device-specific information is included in the *RT–11 Software Support Manual.*

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.

Additional qualifying information for these errors is returned in the first two words of the *blk* argument status block. This information is given in Chapter 10 of the *RT–11 Software Support Manual.*

Example:

```
          .TITLE  SPFUN.MAC

;+
; .SPFUN - This is an example in the use of the .SPFUN request. The
; example rewinds a cassette and writes out a 256-word buffer and
; then a file gap.
;-

          .MCALL      .FETCH,.LOOKUP,.SPFUN,.WRITW
          .MCALL      .EXIT,.PRINT,.WAIT,.CLOSE

START:    .FETCH      #HSPC,#CT               ;Fetch the CT Handler
          BCS         1$                      ;Branch if failed
          .LOOKUP     #AREA,#4,#CT            ;Open channel 4 for output
          BCS         2$                      ;Branch if error (should never hap-
                                              ;pen!)
          .SPFUN      #AREA,#4,#373,#0        ;Rewind to BOT using Synchronous I/O
          BCS         3$                      ;Branch on error
          .WRITW      #AREA,#4,#BUFF,#256.,BLK ;Write one block
          BCS         4$                      ;Branch on error
          .SPFUN      #AREA,#4,#372,#0,,#1    ;Write a file gap with Asynch I/O
          .PRINT      #DONE                   ;Announce that we're done
          .WAIT       #4                      ;Wait for file gap operation to finish
          .CLOSE      #4                      ;Close the file
          .EXIT                               ;then exit the program

1$:       MOV         #FERR,R0                ;Process errors here...
          BR          5$
2$:       MOV         #LKERR,R0
          BR          5$
3$:       MOV         #SPERR,R0
          BR          5$
4$:       MOV         #WERR,R0
5$:       .PRINT                              ;Print error message
          .EXIT                               ;then exit program

AREA:     .WORD       0                       ;EMT Argument block
BLK:      .WORD       0,0,0,0,0
CT:       .RAD50      /CT /                   ;Cassette Device Descriptor
          .WORD       0,0,0                   ;Null filespec
BUFF:     .BLKW       256.                    ;Output buffer
DONE:     .ASCIZ      /All done !/            ;Message text...
FERR:     .ASCIZ      /?.FETCH Error?/
LKERR:    .ASCIZ      /?.LOOKUP Error?/
SPERR:    .ASCIZ      /?Special Function Error?/
WERR:     .ASCIZ      /?Write Error?/
          .EVEN
          HSPC = .                            ;Handler can load in here...
          .END        START
```

2.84 .SPND/.RSUM (FB and XM Only)

The .SPND/.RSUM requests control execution of a job's mainline code (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 one of the completion routines resumes the mainline code. These functions enable a program to wait for a particular I/O or mark time request by suspending the mainline and having the selected event's completion routine issue a .RSUM. This differs from the .WAIT request, which suspends the mainline 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:

1. 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.
2. A program must issue an equal number of .SPND and .RSUM requests.
3. A .RSUM request from the mainline code increments the suspension counter.
4. 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.
5. Since a .TWAIT is simulated in the monitor using suspend and resume, a .RSUM issued from a completion routine without a matching .SPND can cause the mainline to continue past a timed wait before the entire time interval has elapsed.
6. A .SPND or .RSUM, like most other programmed requests, can be issued from within a user-written interrupt service routine if the .INTEN/.SYNCH sequence is followed. All notes referring to .SPND/.RSUM from a completion routine also apply to this case.

Errors:

None.

Example:

```
        .TITLE      SPND.MAC

;+
; .SPND/.RSUM- This is an example in the use of the .SPND/.RSUM requests.
; The example is a simulation of a mainline Foreground program which is
```

```
; currently suspended waiting for a message from the Background, but which
; needs to close a file (perhaps opened by a .ENTER ?) before aborting
; from CTRL-C action. A completion routine periodically inspects the CTRL-C
; status word and resumes the mainline if double CTRL-C is entered.
; NOTE: This example MUST be run as a FG job under an FB monitor.
;-

          .MCALL    .SCCA,.RCVDC,.EXIT,.PRINT,.MRKT
          .MCALL    .QSET,.SPND,.RSUM

START:    .QSET     #QELEM,#1             ;Allocate another Q-Element
          .SCCA     #MAREA,#SCCA          ;Inhibit ^C^C action by monitor
1$:       CALL      CWATCH                ;Start "watchdog" completion rtne
          .RCVDC    #MAREA,#MBUFF,#40.,#MESG ;Look for a message
          ;         .                     ;No errors - there's always BG
          ;         .                     ;Other processing here...
          ;         .                     ;
          .PRINT    #SLEEP                ;Announce we're going to suspend
          .SPND                           ;Suspend to wait for message
          TST       SCCA                  ;We've been .RSUMed...^C^C hit??
          BNE       CLOSE                 ;Branch if yes
          ;         .                     ;otherwise assume message came in...
          ; <Process message here>
          ;         .
          BR        1$                    ;Loop...

CWATCH:   TST       SCCA                  ;Check if ^C^C entered...
          BEQ       MARK                  ;Branch if no
MESG:     .RSUM                           ;Yes...wake up the mainline
          RETURN                          ;then leave completion code
MARK:     .MRKT     #CAREA,#TIME,#CWATCH,#1 ;Schedule to run again in 10 sec.
          RETURN                          ;then leave completion code

CLOSE:    .PRINT    #ABORT                ;Announce we're aborting
          ; <Output file(s) closed here>  ;Proceed with "orderly" abort
          ;         .
          .EXIT                           ;Exit the program

QELEM:    .BLKW     7                     ;Extra Q-Element
MBUFF:    .BLKW     41.                   ;Message buffer
MAREA:    .BLKW     5                     ;EMT Argument blocks
CAREA:    .BLKW     4                     ;
TIME:     .WORD     0,600.                ;Time out in 10 seconds
SCCA:     .WORD     0                     ;^C^C Status word

ABORT:    .ASCIZ    /?! Abort Acknowledged...Closing Output File(s) !?/
SLEEP:    .ASCIZ    /! Mainline Suspending !/

          .END      START
```

2.85 .SRESET

The .SRESET (software reset) request:

1. Cancels any messages sent by the job.
2. Waits for all job I/O to complete, which includes waiting for all completion routines to run.
3. Dismisses any device handlers that were brought into memory via .FETCH calls. Handlers loaded via the keyboard monitor LOAD command remain resident, as does the system device handler.
4. Purges any currently open files. Files opened for output with .ENTER are never made permanent.

5. Reverts to using only 16(decimal) I/O channels. Any channels defined with .CDFN are discarded. A .CDFN must be reissued to open more than 16 channels after a .SRESET is performed.
6. Clears the job's .SPND/.RSUM counter.
7. Resets the I/O queue to one element. A .QSET request must be reissued to allocate extra queue elements.
8. Cancels all outstanding .MRKT requests.

Macro Call: .SRESET

Errors:

None.

Example:

```
          .TITLE     SRESET.MAC
;+
; .SRESET - This is an example in the use of the .SRESET request.
; The example renames a file according to filespecs entered using the
; .CSISPC request.
;-
          .MCALL     .RENAME,.PRINT,.EXIT
          .MCALL     .CSISPC,.FETCH,.SRESET
          ERRBYT     = 52                    ;Error byte location
START:    .CSISPC    #FILESP,#DEFEXT         ;Use .CSISPC to get file specs
          .FETCH     #HANLOD,#FILESP         ;Get Handler from outspec
          BCS        2$                      ;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        1$                      ;Operation successful
          DECB       @#ERRBYT                ;Make error code -1,0 or +1
          BEQ        3$                      ;Branch if File-Not-Found
          MOV        #ILLOP,R0               ;Illegal operation-set up msg
          BR         5$                      ;Branch to report error
1$:       .SRESET                            ;Dismiss handlers
          .EXIT                              ;Exit program
2$:       MOV        #NOHAN,R0               ;Fetch failed-set up message
          BR         5$                      ;Branch to report error
3$:       MOV        #NOFIL,R0               ;File not found-setup message
5$:       .PRINT                             ;Print error message
          BR         1$                      ;Then exit via .SRESET
AREA:     .BLKW      5                       ;EMT Argument block
DEFEXT:   .WORD      0,0,0,0                 ;No default extensions
NOFIL:    .ASCIZ     /?File not found?/      ;Error message text
ILLOP:    .ASCIZ     /?Illegal Operation?/
NOHAN:    .ASCIZ     /?.FETCH Failed?/
          .EVEN
FILESP:   .BLKW      39.                     ;CSISPC Input Area
HANLOD    = .                                ;Handlers can load here...
          .END       START
```

2.86 .SYNCH (Device Handler and Interrupt Service Routine Only)

This macro call enables your program to issue programmed requests from within 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 1	RT–11 maintains this word; its contents should not be altered by the user
2	The current job's number. This must be set up by the user program. It can be obtained by a .GTJB call or from the I/O queue element in a device handler
3	Unused
4	Unused
5	R0 argument. When a successful return is made from .SYNCH, R0 contains this argument
6	Must be –1
7	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. 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.

NOTE

The monitor dismisses the interrupt without returning to the .SYNCH routine if one of the following conditions occur:

1. You specified an illegal job number.
2. The job number does not exist (for example, you specify 2, and there is no foreground job).
3. The job is exited or terminated with an .EXIT programmed request.

You can find out if the block is in use by:

1. Checking location Q.COMP (offset 14 octal). If this location contains a zero, the block is available.
2. 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 word 5 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. Exit from the routine should be done via an RTS PC.

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  /?Write Error?/
SYNERR: .ASCIZ  /?SYNCH Failed?/
        .EVEN

        END     START
```

2.87 .TIMIO (Device Handler Only)

The .TIMIO macro issues the device time-out call in the handler I/O initiation section. 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 XM systems, the completion routine executes with kernel mapping, since it is still a part of the interrupt service routine. (See the *RT–11 Software Support Manual* for more information about interrupt service routines and the XM monitor.) As usual with completion routines, R0 and R1 are available for use. When the completion routine is entered, R0 contains the sequence number of the request that timed out.

Macro Call: .TIMIO tbk,hi,lo

where:

- tbk — is the address of the timer block, a seven-word pseudo timer queue element. (The timer block format is shown in Table 2–1 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 TIM$IT, TIM$IT=0

.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,-1,0   ;.TIMIO argument block

          .DREND   SP                           ;Use .DREND to end code

          .END
```

2.88 .TLOCK

The .TLOCK (test lock) request is used in an FB environment to attempt to gain ownership of the USR. It is similar to .LOCK in that, if successful, the user job returns with the USR in memory (it is identical to .LOCK in the SJ monitor). However, if a job attempts to .LOCK the USR while 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  TLOCK.MAC

;+
; .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         1$              ;Yup - we did
          CALL        JOB2            ;Nope - do it now
1$:       .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

AREA:     .BLKW       5               ;EMT argument block
FILE:     .RAD50      /DK/            ;File spec for Job #1
          .RAD50      /QUFILE/        ;
          .RAD50      /TMP/           ;
LKERR:    .PRINT      #LKMSG          ;Error on .LOOKUP - Report it!
          .EXIT
LKMSG:    .ASCIZ      /?File Not Found?/
J1MSG:    .ASCIZ      /Job #1 Executed/
J2MSG:    .ASCIZ      /Job #2 Executed/
J2SW:     .BYTE       0               ;Switch to control Job #2 execution
          .EVEN

JOB2:     .PRINT      #J2MSG          ;2nd "Job" - Doesn't need USR
          RTS         PC              ;Return when done

          .END        START
```

2.89 .TRPSET

.TRPSET allows the user job to intercept traps to 4 and 10 instead of having the job aborted with a *?MON–F–Trap to 4* or *?MON–F–Trap to 10* message.

If .TRPSET is in effect when an error trap occurs, the user-specified routine is entered. The 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–Stack overflow* in the SJ monitor or *?MON–F–Trap to 4* in the FB and XM monitors (see the *RT–11 System Message Manual*).

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

Request Format:

R0 → area:

3	0
addr	

Notes:

1. 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.

2. In the XM monitor, traps dispatched to a user program by .TRPSET execute in user mode. They appear as interrupts of the user program by a synchronous trap operation. Programs that intercept error traps by trying to steal the trap vectors must be carefully designed to handle two cases accurately: programs that are virtual jobs and programs that are privileged jobs.

   If the program is a virtual job, the stolen vector is in user virtual space that is not mapped to kernel vector space. The proper method is to use .TRPSET; otherwise interception attempts fail and the monitor continues to handle traps to 4 and 10.

   If the program is a privileged job, it is mapped to the kernel vector page. The user can steal the error trap vectors from the monitor, but the benefits of doing so must be carefully evaluated in each case. Trap routines run in the mapping mode specified by bits 14 and 15 of the trap vector PS word. With both bits set to 0, kernel mode is set. However, kernel mapping is not always equivalent to user mapping, particularly when extended memory is being used. With both 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
          TST         @#166666              ;Address non-existent memory - Trap to 4
          .EXIT                             ;Exit program

TRPLOC:                                     ;Trap routine
          BCS         1$                    ;C bit set = TRAP 10
          .PRINT      #TRP4                 ;Report Trap to 4
          BR          2$                    ;Branch to reset trap routine
1$:       .PRINT      #TRP10                ;Report trap to 10
          .TRPSET     #AREA,#TRPLOC         ;Reset trap routine address
2$:       RTI                               ;Return to offending code

AREA:     .WORD       0,0                   ;EMT argument block
TRP4:     .ASCIZ      /?Trap to 4?/         ;Error messages...
TRP10:    .ASCIZ      /?Trap to 10?/

          .END        START
```

2.90 .TTYIN/.TTINR

The requests .TTYIN and .TTINR transfer a character from the console terminal to the user program. The character thus obtained appears right-justified (even byte) in R0. The user can cause the characters to be returned in R0 only, or in R0 and other locations.

The expansion of .TTYIN is:

```
EMT 340
BCS .–2
```

The expansion of .TTINR is:

```
EMT 340
```

If no characters or lines are available when an EMT 340 is executed, return is made with the carry bit set. The implication of these calls is that .TTYIN causes a tight loop waiting for a character/line to appear, while the user can either wait or continue processing using .TTINR.

If the carry bit is set when execution of the .TTINR request is completed, it indicates that no character was available; the user has not yet typed a valid line. Under the FB or XM monitor, .TTINR does not return the carry bit set unless bit 6 of the Job Status Word (JSW) was on when the request was issued.

There are two modes of doing console terminal input. The choice is governed by bit 12 of the job status word. If bit 12 is 0, normal I/O is performed. In this mode, the following conditions apply:

1. The monitor echoes all characters typed.
2. CTRL/U and the DELETE key perform line deletion and character deletion, respectively.
3. A carriage return, line feed, CTRL/Z, or CTRL/C must be struck before characters on the current line are available to the program. When one of these is typed, characters on the line typed are passed one by one to the user program.

If bit 12 is 1, the console is in special mode. The effects are:

1. The monitor does not echo characters typed except for CTRL/C and CTRL/O.
2. CTRL/U and the DELETE key do not perform special functions.
3. Characters are immediately available to the program.

In special mode, the user program must echo the characters received. However, CTRL/C and CTRL/O are acted on by the monitor in the usual way. Bit 12 in the JSW must be set by the user program. This bit is cleared when the program terminates.

Regardless of the setting of bit 12, when a carriage return is entered, both carriage return and line feed characters are passed to the program; if bit 12 is 0, these characters will be echoed.

Lowercase conversion is determined by the setting of bit 14 in the JSW. If bit 14 is 0, lowercase characters are converted to uppercase before being echoed (if bit 12 is 0) and passed to a program; if bit 14 is 1, lowercase characters are echoed (if bit 12 is 0) and passed as received. Bit 14 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 bit 12. The monitor always acts on these characters (unless the SET TT NOFB command is issued).

CTRL/S and CTRL/Q are intercepted by the monitor (unless, under the FB or XM monitor, the SET TT NOPAGE command is issued).

Under the FB or XM monitor, if a terminal input request is made and no character is available, job execution is blocked until a character is ready. This is true for both .TTYIN and .TTINR, and for both normal and special modes. If a program requires execution to continue and the carry bit to be returned, it must set bit 6 of the Job Status Word before the .TTINR request. Bit 6 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 bits 4 and 12 of the JSW are 0, input from a .TTYIN

or .TTINR request will be edited by SL. If either bit 4 or bit 12 is set, SL will not edit input. If SL is editing input, the state of bit 6 (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.

Macro Calls: .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

Errors:

Code	Explanation
0	No characters available in ring buffer.

Example:

Refer to the example following the description of .TTYOUT/.TTOUTR.

2.91 .TTYOUT/.TTOUTR

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.

If the carry bit is set when execution of the .TTOUTR request is completed, it indicates that there is no room in the buffer and that no character was output. Under the FB or XM monitor, .TTOUTR normally does not return the carry bit set. Instead, the job is blocked until room is available in the output buffer. If a job requires execution to continue and the carry bit to be returned, it must turn on bit 6 of the Job Status Word 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.

NOTE

If a foreground job leaves bit 6 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.

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

Errors:

Code	Explanation
0	Output ring buffer full.

Example:

```
          .TITLE  TTYIN.MAC

;+
; .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  TTINR.MAC

;+
; .TTINR / .TTOUTR - This is an example in the use of the .TTINR &
; .TTOUTR requests. Like TTYIN.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
          .MCALL      .TTINR,.TTOUTR,.EXIT

          JSW = 44                          ;Location of Job Status Word in SYSCOM

START:    MOV         #BUFFER,R1            ;Point R1 to buffer
          CLR         R2                    ;Clear character count
          BIS         #100,@#JSW            ;Set bit #6 in JSW so .TTINR/.TTOUTR will
                                            ;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         #100,@#JSW            ;Clear bit #6 in JSW
          .TTYOUT     (R1)                  ;Use .TTYOUT to wait for room
          BIS         #100,@#JSW            ;Finally successful - reset bit #6
          BR          CHROUT                ;then return to output loop

BUFFER:   .BLKW       64.                   ;Buffer
          .END        START
```

2.92 .TWAIT (SYSGEN Option for SJ)

The .TWAIT request suspends the user's job for an indicated length of time. .TWAIT requires a queue element and thus should be considered when the .QSET request is 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:

1. 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.
2. 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.

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 35 sec).
;-

          .MCALL      .TWAIT,.QSET,.EXIT,.PRINT

START:    CALL        TASK                ;Perform task...
1$:       .TWAIT      #AREA,#TIME         ;Go to sleep for 5 seconds
          BCS         NOQ                 ;Branch if no queue element
          CALL        TASK                ;Perform task again
          DEC         COUNT               ;Bump counter - example good for 35 sec
          BNE         1$                  ;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         1$                  ;Branch if not
          .PRINT      #TICK               ;Odd counter prints "tick..."
          RETURN                          ;Return to caller
1$:       .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.,*5.           ;60 ticks/sec * 5 seconds
COUNT:    .WORD       7                   ;Maximum cycles for example
TCNT:     .WORD       0                   ;Tick,tock count

TICK:     .ASCII      /Tick.../<200>      ;Message text
TOCK:     .ASCIZ      /Tock/
BYE:      .ASCIZ      /Example Concluded/
QERR:     .ASCIZ      /?No Q-Element Available?/
          .END        START
```

2.93 .UNLOCK

See .LOCK/.UNLOCK (Section 2.41).

2.94 .UNMAP (XM Only)

The .UNMAP request unmaps a window and flags that portion of the program's virtual address space as being inaccessible. When an unmap operation is performed for a virtual job, attempts to access the unmapped address space cause a memory management fault. For a privileged job, the default (kernel) mapping is restored when a window is unmapped.

Macro Call: .UNMAP area,addr

where:

area is the address of a two-word argument block

addr is the address of the window control block that describes the window to be unmapped

Request Format:

R0 → area:

36	5
addr	

Errors:

Code	Explanation
3	An illegal window identifier was specified.
5	The specified window was not already mapped.

Example:

Refer to the example following the description of .CRAW.

2.95 .UNPROTECT

See .PROTECT/.UNPROTECT (Section 2.60).

2.96 .WAIT

The .WAIT request suspends program execution until all input/output requests on the specified channel are completed. The .WAIT request, combined with the .READ/.WRITE requests, makes double buffering a simple process.

.WAIT also conveys information through its error returns. An error is returned if either 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. Under these conditions, the subsequent operation is not initiated.

In an FB 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:

```
          .TITLE  WAIT.MAC
;+
; .WAIT - This is an example in the use of the .WAIT request. 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 out-
; puts 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 a 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

          ERRBYT      = 52                    ;Error Byte location 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
1$:       .READ       R5,#3                   ;Read a block...
          BCS         6$                      ;Branch on error
          ;           .
          BIT         #1,IOBLK                ;Then simulate
          BNE         2$                      ;some other
          .PRINT      #MESSG                  ;meaningful(?) process...
          ;           .
2$:       .WAIT       #3                      ;Did read finish OK?
          BCS         5$                      ;Branch if not
          .WRITE      R5,#0                   ;Now write the block just read
          BCS         3$                      ;Branch on error
          ;           .                       ;Could do some more processing here...
          ;           .
          INC         IOBLK                   ;Bump block # for next read
          .WAIT       #0                      ;Wait for write to finish
          BCC         1$                      ;Branch if successful
3$:       MOV         #WRERR,R0               ;R0 => Write error msg
4$:       .PRINT                              ;Report error
          .EXIT                               ;then exit program
5$:       MOV         #RDERR,R0               ;R0 => Read error msg
          BR          4$                      ;Branch to report error
6$:       TSTB        @#ERRBYT                ;Read error...EOF?
          BNE         5$                      ;Branch if not
          .PRINT      #DONE                   ;Yes...announce completion
          .CLOSE      #0                      ;Make output file permanent
          .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                        ;
BUFF:     .BLKW     256.                     ;I/O buffer
DEFEXT:   .WORD     0,0,0,0                  ;No default extensions for CSIGEN
DONE:     .ASCIZ    /I-O Transfer Complete/  ;Messages...
MESSG:    .ASCIZ    <15><12>/< Simulating Mainline Processing >/
WRERR:    .ASCIZ    /?Write Error?/
RDERR:    .ASCIZ    /?Read Error?/
EOF:      .BYTE     0                        ;EOF flag
          .EVEN
DSPACE    = .                                ;Handlers may be loaded starting here

          .END      START
```

2.97 .WDBBK (XM Only)

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 Chapter 4 of the *RT–11 Software Support Manual* for an example that uses the .WDBBK macro and a detailed description of the extended memory feature.

2.98 .WDBDF (XM Only)

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 of the window definition block by setting up the following symbol:

W.NLGH = 16

The .WDBDF macro does not reserve any space for the window definition block (see .WDBBK).

Macro Call: .WDBDF

The .WDBDF macro expands as follows:

```
W.NID   = 0
W.NAPR  = 1
W.NBAS  = 2
W.NSIZ  = 4
W.NRID  = 6
W.NOFF  = 10
W.NLEN  = 12
W.NSTS  = 14
W.NLGH  = 16
WS.CRW  = 100000
WS.UNM  =  40000
WS.ELW  =  20000
WS.MAP  =    400
```

2.99 .WRITE/.WRITC/.WRITW

Write operations for the three modes of RT–11 I/O are done using 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 an FB monitor with the 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

where:

area is the address of a five-word EMT argument block

chan is a channel number in the range 0 to 376(octal)

buf is the address of the memory buffer to be used for output

wcnt is the number of words to be written

blk is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. The user program should normally update *blk* before it is used again. Some devices, such as LP, may assign the *blk* argument special meaning. For example, if *blk* = 0, LP: issues a form feed

Request Format:

R0 → area:

11	chan
blk	
buf	
wcnt	
1	

NOTE

When any .WRITE, .WRITC, or .WRITW programmed request is returned, R0 contains the number of words requested if the write is to a sequential-access device (for example, magtape). If the write is to a random-access device (disk or DECtape), R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. The shortened word count is returned in R0. 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).

Errors:

Code	Explanation
0	Attempted to write past end-of-file.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to the example following .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 .WRITC is complete, then control passes to the routine specified in the request. When an RTS PC is encountered in the completion routine, control returns to the user program.

Macro Call: .WRITC area,chan,buf,wcnt,crtn,blk

where:

area is the address of a five-word EMT argument block

chan is a channel number in the range 0 to 376(octal)

buf is the address of the memory buffer to be used for output

wcnt is the number of words to be written

crtn is the address of the completion routine to be entered

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 Software Support Manual* for the significance of the block number for devices such as line printers and paper tape readers

Request Format: WRITC

R0 → area:

11	chan
blk	
buf	
wcnt	
crtn	

NOTE

When any .WRITE, .WRITC, or .WRITW programmed request is returned, R0 contains the number of words requested if the write is to a sequential-access device (for example, magtape). If the write is to a random-access device (disk or DECtape), R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. The shortened word count is returned in R0. If a write goes past EOF on magtape, the handler returns an error 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).

When a .WRITC completion routine is entered, the following conditions are true:

1. R0 contains the contents of the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer: Consequently, the data may be unreliable.
2. R1 contains the octal channel number of the operation. This is useful when the same completion routine is to be used for several different transfers.
3. Registers R0 and R1 are available for use by the routine, but all other registers must be saved and restored. Data cannot be passed between the main program and completion routines in any register or on the stack.

Errors:

Code	Explanation
0	End-of-file on output. Tried to write outside limits of file.
1	Hardware error occurred.
2	Specified channel is not open.

Example:

Refer to the 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

where:

area is the address of a five-word EMT argument block

chan is a channel number in the range 0 to 376(octal)

buf is the address of the buffer to be used for output

wcnt is the number of words to be written. The number must be positive

blk is the block number to be written. For a file-structured .LOOKUP or .ENTER, the block number is relative to the start of the file. For a non-file-structured .LOOKUP or .ENTER, the block number is the absolute block number on the device. Your program should normally update *blk* before it is used again. See the *RT–11 Software Support Manual* for the significance of the block number for devices such as line printers and paper tape readers

Request Format:

R0 → area:

11	chan
blk	
buf	
wcnt	
0	

NOTE

When any .WRITE, .WRITC, or .WRITW programmed request is returned, R0 contains the number of words requested if the write is to a sequential-access device (for example, magtape). If the write is to a random-access device (disk or DECtape), R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. The shortened word count is returned in R0. If a write goes past end-of-file on magtape, the handler returns an error 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:

Refer to the example following .READW.

Chapter 3
System Subroutine Description and Examples

This chapter presents all SYSLIB functions and subroutines in alphabetical order and provides a detailed description of each one. An example of each call in a FORTRAN program is given.

3.1 AJFLT

The AJFLT function converts an INTEGER*4 value to a REAL*4 value and returns that result as the function value.

Form: a = AJFLT (jsrc)

where:

jsrc is the INTEGER*4 variable to be converted

Function Results:

The function result is a REAL*4 value.

Errors:

None.

Example:

The following example converts the INTEGER*4 value contained in JVAL to single precision (REAL*4), multiplies it by 3.5, and stores the result in VALUE.

```
REAL*4 VALUE,AJFLT
INTEGER*4 JVAL
.
.
.
VALUE=AJFLT(JVAL)*3.5
```

3.2 CHAIN

The CHAIN subroutine allows a background program (or any program in the single-job system) to transfer control directly to another background program and pass specified information to it. CHAIN cannot be called from a completion or interrupt routine. The FORTRAN impure area is not preserved across a chain. Therefore, when chaining from one program to another, the information must be reset in the program being chained to. When chaining to any other program, the user should explicitly close the opened logical units with calls to the CLOSE routine. Any routines specified in a FORTRAN USEREX library call are not executed if a CHAIN is accomplished (see Appendix B in the *RT–11/RSTS/E FORTRAN IV User's Guide*).

Form: CALL CHAIN (dblk,var,wcnt)

where:

dblk is the address of a four-word Radix–50 descriptor of the file specification for the program to be run (see the *PDP–11 FORTRAN Language Reference Manual* for the format of the file specification)

var is the first variable (which must start on a word boundary) in a sequence of variables with increasing memory addresses to be passed between programs in the chain parameter area (absolute locations 510 to 777). A single array or a COMMON block (or portion of a COMMON block) is a suitable sequence of variables

wcnt is a word count specifying the number of words (beginning at *var*) to be passed to the called program. The argument *wcnt* may not exceed 60. If no words are passed, then a word count of 0 must be supplied

If the size of the chain parameter area is insufficient, it can be increased by specifying the /B (or /BOTTOM) option to LINK for both the program executing the CHAIN call and the program receiving control.

The data passed can be accessed through a call to the RCHAIN routine. For more information on chaining to other programs, see the .CHAIN programmed request (Section 2.3).

Errors:

None.

Example:

The following example transfers control from the main program to PROG.SAV on DT0, and passes it variables.

```
DIMENSION  SPEC(2)
INTEGER*2  DATA(10)
DATA       SPEC/6RDTOPRO, 6RG  SAV/
.
.
.
CALL CHAIN (SPEC,DATA,10)
```

3.3 CLOSEC/ICLOSE

The CLOSEC subroutine terminates activity on the specified channel and frees it for use in another operation. The handler for the associated device must be in memory. CLOSEC cannot be called from a completion or interrupt routine.

Form: CALL CLOSEC (chan[,i])
i = CLOSEC(chan)
CALL ICLOSE (chan[,i])
i = ICLOSE(chan)

where:

chan — is the channel number to be closed. This argument must be located so that the USR cannot swap over it

i — is the error return if a protection violation occurs

A CLOSEC or PURGE must eventually be issued for any channel opened for input or output. A CLOSEC call specifying a channel that is not open is ignored.

A CLOSEC performed on a file that was opened via an IENTER causes the device directory to be updated to make that file permanent. If the device associated with the specified channel already contains a file with the same name and type, the old copy is deleted when the new file is made permanent. If the file name is protected, then a protection error is generated. A CLOSEC on a file opened via LOOKUP does not require any directory operations.

When an entered file is closed, its permanent length reflects the highest block of the file written since the file was entered; for example, if the highest block written is block number 0, the file is given a length of 1; if the file was never written, it is given a length of 0. If this length is less than the size of the area allocated at IENTER time, the unused blocks are reclaimed as an empty area on the device.

Errors:

i = 0 — Normal return.

= –4 — A protected file with the same name already exists on a device. The CLOSEC is performed, resulting in two files on the device with the same name.

Example:

The following example creates and processes a 56-block file.

```
      REAL*4 DBLK(2)
      DATA DBLK/6RSYONEW,6RFILDAT/
      DATA ISIZE/56/
      .
      .
      .
      ICHAN=IGETC()
      IF(ICHAN.LT.0) GOTO 100
      IERR=IENTER(ICHAN,DBLK,ISIZE)
      IF(IERR.LT.0)GOTO 20
20    GOTO(110,120,130)ABS(IER)
      CALL ICLOSE (ICHAN,I)
      IF(I.EQ.-4) GOTO 200
      CALL IFREEC(ICHAN)
      CALL EXIT
100   STOP 'NO AVAILABLE CHANNELS'
110   STOP 'CHANNEL ALREADY IN USE'
120   STOP 'NOT ENOUGH ROOM ON DEVICE'
130   STOP 'DEVICE IN USE'
200   STOP 'PROTECTION ERROR'
      END
```

3.4 CONCAT

The CONCAT subroutine concatenates two character strings.

Form: CALL CONCAT (a,b,out[,len[,err]])

where:

- a — is the array containing the left string. The string must be terminated with a null byte
- b — is the array containing the right string. The string must be terminated with a null byte
- out — is the array into which the concatenated result is placed. This array must be at least one element longer than the maximum length of the resultant string (that is, one greater than the value of *len*, if specified)
- len — is the integer number of characters representing the maximum length of the output string. The effect of *len* is to truncate the output string to a given length, if necessary
- err — is the logical error flag set if the output string is truncated to the length specified by *len*

CONCAT sets the string in the array *out* to be the string in array *a* immediately followed on the right by the string in array *b* and a terminating null character.

NOTE

Any combination of string arguments is allowed, so long as *b* and *out* do not specify the same array.

Concatenation stops when a null character is detected in *b*, or when the number of characters specified by *len* has been moved.

If either the left or right string is a null string, the other string is copied to *out*. If both are null strings, then *out* is set to a null string. The old contents of *out* are lost when this routine is called.

Errors:

Error conditions are indicated by *err*, if specified. If *err* is given and the output string would have been longer than *len* characters, then *err* is set to .TRUE.; otherwise, *err* is unchanged.

Example:

The following example concatenates the string in array STR and the string in array IN and stores the resultant string in array OUT. OUT cannot be larger than 29 characters.

```
LOGICAL*1 IN(22),OUT(30),STR(7)
.
.
.
CALL CONCAT(STR,IN,OUT,29)
```

3.5 CVTTIM

The CVTTIM subroutine converts a two-word internal format time to hours, minutes, seconds, and ticks.

Form: CALL CVTTIM (time,hrs,min,sec,tick)

where:

time is the two-word internal format time to be converted. If time is considered as a two-element INTEGER*2 array, then:

time (1) is the high-order time
time (2) is the low-order time

hrs is the integer number of hours

min is the integer number of minutes

sec is the integer number of seconds

tick is the integer number of ticks (1/60 of a second for 60-cycle clocks; 1/50 of a second for 50-cycle clocks)

Errors:

None.

Example:

```
INTEGER*4 ITIME
.
.
.
CALL GTIM(ITIME)                    !GET CURRENT TIME-OF-DAY
CALL CVTTIM(ITIME,IHRS,IMIN,ISEC,ITCK)
IF(IHRS.GE.12.AND. IHRS.LT.13) GOTO 100 !TIME FOR LUNCH
```

3.6 DEVICE (FB and XM Only)

The DEVICE subroutine allows you to set up a list of addresses to be loaded with specified values when the program is terminated. If a job terminates or is aborted with a CTRL/C from the terminal, this list is picked up by the system and the appropriate addresses are set up with the corresponding values.

This function is primarily designed to allow user programs to load device registers with necessary values. In particular, it is used to turn off a device's interrupt enable bit when the program servicing the device terminates.

Only one address list can be active at any given time; hence, if multiple DEVICE calls are issued, only the last one has any effect. The list must not be modified by the program after the DEVICE call has been issued, and the list must not be located in an overlay or an area over which the USR swaps.

The second argument of the call (*link*) provides support for a linked list of tables. The *link* argument is optional and causes the first word of the list to be processed as the link word.

Form: CALL DEVICE (ilist[,link])

where:

ilist is an integer array that contains two-word elements, each composed of a one-word address and a one-word value to be put at that address, terminated by a zero word. On program termination, each value is moved to the corresponding address

link is an optional argument that can be any value. This indicates that a linked list table is to be used

If the linked list form is used the first word of the array is the link list pointer

For more information on loading values into device registers, see the .DEVICE programmed request (Section 2.16).

Errors:

None.

Example:

```
INTEGER*2 IDR11(3)          !DEVICE ARRAY SPEC
DATA IDR11(1)/"167770/      !DR11 CSR ADDRESS (OCTAL)
DATA IDR11(2)/0/            !VALUE TO CLEAR INTERRUPT ENABLE
DATA IDR11(3)/0/            !AND END-OF-LIST FLAG
CALL DEVICE(IDR11)          !SET UP FOR ABORT
```

3.7 DJFLT

The DJFLT function converts an INTEGER*4 value into a REAL*8 (DOUBLE PRECISION) value and returns that result as the function value.

Form: d = DJFLT (jsrc)

where:

jsrc specifies the INTEGER*4 variable to be converted

Notes:

If DJFLT is used, it must be defined in the FORTRAN program, either explicitly (REAL*8 DJFLT) or implicitly (IMPLICIT REAL*8 (D)). Without a definition, DJFLT is assumed to be REAL*4 (single precision).

Function Results:

The function result is the REAL*8 value that is the result of the operation.

Errors:

None.

Example:

```
INTEGER*4 JVAL
REAL*8 DJFLT,D
.
.
.
D=DJFLT(JVAL)
```

3.8 GETSTR

The GETSTR subroutine reads a formatted ASCII record from a specified FORTRAN logical unit into a specified array. The data is truncated (trailing blanks removed) and a null byte is inserted at the end to form a character string.

GETSTR can be used in main program routines or in completion routines, but it cannot be used in both at the same time. If GETSTR is used in a completion routine, it cannot be the first I/O operation on the specified logical unit.

Form: CALL GETSTR (lun,out,len,err)

where:

lun is the integer FORTRAN logical unit number of a formatted sequential file from which the string is to be read

out is the array to receive the string; this array must be at least one element longer than *len*

len is the integer number representing the maximum length of the string that is allowed to be input

err is the LOGICAL*1 error flag that is set to .TRUE. if an error occurred. If an error did not occur, the flag is .FALSE

Errors:

Error conditions are indicated by *err*. If *err* is .TRUE., the values returned are as follows:

err = –1 End-of-file for a read operation.
err = –2 Hard error for a read operation.
err = –3 More than *len* bytes were contained in a record.

Example:

The following example reads a string of up to 80 characters from logical unit 5 into the array STRING.

```
LOGICAL*1 STRING(81),ERR
.
.
.
CALL GETSTR(5,STRING,80,ERR)
```

3.9 GTIM

The GTIM subroutine returns the current time of day. The time is returned in two words and is given in terms of clock ticks past midnight. If the system does not have a line clock, a value of 0 is returned. If an RT–11 monitor TIME command has not been entered, the value returned is the time elapsed since the system was bootstrapped, rather than the time of day.

Form: CALL GTIM (itime)

where:

itime is the two-word area to receive the time of day

The high-order time is returned in the first word, the low-order time in the second word. The CVTTIM routine (see Section 3.5) can be used to convert the time into hours, minutes, seconds, and ticks. CVTTIM performs the conversion based on the monitor configuration word for 50- or 60-cycle clocks. Under an FB or XM monitor, the time-of-day is automatically reset after 24:00 when a GTIM is executed; under the single-job monitor, it is not.

Errors

None.

Example:

```
INTEGER*4 JTIME
.
.
.
CALL GTIM(JTIME)
```

3.10 GTJB/IGTJB

The GTJB subroutine returns information about a job in the system.

Form: CALL GTJB (addr,[jobblk [,i]])
i = GTJB (addr,[jobblk])
CALL IGTJB (addr,[jobblk [,i]])
i = IGTJB (addr,[jobblk])

where:

addr is the address of an eight- or twelve-word block into which the parameters are passed

The parameters returned are as follows:

Word 1 Job Number = priority level*2 (background job is 0, system jobs are 2, 4, 6, 10, 12, 14, foreground job is 16 in system job monitors; background job is 0, foreground job is 2 in FB and XM monitors; job number is 0 in SJ monitor)

2 High-memory limit of job partition (last location plus 2)

3 Low-memory limit of job partition (first location)

4 Pointer to I/O channel space

5 Address of job's impure area in FB and XM monitors (0 in SJ)

6	Low byte: unit number of job's console terminal (only if the multiterminal option is present; 0 when the multiterminal feature is not used)
7	Virtual high limit for a job created with the linker /V option (XM only; 0 in SJ and FB and where the Linker /V option is not used)
8–9	Reserved for future use
10–12	ASCII logical job name (system job monitors only)

jobblk is a pointer to a three-word ASCII job name for which data is being requested. Do not specify this argument when requesting the eight-word block

i is an error return if the job is not running

If one argument is used with the call, only the first eight parameters will be passed. For example,

```
INTEGER IJPARM(8)
CALL GTJB (IJPARM)
I = GTJB (IJPARM)
```

At least a comma must follow the argument to pass the information into a 12-word block. For example,

```
INTEGER IJPARM(12)
CALL GTJB (IJPARM,)
I = GTJB (IJPARM,)
```

Errors:

i = 0	Normal return.
= –1	No such job currently running.

Example:

```
      C THIS IS AN EXAMPLE UNDER A SYSTEM
      C JOB MONITOR TO SEE IF THE FOREGROUND
      C JOB IS RUNNING
      DIMENSION JDATA(12)
      .
      .
      .
      I = GTJB (JDATA, 16)
      IF (I.EQ.0) GOTO 20
      TYPE 10
10    FORMAT('NO FG JOB!')
      STOP
20    .
      .
      .
```

3.11 GTLIN

The GTLIN subroutine transfers a line of input from the console terminal or an active indirect command file to the user program. This request allows

you to input information at the console terminal, and it allows the program to operate through indirect files. This subroutine requires the USR. The maximum size of the input line is 80 characters. See the .GTLIN programmed request for setting bits in the Job Status Word to pass lowercase letters and to establish a nonterminating condition.

Form: CALL GTLIN (result[,prompt])

where:

result is the array receiving the string. This LOGICAL*1 array contains a maximum of 80 characters plus 0 as the end indicator, and therefore must be dimensioned to at least 81 elements

prompt is a LOGICAL*1 array containing an optional prompt string to be printed before the input line is received. The string format is the same as that used by the PRINT subroutine. If this argument is not present, no prompt is printed

Errors:

None.

Example:

```
LOGICAL*1 INP(80),PROMT(6)
DATA PROMT /'N','A','M','E','?',"200/
.
.
.
CALL GTLIN(INP,PROMT)
.
.
.
```

3.12 IABTIO

The IABTIO function aborts I/O on a specified channel.

Form: CALL IABTIO (chan)

where:

chan is the channel number for which to abort I/O

Errors:

None.

3.13 IADDR

The IADDR function returns the 16-bit absolute memory address of its argument as the integer function value.

Form: i = IADDR (arg)

where:

arg is the variable or constant whose memory address is to be obtained. The value obtained by passing an expression as *arg* is unpredictable

Errors:

None.

Example:

IADDR can be used to find the address of an assembly language global area. For example:

```
EXTERNAL CAREA
J=IADDR(CAREA)
```

3.14 IAJFLT

The IAJFLT function converts an INTEGER*4 value to a REAL*4 value and stores the result.

Form: i = IAJFLT (jsrc,ares)

where:

jsrc is the INTEGER*4 variable to be converted

ares is the REAL*4 variable or array element to receive the converted value

Function Results:

i =	–1	Normal return; the result is negative.
=	0	Normal return; the result is 0.
=	1	Normal return; the result is positive.

Errors:

i = –2 Significant digits were lost during the conversion.

Example:

```
      INTEGER*4 JVAL
      REAL*4 RESULT
      .
      .
      .
      IF(IAJFLT(JVAL,RESULT).EQ.-2) TYPE 99
99    FORMAT (' OVERFLOW IN INTEGER*4 TO REAL CONVERSION')
```

3.15 IASIGN

The IASIGN function sets information in the FORTRAN logical unit table (overriding the defaults) for use when the FORTRAN Object Time System (OTS) opens the logical unit. This function can be used with ICSI (see

Section 3.20) to allow a FORTRAN program to accept a standard CSI input specification. IASIGN must be called before the unit is opened; that is, before any READ, WRITE, PRINT, TYPE, ACCEPT, or OPEN statements are executed that reference the logical unit.

Form: i = IASIGN (lun,idev[,ifiltyp[,isize[,itype]]])

where:

lun — is an INTEGER*2 variable, constant, or expression specifying the FORTRAN logical unit for which information is being specified

idev — is a one-word Radix–50 device name; this can be the first word of an ICSI input or output file specification

ifiltyp — is a three-word Radix–50 file name and file type; this can be words 2 through 4 of an ICSI input or output file specification

isize — is the length (in blocks) to allocate for an output file; this can be the fifth word of an ICSI output specification. If 0, the larger of either one-half the largest empty segment or the entire second largest empty segment is allocated. If the value specified for length is –1, the entire largest empty segment is allocated

itype — is an integer value determining the optional attributes to be assigned to the file. This value is obtained by adding the values that correspond to the desired operations:

- 1 Use double buffering for output.
- 2 Open the file as a temporary file.
- 4 Force a LOOKUP on an existing file during the first I/O operation. (Otherwise, the first FORTRAN I/O operation determines how the file is opened. Normally if the first I/O operation is a write, an IENTER would be performed on the specified logical unit. A read always causes a LOOKUP.)
- 8 Do not expand carriage control information.
- 16 Expand carriage control information (see Notes below).
- 32 File is read only.

Notes:

Expanded carriage control information applies only to formatted output files and means that the first character of each record is used as a carriage control character when processing a write operation to the given logical unit. The first character is removed from the record and converted to the appropriate ASCII characters to simulate the requested carriage control.

If carriage control information is not expanded, the first character of each record is unmodified and the FORTRAN OTS outputs a line feed, followed by the record, followed by a carriage return.

If carriage control is unspecified, the FORTRAN OTS sends expanded carriage control information to the terminal and line printer and sends unex-

panded carriage control information to all other devices and files. See the *PDP–11 FORTRAN Language Reference Manual* for further carriage control information.

Errors:

i = 0	Normal return.
<> 0	The specified logical unit is already in use, or there is no space for another logical unit association.

Example:

The following example (1) creates an output file on logical unit 3, using the first output file given to the RT–11 Command String Interpreter (CSI), (2) sets up the output file for double buffering, (3) creates an input file on logical unit 4, based on the first input file specification given to the RT–11 CSI, and (4) makes the input file available for read-only access.

```
      INTEGER*2 SPEC(39)
      REAL*4 EXT(2)
      DATA EXT/6RDATDAT,6RDATDAT/    !DEFAULT FILE TYPE IS DAT
      .
      .
      .
10    IF(ICSI(SPEC,EXT,,,0).NE.0) GOTO 10
C
C     DO NOT ACCEPT ANY SWITCHES
C
      CALL IASIGN(3,SPEC(1),SPEC(2),SPEC(5),1)
      CALL IASIGN(4,SPEC(16),SPEC(17),0,32)
```

3.16 ICDFN

The ICDFN function increases the number of input/output channels. Note that ICDFN defines new channels; any channels defined with an earlier ICDFN function are not used. Thus, an ICDFN for 20(decimal) channels (while the 16[decimal] original channels are defined) causes only 20 I/O channels to exist; the space for the original 16 is unused. The space for the new channel area is allocated out of the free space managed by the FORTRAN system.

Form: i = ICDFN (num[,area])

where:

num	is the integer number of channels to be allocated. The number of channels must be greater than 16 and can be a maximum of 256. The program can use all new channels greater than 16 without a call to IGETC; the FORTRAN system input/output uses only the first 16 channels. This argument must be positioned so that the USR cannot swap over it
area	is the space allocated from within the calling program. Under FB and SJ monitors; be sure that the space is outside the USR swapping area. If this argument is not specified, the space for the channels is allocated in the FORTRAN OTS work area

Notes:

1. ICDFN cannot be issued from a completion or interrupt routine.
2. It is recommended that the ICDFN function be used at the beginning of the main program before any I/O operations are initiated.
3. If ICDFN is executed more than once, a completely new set of channels is created each time ICDFN is called.
4. ICDFN requires that extra memory space be allocated to foreground programs (see Section 1.2.4.1).
5. Any channels that were open prior to the ICDFN are copied over to the new set of channel status tables.

Function Results:

i = 0	Normal return.
= 1	An attempt was made to allocate fewer channels than already exist.
= 2	Not enough free space is available for the channel area.

Example:

```
IF(ICDFN(24),EQ,2) STOP 'NOT ENOUGH MEMORY'
```

3.17 ICHCPY (FB and XM Only)

The ICHCPY function opens a channel for input, logically connecting it to a file that is currently open by another job for either input or output. This function can be used by either the foreground or the background job. An ICHCPY must be done before the first read or write for the given channel.

Form: i = ICHCPY (chan,ochan[,jobblk])

where:

chan	is the channel the job will use to read the data. You must obtain this channel through an IGETC call, or you can use channel 16 or higher if you have done an ICDFN call
ochan	is the channel number of the other job that is to be copied
jobblk	is a pointer to a three-word ASCII job name

Notes:

1. If the other job's channel was opened with an IENTER function or a .ENTER programmed request to create a file, your channel indicates a file that extends to the highest block that the creator of the file had written at the time the ICHCPY was executed.
2. A channel that is open on a sequential-access device should not be copied, because buffer requests can become intermixed.

3. Your program can write on a copied channel to a file that is being created by the other job, just as your program could if it were the creator. When your channel is closed, however, no directory update takes place.

Errors:

i	= 0	Normal return.
	= 1	Specified job does not exist or does not have the specified channel (*ochan*) open.
	= 2	Channel (*chan*) is already open.

3.18 ICLOSE

See the SYSLIB subroutine CLOSEC.

3.19 ICMKT

The ICMKT function cancels one or more scheduling requests (made by an ISCHED, ITIMER, or MRKT routine). Support for ICMKT in SJ requires that timer support be created through SYSGEN.

Form: i = ICMKT (id,time)

where:

id	is the identification integer of the request to be canceled. If *id* is equal to 0, all scheduling requests are canceled
time	is the name of a two-word area in which the monitor returns the amount of time remaining in the canceled request

For further information on canceling scheduling requests, see the .CMKT programmed request (Section 2.5).

Errors:

i	= 0	Normal return.
	= 1	*id* was not equal to 0 and no scheduling request with that identification could be found.

Example:

```
INTEGER*4 J
.
.
.
CALL ICMKT(0,J)      !ABORT ALL TIMER REQUESTS NOW
.
.
.
END
```

3.20 ICSI

The ICSI function calls the RT–11 Command String Interpreter in special mode to parse a command string and return file descriptors and options to the program. In this mode, the CSI does not perform any handler IFETCHes, CLOSECs, IENTERs, or LOOKUPs. ICSI cannot be called from a completion or interrupt routine. This subroutine requires the USR.

Form: i = ICSI (filspc,deftyp,[cstring],[option],n)

where:

filspc is the 39-word area to receive the file specifications. The format of this area (considered as a 39-element INTEGER*2 array) is:

Word	1	output file number 1
	4	specification
	5	output file number 1 length
	6	output file number 2
	9	specification
	10	output file number 2 length
	11	output file number 3
	14	specification
	15	output file number 3 length
	16	input file number 1
	19	specification
	20	input file number 2
	23	specification
	24	input file number 3
	27	specification
	28	input file number 4
	31	specification
	32	input file number 5
	35	specification
	36	input file number 6
	39	specification

deftyp is the table of Radix–50 default file types to be assumed when a file is specified without a file type:

deftyp(1)	is the default for all input file types
deftyp(2)	is the default file type for output file number 1
deftyp(3)	is the default file type for output file number 2
deftyp(4)	is the default file type for output file number 3

cstring is the area that contains the ASCIZ command string to be interpreted; the string must end in a zero byte. If the argument is omitted, the system prints the prompt character (*) at the terminal and accepts a command string. If input is from an indirect command file, the next line of that file is used

option — is the name of an INTEGER*2 array dimensioned (4,n) where *n* represents the number of options defined to the program. This argument must be present if the value specified for *n* is non-zero. This array has the following format for the jth option described by the array:

option(1,j)	is the one-character ASCII name of the option
option(2,j)	is set by the routine to 0, if the option did not occur; to 1, if the option occurred without a value; to 2, if the option occurred with a value
option(3,j)	is set to the file number on which the option is specified
option(4,j)	is set to the specified value if option(2,n) is equal to 2

n — is the number of options defined in the array *option*

Notes:

1. The array *option* must be set up to contain the names of the valid options. For example, use the following to set up names for five options:

```
INTEGER*2 SW(4,5)
DATA SW(1,1)/'S'/,SW(1,2)/'M'/,SW(1,3)/'I'/
DATA SW(1,4)/'L'/,SW(1,5)/'E'/
```

2. Multiple occurrences of the same option are supported by allocating an entry in the *option* array for each occurrence of the option. Each time the option occurs in the *option* array, the next unused entry for the named option is used.

3. The arguments of ICSI must be positioned so that the USR cannot swap over them. For more information on calling the Command String Interpreter, see the .CSISPC programmed request (Section 2.11).

Errors:

i = 0	Normal return.
= 1	Illegal command line; no data was returned.
= 2	An illegal device specification occurred in the string.
= 3	An illegal option was specified, or a given option was specified more times than were allowed for in the *option* array.

Example:

The following example causes the program to loop until a valid command is typed at the console terminal.

```
      INTEGER*2 SPEC(39)
      REAL*4 EXT(2)
      DATA EXT/6RDATDAT,6RDATDAT/
      .
      .
      .
10    TYPE 99
99    FORMAT (' ENTER VALID CSI STRING WITH NO OPTIONS')
      IF(ICSI(SPEC,EXT,,,0).NE.0) GOTO 10
```

3.21 ICSTAT

The ICSTAT function obtains information about a channel.

Form: i = ICSTAT (chan,addr)

where:

chan is the channel whose status is desired

addr is a six-word area to receive the status information. The area, as a six-element INTEGER*2 array, has the following format:

Word 1 channel status word
2 starting absolute block number of file on this channel
3 length of file
4 highest block number written since file was opened
5 unit number of device with which this channel is associated
6 Radix–50 of device name with which the channel is associated

Errors:

i = 0 Normal return.
= 1 Channel specified is not open.

Example:

The following example obtains channel status information about channel I.

```
      INTEGER*2 AREA(6)
      I=7
      IF(ICSTAT(I,AREA),NE,0) TYPE 99,I
99    FORMAT(1X,'CHANNEL',I4,'IS NOT OPEN')
```

3.22 IDELET

The IDELET function deletes a named file from an indicated device. IDELET requires the USR and cannot be issued from a completion or interrupt routine.

Form: i = IDELET (chan,dblk[,seqnum])

where:

chan is the channel to be used for the delete operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

dblk is the four-word Radix–50 specification (dev:filnam.typ) for the file to be deleted

seqnum is the file number for cassette operations: if this argument is blank, a value of 0 is assumed

For magtape operation, it describes a file sequence number that can have the following values:

Value	Meaning
–1	This value suppresses rewinding and searching for a file name from the current tape position. Note that if the position is unknown, the handler executes a positioning algorithm that involves backspacing until an end-of-file label is found. The user should not use any other value since all other negative values are reserved for future use.
0	This value rewinds the magtape and spaces forward until the file name is found.
n	Where *n* is any positive number. This value positions the magtape at file sequence number *n*. If the file represented by the file sequence number is greater than two files away from the beginning of the tape, a rewind is performed. If not, the tape is backspaced to the file.

NOTE

The arguments of IDELET must be located so that the USR cannot swap over them.

The specified channel is left inactive when the IDELET is complete. IDELET requires that the handler to be used be resident (via an IFETCH call or a LOAD command from KMON) at the time the IDELET is issued. If the handler is not resident, a monitor error occurs.

For further information on deleting files, see the .DELETE programmed request (Section 2.15).

Errors:

i = 0	Normal return.
= 1	Channel specified is already open.
= 2	File specified was not found.
= 3	Device in use.
= 4	The file is protected and cannot be deleted.

Example:

The following example deletes a file named FTN5.DAT from SY0.

```
REAL*4 FILNAM(2)
DATA FILNAM/6RSYOFTN,6R5  DAT/
.
.
.
I=IGETC()
IF(I.LT.0) STOP 'NO CHANNEL'
CALL IDELET(I,FILNAM)
CALL IFREEC(I)
```

3.23 IDJFLT

The IDJFLT function converts an INTEGER*4 value into a REAL*8 (DOUBLE PRECISION) value and stores the result.

Form: i = IDJFLT (jsrc,dres)

where:

jsrc specifies the INTEGER*4 variable that is to be converted

dres specifies the REAL*8 (or DOUBLE PRECISION) variable to receive the converted value

Function Results:

i = –1 Normal return; the result is negative.
= 0 Normal return; the result is 0.
= 1 Normal return; the result is positive.

Errors:

None.

Example:

```
      INTEGER*4 JJ
      REAL*8 DJ
      .
      .
      .
      IF(IDJFLT(JJ,DJ).LE.0) TYPE 99
99    FORMAT (' VALUE IS NOT POSITIVE')
```

3.24 IDSTAT

The IDSTAT function obtains information about a particular device. It requires the USR and cannot be issued from a completion or interrupt routine.

Form: i = IDSTAT (devnam,cblk)

where:

devnam is the Radix–50 device name

cblk is the four-word area used to store the status information. The area, as a four-element INTEGER*2 array, has the following format:

Word 1 device status word (see Section 2.25)
2 size of handler in bytes
3 entry point of handler (non-zero implies that the handler is in memory)
4 size of the device (in 256-word blocks) for block-replaceable devices; zero for sequential-access devices

NOTE

The arguments of IDSTAT must be positioned so that the USR cannot swap over them.

IDSTAT looks for the device specified by *devnam* and, if found, returns four words of status in *cblk*.

Errors:

i = 0	Normal return.
= 1	Device not found in monitor tables.

Example:

The following example determines whether the line printer handler is in memory. If it is not, the program stops and prints a message to indicate that the handler must be loaded.

```
INTEGER IDNAM
INTEGER*2 CBLK(4)
DATA IDNAM/3RLP /
DATA CBLK/4*0/
CALL IDSTAT(IDNAM,CBLK)
IF(CBLK(3).EQ.0) STOP 'LOAD THE LP HANDLER AND RERUN'
```

3.25 IENTER

The IENTER function allocates space on the specified device and creates a tentative directory entry for the named file. If a file of the same name already exists on the specified device, it is not deleted until the tentative entry is made permanent by CLOSEC or ICLOSE. The file is attached to the channel number specified. This routine requires the USR.

Form: i = IENTER (chan,dblk,length[,seqnum])

where:

chan	is the integer specification for the RT–11 channel to be associated with the file. You must obtain this channel through an IGETC call, or you can use channel 16 or higher if you have done an ICDFN call
dblk	is the four-word Radix–50 descriptor of the file to be operated upon
length	is the integer number of blocks to be allocated for the file. If 0, the larger of either one-half the largest empty segment or the entire second largest empty segment is allocated. If the value specified for length is –1, the entire largest empty segment is allocated (see the .ENTER programmed request, Section 2.28)
seqnum	is a file number for cassette. If this argument is blank, a value of 0 is assumed.

For magtape, it describes a file sequence number that can have the following values:

–2	Rewind the magtape and space forward until the file name is found, or until logical-end-of-tape is detected. The magtape is now positioned correctly. A new logical-end-of-tape is implied.
–1	Space to the logical-end-of-tape and enter file.
0	Rewind the magtape and space forward until the file name is found or the 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.
n	Position magtape at file sequence number *n* if *n* is greater than zero and the file name is not null.

Notes:

1. IENTER cannot be issued from a completion or interrupt routine.
2. IENTER requires that the appropriate device handler be in memory.
3. The arguments of IENTER must be positioned so that the USR does not swap over them.

For further information on creating tentative directory entries, see the .ENTER programmed request (Section 2.28).

Errors:

i = n	Normal return; number of blocks actually allocated (*n* = 0 for non-file-structured IENTER).
= –1	Channel (*chan*) is already in use.
= –2	In a fixed-length request, no space greater than or equal to *length* was found.
= –3	Device in use.
= –4	A file by that name already exists and is protected.
= –5	File sequence number not found.

Example:

The following example allocates a channel for file TEMP.TMP on SY0. If no channel is available, the program prints a message and halts.

```
      REAL*4 DBLK(2)
      DATA DBLK/6RSYOTEM,6RP  TMP/
      ICHAN=IGETC()
      IF(ICHAN.LT.0) STOP 'NO AVAILABLE CHANNEL'
C
C     CREATE TEMPORARY WORK FILE
C
      IF(IENTER(ICHAN,DBLK,20).LT.0) STOP 'ENTER FAILURE'
      .
      .
      .
      CALL PURGE(ICHAN)
      CALL IFREEC(ICHAN)
```

3.26 IFETCH

The IFETCH function loads a device handler into memory from the system device, making the device available for input/output operations. The handler is loaded into the free area managed by the FORTRAN system. Once the handler is loaded, it cannot be released and the memory in which it resides cannot be reclaimed. IFETCH requires the USR and cannot be issued from a completion or interrupt routine. IFETCH issued from a foreground job will fail unless the handler is already in memory.

Form: i = IFETCH (devnam)

where:

devnam — is the one-word Radix–50 name of the device for which the handler is desired. This argument can be the first word of an ICSI input or output file specification. This argument must be positioned so that the USR cannot swap over it

For further information on loading device handlers into memory, see the .FETCH programmed request (Section 2.30).

Errors:

i = 0 — Normal return.
= 1 — Device name specified does not exist.
= 2 — Not enough room exists to load the handler.
= 3 — No handler for the specified device exists on the system device.

Example:

The following example requests that the DX handler be loaded into memory; execution stops if the handler cannot be loaded.

```
REAL*4 IDNAM
DATA IDNAM/3RDX/
.
.
.
IF (IFETCH(IDNAM).NE.0) STOP 'FATAL ERROR FETCHING HANDLER'
```

3.27 IFPROT

The IFPROT function sets or removes file protection for a file.

Form: i = IFPROT (chan,filspc,prot)

where:

chan — is the channel number to be used for the protect operation. You must obtain this channel through an IGETC call, or you can use the channel 16(decimal) or higher if you have done an ICDFN call

filspc — is the file specification of the file to be protected or unprotected, in the format dev:filnam.typ

prot — 1 = protect the file
0 = remove protection from the file

Errors:

i	= 0	Normal return.
	= 1	Channel is in use.
	= 2	File not found.
	= 3	Invalid operation.
	= 4	Invalid *prot* value.

Example:

This example protects the file SY:RT11FB.SYS against deletion.

```
ICHAN = IGETC()              !ALLOCATE CHANNEL
IF (ICHAN.LT.0) STOP 'CANNOT ALLOCATE CHANNEL'
I=IFPROT(ICHAN,'SY:RT11FB.SYS',1)
.
.
.
END
```

3.28 IFREEC

The IFREEC function returns a specified RT–11 channel to the available pool of channels. Before IFREEC is called, the specified channel must be closed or deactivated with a CLOSEC or ICLOSE (see Section 3.3) or a PURGE (see Section 3.92) call. IFREEC cannot be called from a completion or interrupt routine. IFREEC calls must be issued only for channels that have been successfully allocated by IGETC calls; otherwise, the results are unpredictable.

Form: i = IFREEC (chan)

where:

chan is the integer number of the channel to be freed

Errors:

i	= 0	Normal return.
	= 1	Specified channel is not currently allocated.

Example:

See the example under IGETC.

3.29 IGETC

The IGETC function allocates an RT–11 channel, in the range 0 to 15(decimal), to be used by other SYSLIB routines and marks it in use so that the FORTRAN I/O system will not access it. IGETC cannot be issued from a completion or interrupt routine.

Form: i = IGETC()

Function Result:

i = n Channel *n* has been allocated.

Error:

i = –1 No channels are available.

Example:

```
ICHAN=IGETC()                    !ALLOCATE CHANNEL
IF(ICHAN.LT.0) STOP 'CANNOT ALLOCATE CHANNEL'
.
.
.
CALL IFREEC(ICHAN)               !FREE IT WHEN THROUGH
.
.
.
END
```

3.30 IGETSP

The IGETSP subroutine obtains free space from the FORTRAN system and returns the address and size (in number of words) of the allocated space. When this space is obtained, it is allocated for the duration of the program.

Form: i = IGETSP (min,max,iaddr)

where:

min — is the minimum space to be obtained without an error indicating that the desired amount of space is not available

max — is the maximum space to be obtained

iaddr — is the integer specifying the address of the start of the free space (buffer). Note that *iaddr* does not directly denote the storage area as a standard FORTRAN variable would. Rather, it denotes a word that contains the address of the storage space. It is most useful with IPEEK and IPOKE, or with assembly language subroutines

NOTE

Extreme caution should be exercised to avoid using all of the free space allocated by the FORTRAN system. If the FORTRAN system runs out of dynamic free space, fatal errors (Error 29, 30, 42, and so forth) occur. See the *RT–11 System Message Manual.*

Function Results:

i = n — The actual size allocated whose value is *min* .LE. *n* .LE. *max*. The size (*min, max, n*) is specified in words.

Error:

i = –1 — Not enough free space is available to meet the minimum requirements; no allocation was taken from the FORTRAN system free space.

Example:

```
N=IGETSP(256,256,IBUFF)                    !GET 256 WORD BUFFER
IF(N.LT.0) STOP 'CANNOT GET BUFFER SPACE!' !NO SPACE AVAILABLE
```

3.31 IGTJB

See the SYSLIB subroutine GTJB, Section 3.10.

3.32 IJCVT

The IJCVT function converts an INTEGER*4 value to INTEGER*2 format. If *ires* is not specified, the result returned is the INTEGER*2 value of *jsrc*. If *ires* is specified, the result is stored there.

Form: i = IJCVT (jsrc[,ires])

where:

jsrc specifies the INTEGER*4 variable or array element whose value is to be converted

ires specifies the INTEGER*2 entity to receive the conversion result

Function Results (if *ires* is specified):

i = –2 An overflow occurred during conversion.
= –1 Normal return; the result is negative.
= 0 Normal return; the result is 0.
= 1 Normal return; the result is positive.

Errors:

None.

Example:

```
      INTEGER*4 JVAL
      INTEGER*2 IVAL
      .
      .
      .
      IF(IJCVT(JVAL,IVAL).EQ.-2) TYPE 99
99    FORMAT(' NUMBER TOO LARGE IN IJCVT CONVERSION')
```

3.33 ILUN

The ILUN function returns the RT–11 channel number with which a FORTRAN logical unit is associated.

Form: i = ILUN (lun)

where:

lun is an integer expression whose value is a FORTRAN logical unit number in the range 1–99

Function Results:

= +n RT–11 channel number *n* is associated with *lun*.

Errors:

i = –1 Logical unit is not open.
= –2 Logical unit is opened to console terminal.

Example:

```
      PRINT 99
99    FORMAT(' PRINT DEFAULTS TO LOGICAL UNIT 6, WHICH FURTHER DEFAULTS TO LP:')
      ICHAN=ILUN(6)                !WHICH RT-11 CHANNEL IS RECEIVING I/O?
```

3.34 INDEX

The INDEX subroutine searches a source string for the occurrence of a pattern string and returns the character position of the first occurrence of the pattern within the source.

Form: CALL INDEX (a,pattrn,[i],m)

or

m = INDEX (a,pattrn[,i])

where:

a	is the array containing the source string to be searched; it must be terminated by a null byte
pattrn	is the string being sought; it must be terminated by a null byte
i	is the integer starting character position of the search in *a*. If *i* is omitted, *a* is searched beginning at the first character position
m	is an integer variable to store the result of the search; *m* is set to the starting character position of *pattrn* in *a*, if found; otherwise *m* is 0

Errors:

None.

Example:

The following example searches the array STRING for the first occurrence of strings EFG and XYZ and searches the string ABCABCABC for the occurrence of string ABC after position 5.

```
CALL SCOPY('ABCDEFGHI',STRING)      !INITIALIZE STRING
CALL INDEX(STRING,'EFG',,M)         !M=5
CALL INDEX(STRING,'XYZ',,N)         !N=0
CALL INDEX('ABCABCABC','ABC',5,L)   !L=7
```

3.35 INSERT

The INSERT subroutine replaces a portion of one string with another string.

Form: CALL INSERT (in,out,i[,m])

where:

in — is the array containing the string being inserted. The string must be terminated with a null if the number of characters is less than the value of *m* (below), or if *m* is not specified

out — is the array containing the string being modified. The string must be terminated with a null

i — is the integer specifying the character position in *out* at which the insertion begins

m — is the integer maximum number of characters to be inserted

If the maximum number of characters (*m*) is not specified, all characters to the right of the specified character position (*i*) in the string being modified are replaced by the string being inserted. The insert string (*in*) and the string being modified (*out*) can be in the same array only if the maximum number of characters (*m*) is specified and is less than or equal to the difference between the position of the insert (*i*) and the maximum string length of the array.

Errors:

None.

Example:

```
CALL SCOPY('ABCDEFGHIJ',S1)          !INITIALIZE STRING 1
CALL SCOPY(S1,S2)                    !INITIALIZE STRING 2
CALL INSERT('123',S1,6,3)            !S1 = 'ABCDE123IJ'
CALL INSERT('123',S2,4)              !S2 = 'ABC123'
```

3.36 INTSET

The INTSET function establishes a FORTRAN subroutine as an interrupt service routine, assigns it a priority, and attaches it to a vector. INTSET requires that extra memory be allocated to foreground programs that use it (see Section 1.2.4.1).

Form: i = INTSET (vect,pri,id,crtn)

where:

vect — is the integer specifying the address of the interrupt vector to which the subroutine is to be attached

pri — is the integer specifying the actual priority level (4–7) at which the device interrupts

id — is the identification integer to be passed as the single argument to the FORTRAN routine when an interrupt occurs. This allows a single *crtn* to be associated with several INTSET calls

crtn is a FORTRAN subroutine to be established as the interrupt routine. This name should be specified in an EXTERNAL statement in the FORTRAN program that calls INTSET. The subroutine has one argument:

```
SUBROUTINE crtn(id)
INTEGER id
```

When the routine is entered, the value of the integer argument is the value specified for *id* in the appropriate INTSET call

Notes:

1. The *id* argument can be used to distinguish between interrupts from different vectors if the routine to be activated services multiple devices.
2. When using INTSET in FB or XM, the SYSLIB call DEVICE must be used in almost all cases to prevent interrupts from interrupting beyond program termination.
3. If the interrupt routine (*crtn*) has control for a period of time longer than the time in which two more interrupts using the same vector occur, interrupt overrun is considered to have occurred. The error message:

   ?SYSLIB–F–Interrupt overrun

   is printed and the job is aborted. Jobs requiring very fast interrupt response are not viable with FORTRAN, since FORTRAN overhead lowers RT–11's interrupt response rate.
4. The interrupt routine (*crtn*) is actually run as a completion routine by the RT–11 .SYNCH macro. The *pri* argument is used for the RT–11 .INTEN macro.
5. A .PROTECT request is issued for the vector, but no attempt is made to report an error if the vector is already protected; furthermore, the vector is taken over unconditionally. See the .PROTECT programmed request (Section 2.60) for more information.
6. The FORTRAN interrupt service subroutine (*crtn*) cannot call the USR.
7. INTSET cannot be called from a completion or interrupt routine.
8. Interrupt enable should not be set on the associated device until the INTSET call has been successfully executed.

Errors:

i	= 0	Normal return.
	= 1	Invalid vector specification.
	= 2	Reserved for future use.
	= 3	No space is available for the linkage setup.

Example:

```
EXTERNAL CLKSUB                    !SUBR TO HANDLE KW11-P CLOCK
.
.
.
I=INTSET("104,6,0,CLKSUB)          !ATTACH ROUTINE
IF (I.NE.0) GOTO 100               !BRANCH IF ERROR
.
.
.
END
SUBROUTINE CLKSUB(ID)
.
.
.
END
```

3.37 IPEEK

The IPEEK function returns the contents of the word located at a specified absolute 16-bit memory address. This function can examine device registers or any location in memory.

Form: i = IPEEK (iaddr)

where:

iaddr is the integer specification of the absolute address to be examined. If this argument is not an even value, a trap results (except on LSI–11 or a PDP–11/23)

Function Result:

The function result (*i*) is set to the value of the word examined.

Example:

```
ISWIT = IPEEK("177570)      !GET VALUE OF CONSOLE SWITCHES
```

3.38 IPEEKB

The IPEEKB subroutine returns the contents of a byte located at a specified absolute byte address. Since this routine operates in a byte mode, the address supplied can be odd or even. This subroutine can examine device registers or any byte in memory. The return is zero extended, that is, the high byte is 0.

Form: i = IPEEKB (iaddr)

where:

iaddr is the integer specification of the absolute byte address to be examined. Unlike the IPEEK subroutine, the IPEEKB subroutine allows odd addresses

Function Result:

The function result (*i*) is set to the value of the byte examined.

Example:

```
IERR = IPEEKB("53)  !Get error byte
```

3.39 IPOKE

The IPOKE subroutine stores a specified 16-bit integer value into a specified absolute memory location. This subroutine can store values in device registers.

Form: CALL IPOKE (iaddr,ivalue)

where:

iaddr — is the integer specification of the absolute address to be modified. If this argument is not an even value, a trap results (except on LSI–11 or PDP–11/23)

ivalue — is the integer value to be stored in the given address specified by the iaddr argument

Errors:

None.

Example:

The following example displays the value of IVAL in the console display register (this is possible only on certain processors).

```
CALL IPOKE("177570,IVAL)
```

To set bit 12 in the JSW without zeroing any other bits in the JSW, use the following procedure.

```
CALL IPOKE("44,"10000.OR.IPEEK("44))
```

3.40 IPOKEB

The IPOKEB subroutine stores a specified eight-bit integer value into a specified byte location. Since this routine operates in a byte mode, the address supplied can be odd or even. This subroutine can store values in device registers.

Form: CALL IPOKEB (iaddr,ivalue)

where:

iaddr — is the integer specification of the absolute address to be modified. Unlike the IPOKE subroutine, the IPOKEB subroutine allows odd addresses

ivalue — is the integer value to be stored in the given address specified by the *iaddr* argument

Errors:

None.

Example:

```
CALL IPOKEB("53,"20) ! Tell KMON unconditionally fatal error
```

3.41 IPUT

The IPUT function replaces the value of a monitor fixed offset. IPUT uses the monitor .PVAL programmed request.

Form: i = IPUT (ioff,value)

where:

ioff — is the offset (from the base of RMON) to be modified

value — is the integer value to replace the current contents of the offset location

Function Result:

i = — old (replaced) value of the fixed offset location.

Example:

```
ISIZE = IPUT ("314, 100) ! Change default file size used by ENTER
```

3.42 IQSET

The IQSET function is used to make the RT–11 I/O queue larger — that is, to add available elements to the queue. These elements are allocated out of the free space managed by the FORTRAN system. IQSET cannot be called from a completion or interrupt routine.

Form: i = IQSET (qleng[,area])

where:

qleng — is the integer number of elements to be added to the queue. This argument must be positioned so that the USR does not swap over it

area — is the space allocated from within the calling program. Under FB and SJ monitors, make sure that the space is outside the USR swapping area. If this argument is not specified, the space for the elements is allocated in the FORTRAN OTS work area

All RT–11 I/O transfers are done through a centralized queue management system. If I/O traffic is very heavy and not enough queue elements are available, the program issuing the I/O requests is suspended until a queue element becomes available. In an FB or XM system, the other job can run while the first program waits for the element. When IQSET is used in a program to be run in the foreground, the FRUN command must be modified to allocate space for the queue elements (see Section 1.2.4.1).

A general rule to follow is that each program should contain one more queue element than the total number of I/O and timer requests that will be active simultaneously. Timing functions such as ITWAIT and MRKT also cause elements to be used and must be considered when allocating queue elements for a program. Note that if synchronous I/O is done (for example, IREADW/IWRITW) and no timing functions are done, no additional queue elements need be allocated. Note also that FORTRAN IV allocates four queue elements by default.

The following subroutines require queue elements:

IRCVD/IRCVDC/IRCVDF/IRCVDW	ITIMER
IREAD/IREADC/IREADF/IREADW	ITWAIT
ISCHED	IUNTIL
ISDAT/ISDATC/ISDATF/ISDATW	IWRITE/IWRITC/IWRITF/IWRITW
ISLEEP	MRKT
ISPFN/ISPFNC/ISPFNF/ISPFNW	MWAIT

For further information on adding elements to the queue, see the .QSET programmed request.

Errors:

i = 0	Normal return.
= 1	Not enough free space is available for the number of queue elements to be added; no allocation was made.

Example:

```
IF(IQSET(5).NE.0) STOP 'NOT ENOUGH FREE SPACE FOR QUEUE ELEMENTS'
```

3.43 IRAD50

The IRAD50 function converts a specified number of ASCII characters to Radix–50 and returns the number of characters converted. Conversion stops on the first non-Radix–50 character encountered in the input, or when the specified number of ASCII characters have been converted.

Form: n = IRAD50 (icnt,input,output)

where:

n	is the integer number of input characters actually connected
icnt	is the number of ASCII characters to be converted
input	is the area from which input characters are taken
output	is the area in which Radix–50 words are stored

Three characters of text are packed into each word of output. The number of output words modified is computed by the expression (in integer words):

(icnt + 2)/3

Thus, if a count of 4 is specified, two words of output are written even if only a one-character input string is given as an argument.

Function Result:

The integer number of input characters actually converted (*n*) is returned as the function result.

Example:

```
REAL*8 FSPEC
CALL IRAD50(12,'SYOTEMP  DAT',FSPEC)
```

3.44 IRCVD/IRCVDC/IRCVDF/IRCVDW (FB and XM Only)

There are four forms of the receive data function; these are used in conjunction with the ISDAT (send data) functions to allow a general data/message

transfer system. The receive data functions issue RT–11 receive data programmed requests (see Section 2.66). These functions require a queue element; this should be considered when the IQSET function (Section 3.42) is executed.

IRCVD
The IRCVD function receives data and continues execution. The operation is queued and the issuing job continues execution. When the job has to receive the transmitted message, an MWAIT should be executed. This causes the job to be suspended until all pending messages have been received.

Form: i = IRCVD (buff,wcnt)

where:

buff is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVD is complete

wcnt is the maximum integer number of words that can be received

Errors:

i = 0 Normal return.
= 1 No such job exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(41)
.
.
.
CALL IRCVD(MSG,40)
.
.
.
CALL MWAIT
```

IRCVDC
The IRCVDC function receives data and enters an assembly language completion routine when the message is received. The IRCVDC is queued, and program execution stays with the issuing job. When the other job sends a message, the completion routine specified is queued and run according to standard scheduling of completion routines.

Form: i = IRCVDC (buff,wcnt,crtn)

where:

buff is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDC is complete

wcnt is the maximum integer number of words to be received

crtn is the assembly language completion routine to be entered. This name must be specified in a FORTRAN EXTERNAL statement in the routine that issues the IRCVDC call

Errors:

i = 0 Normal return.
= 1 No such job exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

IRCVDF
The IRCVDF function receives data and enters a FORTRAN completion subroutine when the message is received. The IRCVDF is queued, and program execution continues with the issuing job. When the other job sends a message, the FORTRAN completion routine specified is entered.

Form: i = IRCVDF (buff,wcnt,area,crtn)

where:

buff is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDF is complete

wcnt is the maximum integer number of words to be received

area is a four-word area to be set aside for linkage information. This area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion routines when *crtn* has been entered

crtn is the FORTRAN completion routine to be entered. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IRCVDF call

Errors:

i = 0 Normal return.
= 1 No such job exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(41),AREA(4)
EXTERNAL RMSGRT
.
.
.
CALL IRCVDF(MSG,40,AREA,RMSGRT)
```

IRCVDW
The IRCVDW function receives data and waits. This function queues a message request and suspends the job issuing the request until the other

job sends a message. When execution of the issuing job resumes, the message has been received, and the first word of the buffer indicates the number of words transmitted.

Form: i = IRCVDW (buff,wcnt)

where:

buff is the array to be used to buffer the data received. The array must be one word larger than the message to be received because the first word contains the integer number of words actually transmitted when IRCVDW is complete

wcnt is the maximum integer number of words to be received

Errors:

i = 0 Normal return.
= 1 No such job exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(41)
IF(IRCVDW(MSG,40).NE.0) STOP 'UNEXPECTED ERROR'
```

3.45 IREAD/IREADC/IREADF/IREADW

The functions IREAD, IREADC, IREADF, and IREADW transfer a specified number of words from a file into memory. These functions require a queue element, which should be considered when the IQSET function (Section 3.42) is executed.

IREAD
The IREAD function transfers into memory a specified number of words from the file associated with the indicated channel. Control returns to the user program immediately after the IREAD function is initiated. No special action is taken when the transfer is completed.

Form: i = IREAD (wcnt,buff,blk,chan)

where:

wcnt is the relative integer number of words to be transferred

buff is the array to be used as the buffer; this array must contain at least *wcnt* words

blk is the integer block number of the file to be read. The first block of a file is block number 0. The *blk* argument must be updated when necessary. For example, if the program is reading two blocks at a time, *blk* should be updated by 2

chan is the integer specification for the RT–11 channel to be used

When the user program needs to access the data read on the specified channel, an IWAIT function should be issued. This makes sure that the

IREAD operation has been completed. If an error occurred during the transfer, the IWAIT function indicates the error.

Errors:

i = n Normal return; *n* equals the number of words requested (0 for non-file-structured read, multiple of 256[decimal] for file-structured read). If the read is from a magtape, the number of words requested is returned. For example:

If *wcnt* is a multiple of 256 and less than that number of words remain in the file, *n* is shortened to the number of words that remain in the file; thus, if *wcnt* is 512 and only 256 words remain, $i = 256$.

If *wcnt* is not a multiple of 256 and more than *wcnt* words remain in the file, *n* is rounded up to the next block; thus, if *wcnt* is 312 and more than 312 words remain, $i = 512$, but only 312 are read.

If *wcnt* is not a multiple of 256 and less than *wcnt* words remain in the file, *n* equals a multiple of 256 that is the actual number of words being read.

= –1 Attempt to read past end-of-file; no words remain in the file.
= –2 Hardware error occurred on channel.
= –3 Specified channel is not open.

NOTE

If an asynchronous operation on a channel (for example, IREAD) results in end-of-file, the following IWAIT will not detect it. IWAIT detects only hard error conditions. A subsequent operation on that channel will detect end-of-file and returns to the user with the end-of-file error code. Under these conditions, the subsequent operation is not initiated.

Example:

```
      INTEGER*2 BUFFER(256),RCODE,BLK
      .
      .
      .
      RCODE = IREAD(256,BUFFER,BLK,ICHAN)
      IF(RCODE+1) 1010,1000,10
C     IF NO ERROR, START HERE
10    .
      .
      .
      IF(IWAIT(ICHAN).NE.0) GOTO 1010
      .
      .
      .
1000  CONTINUE
C     END OF FILE PROCESSING
      .
      .
      .
      CALL EXIT                  !NORMAL END OF PROGRAM
1010  STOP 'FATAL READ'
      END
```

IREADC

The IREADC function transfers a specified number of words from the indicated channel into memory. Control returns to the user program immediately after the IREADC function is initiated. When the operation is complete, the specified assembly language routine (*crtn*) is entered as an asynchronous completion routine.

Form: i = IREADC (wcnt,buff,blk,chan,crtn)

where:

wcnt	is the integer number of words to be transferred
buff	is the array to be used as the buffer; this array must contain at least *wcnt* words
blk	is the integer block number of the file to be read. The user program normally updates *blk* before it is used again. The first block of a file is block number 0
chan	is the integer specification for the RT–11 channel to be used
crtn	is the assembly language routine to be activated when the transfer is complete. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IREADC call

Errors:

See the errors under IREAD.

Example:

```
INTEGER*2 IBUF(256),RCODE,IBLK
EXTERNAL RDCMP
.
.
.
RCODE=IREADC(256,IBUF,IBLK,ICHAN,RDCMP)
```

IREADF

The IREADF function transfers a specified number of words from the indicated channel into memory. Control returns to the user program immediately after the IREADF function is initiated. When the operation is complete, the specified FORTRAN subprogram (*crtn*) is entered as an asynchronous completion routine (see Section 1.2.1.2).

Form: i = IREADF (wcnt,buff,blk,chan,area,crtn)

where:

wcnt	is the integer number of words to be transferred
buff	is the array to be used as the buffer; this array must contain at least *wcnt* words
blk	is the integer block number of the file to be used. The user program normally updates *blk* before it is used again. The first block of a file is block number 0

chan is the integer specification for the RT–11 channel to be used

area is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program or swapped over by the USR. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

crtn is the FORTRAN routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the routine that issues the IREADF call. Section 1.2.1.2 describes completion routines

Errors:

See the errors under IREAD.

Example:

```
      INTEGER*2 DBLK(4),BUFFER(256),BLKNO
      DATA DBLK/3RDX0,3RINP,3RUT ,3RDAT/,BLKNO/0/
      EXTERNAL RCMPLT
      .
      .
      .
      ICHAN=IGETC()
      IF(ICHAN.LT.0) STOP 'NO CHANNEL AVAILABLE'
      IF(IFETCH(DBLK).NE.0) STOP 'BAD FETCH'
      IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
      .
      .
      .
20    IF(IREADF(256,BUFFER,BLKNO,ICHAN,DBLK,RCMPLT).LT.0) GOTO
100
C     PERFORM OVERLAP PROCESSING
      .
      .
      .
C     SYNCHRONIZER
      CALL IWAIT(ICHAN) !WAIT FOR COMPLETION ROUTINE TO RUN
      BLKNO=BLKNO+1     !UPDATE BLOCK NUMBER
      GOTO 20
      .
      .
      .
C     END OF FILE PROCESSING
100   CALL ICLOSE(ICHAN,I)
      I=ICLOSE()
      CALL IFREEC(ICHAN)
      .
      .
      .
      CALL EXIT
      END
      SUBROUTINE RCMPLT(I,J)
C     THIS IS THE COMPLETION ROUTINE
      .
      .
      .
      RETURN
      END
```

IREADW

The IREADW function transfers a specified number of words from the indicated channel into memory. Control returns to the user program when the transfer is complete or when an error is detected.

Form: i = IREADW (wcnt,buff,blk,chan)

where:

wcnt is the integer number of words to be transferred

buff is the array to be used as the buffer; this array must contain at least *wcnt* words

blk is the integer block number of the file to be read. The user program normally updates *blk* before it is used again

chan is the integer specification for the RT–11 channel to be used

Errors:

See the errors under IREAD.

Example:

```
      INTEGER*2 IBUF(1024)
      .
      .
      .
      ICODE=IREADW(1024,IBUF,IBLK,ICHAN)
      IF(ICODE.EQ.-1) GOTO 100      !END OF FILE PROCESSING AT 100
      IF(ICODE.LT.-1) GOTO 200      !ERROR PROCESSING AT 200
C
C     MODIFY BLOCKS
C
      .
      .
      .
C
C     WRITE THEM OUT
C
      ICODE=IWRITW(1024,IBUF,IBLK,ICHAN)
```

3.46 IRENAM

The IRENAM function causes an immediate change of the name of a specified file.

Form: i = IRENAM (chan,dblk)

where:

chan is the integer specification for the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call. The channel is again available for use once the rename operation is completed

dblk is the eight-word area specifying the name of the existing file and the new name to be assigned. If considered as an eight-element INTEGER*2 array, *dblk* has the form:

Words 1–4 specify the Radix–50 file descriptor for the old file name

Words 5–8 specify the Radix–50 file descriptor for the new file name

NOTE

The arguments of IRENAM must be positioned so that the USR does not swap over them.

If a file already exists with the same name as the new file on the indicated device, it is deleted. IRENAM requires that the handler to be used be resident at the time the IRENAM is issued. If it is not, a monitor error occurs. The device names specified in the file descriptors must be the same.

For more information on renaming files, see the .RENAME programmed request (Section 2.71).

Errors:

i = 0 Normal return.
= 1 Specified channel is already open.
= 2 Specified file was not found.
= 3 A file by that name already exists and is protected.

Example:

```
REAL*8 NAME(2)
DATA NAME/12RDKOFTN2  DAT,12RDKOFTN2  OLD/
.
.
.
ICHAN=IGETC()
IF(ICHAN.LT.0) STOP 'NO CHANNEL'
CALL IRENAM(ICHAN,NAME)      !PRESERVE OLD DATA FILE
CALL IFREEC(ICHAN)
```

3.47 IREOPN

The IREOPN function reassociates a specified channel with a file on which an ISAVES was performed. The ISAVES/IREOPN combination is useful when a large number of files must be operated on at one time. Necessary files can be opened with LOOKUP and their status preserved with ISAVES. When data is required from a file, an IREOPN enables the program to read from the file. The IREOPN need not be done on the same channel as the original LOOKUP and ISAVES.

Form: i = IREOPN (chan,cblk)

where:

chan is the integer specification for the RT–11 channel to be associated with the reopened file; this channel must be initially inactive

cblk is the five-word block where the channel status information was stored by a previous ISAVES. This block, considered as a five-element INTEGER*2 array, has the following format:

Word	1	Channel status word.
	2	Starting block number of the file; zero for non-file-structured devices.
	3	Length of file (in 256-word blocks).
	4	Reserved for future use.
	5	Two information bytes. Even byte: I/O count of the number of requests outstanding on this channel. Odd byte: unit number of the device associated with the channel.

Errors:

i	= 0	Normal return.
	= 1	Specified channel is already in use.

Example:

```
INTEGER*2 SAVES(5,10)
DATA ISVPTR/1/
.
.
.
CALL ISAVES(ICHAN,SAVES(1,ISVPTR))
.
.
.
CALL IREOPN(ICHAN,SAVES(1,ISVPTR))
```

3.48 ISAVES

The ISAVES function stores five words of channel status information into a user-specified array. These words contain all the information that RT–11 requires to completely define a file. When an ISAVES is finished, the data words are placed in memory and the specified channel is closed, so that it is again available for use. When the saved channel data is required, the IREOPN function (Section 3.47) is used.

ISAVES can be used only if a file was opened with a LOOKUP call (see Section 3.79). If IENTER was used, ISAVES returns an error. Note that ISAVES is not legal on magtape or cassette files.

Form: i = ISAVES (chan,cblk)

where:

chan is the integer specification for the RT–11 channel whose status is to be saved. You must obtain this channel through an IGETC call, or you can use channel 16 or higher if you have done an ICDFN call

cblk is a five-word block in which the channel status information describing the open file is stored (see Section 3.47 for the format of this block).

The ISAVES/IREOPN combination is very useful, but care must be exercised when using it. In particular, the following cases should be avoided.

1. If an ISAVES is performed on a file and the same file is then deleted before it is reopened, the space occupied by the file becomes available as an empty space which could then be used by the IENTER function. If this sequence occurs, there is a change in the contents of the file whose status was supposedly saved.
2. Although the handler for the required peripheral need not be in memory for execution of an IREOPN, a fatal error is generated if the handler is not in memory when an IREAD or IWRITE is executed.

Errors:

i = 0 Normal return.
= 1 The specified channel is not currently associated with any file.
= 2 The file was opened with an IENTER call.

Example:

```
INTEGER*2 BLK(5)
.
.
.
IF(ISAVES(ICHAN,BLK).NE.0) STOP 'ISAVES ERROR'
```

3.49 ISCHED

The ISCHED function schedules a specified FORTRAN subroutine to be run as an asynchronous completion routine at a specified time of day. Support for ISCHED in SJ requires timer support.

Form: i = ISCHED (hrs,min,sec,tick,area,id,crtn)

where:

hrs is the integer number of hours

min is the integer number of minutes

sec is the integer number of seconds

tick is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks)

area is a four-word area that must be provided for link information; this area must never be modified by the FORTRAN program, and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

id is the identification integer to be passed to the routine being scheduled

crtn is the name of the FORTRAN subroutine to be entered at the time of day specified. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISCHED call. The subroutine has one argument. For example:

```
SUBROUTINE crtn(id)
INTEGER id
```

When the routine is entered, the value of the integer argument is the value specified for *id* in the appropriate ISCHED call

Notes:

1. The scheduling request made by ISCHED can be canceled at a later time by an ICMKT function call.
2. If the system is busy, the actual time of day that the completion routine is run may be later than the requested time of day.
3. A FORTRAN subroutine can periodically reschedule itself by issuing its own ISCHED or ITIMER calls from within the routine.
4. ISCHED requires a queue element; this should be considered when the IQSET function (Section 3.42) is executed.

Errors:

i	= 0	Normal return.
	= 1	No queue elements available; unable to schedule request.

Example:

```
      INTEGER*2 LINK(4)                    !LINKAGE AREA
      EXTERNAL NOON                        !NAME OF ROUTINE TO RUN
      .
      .
      .
      I=ISCHED(12,0,0,0,LINK,0,NOON)       !RUN SUBR NOON AT 12 PM
      .
      .   (rest of main program)
      .
      END
      SUBROUTINE NOON(ID)
C
C     THIS ROUTINE WILL TERMINATE EXECUTION AT LUNCHTIME,
C     IF THE JOB HAS NOT COMPLETED BY THAT TIME.
C
      STOP      'ABORT JOB -- LUNCHTIME'
      END
```

3.50 ISCOMP

(See SYSLIB subroutine SCOMP.)

3.51 ISDAT/ISDATC/ISDATF/ISDATW (FB and XM Only)

The functions ISDAT, ISDATC, ISDATF, and ISDATW are used with the IRCVD, IRCVDC, IRCVDF, and IRCVDW calls to allow message transfers under the FB or XM monitor. Note that the buffer containing the message

should not be modified or reused until the message has been received by the other job. These functions require a queue element, which should be considered when the IQSET function (see Section 3.42) is executed.

ISDAT

The ISDAT function transfers a specified number of words from one job to the other. Control returns to the user program immediately after the transfer is queued. This call is used with the MWAIT routine (see Section 3.90).

Form: i = ISDAT (buff,wcnt)

where:

buff is the array containing the data to be transferred

wcnt is the integer number of data words to be transferred

Errors:

i = 0 Normal return.
= 1 No such job currently exists in the system. (A job exists as long is it is loadable, whether or not it is active.)

Example:

```
      INTEGER*2 MSG(40)
      .
      .
      .
      CALL ISDAT(MSG,40)
      .
      .
      .
      CALL MWAIT
C     PUT NEW MESSAGE IN BUFFER
```

ISDATC

The ISDATC function transfers a specified number of words from one job to another. Control returns to the user program immediately after the transfer is queued. When the other job accepts the message through a receive data request, the specified assembly language routine (*crtn*) is activated as an asynchronous completion routine.

Form: i = ISDATC (buff,wcnt,crtn)

where:

buff is the array containing the data to be transferred

wcnt is the integer number of data words to be transferred

crtn is the name of an assembly language routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISDATC call

Errors:

i = 0 Normal return.
= 1 No such job currently exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(40)
EXTERNAL RTN
.
.
.
CALL ISDATC(MSG,40,RTN)
```

ISDATF

The ISDATF function transfers a specified number of words from one job to the other. Control returns to the user program immediately after the transfer is queued and execution continues. When the other job accepts the message through a receive data request, the specified FORTRAN subprogram (*crtn*) is activated as an asynchronous completion routine (see Section 1.2.1.2).

Form: i = ISDATF (buff,wcnt,area,crtn)

where:

buff is the array containing the data to be transferred

wcnt is the integer number of data words to be transferred

area is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

crtn is the name of a FORTRAN routine to be activated on completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISDATF call

Errors:

i = 0 Normal return.
= 1 No such job currently exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(40),SPOT(4)
EXTERNAL RTN
.
.
.
CALL ISDATF(MSG,40,SPOT,RTN)
```

ISDATW

The ISDATW function transfers a specified number of words from one job to the other. Control returns to the user program when the other job has accepted the data through a receive data request.

Form: i = ISDATW (buff,wcnt)

where:

buff is the array containing the data to be transferred

wcnt is the integer number of data words to be transferred

Errors:

i = 0 Normal return.
= 1 No such job exists in the system. (A job exists as long as it is loaded, whether or not it is active.)

Example:

```
INTEGER*2 MSG(40)
.
.
.
IF (ISDATW(MSG,40),NE,0) STOP 'FOREGROUND JOB NOT RUNNING'
```

3.52 ISDTTM

The ISDTTM function sets the system date and time. An argument of –1 leaves the corresponding value unchanged.

Form: CALL ISDTTM (date,hitime,lotime)

where:

date is the new system date

hitime is the high-order time of day, in ticks past midnight

lotime is the low-order time of day, in ticks past midnight

Example:

```
.
.
.
C  DEFINE NEW SYSTEM DATE BUT LEAVE TIME UNCHANGED
IDATE = IMONTH*1024+IDAY*32+(IYEAR-1972)
CALL ISDTTM (IDATE, -1, -1)
.
.
.
```

3.53 ISFDAT

The ISFDAT function allows user programs to modify the creation date of an RT–11 file. The device must have an RT–11 file structure.

Form: i = ISFDAT (chan,dblk,idate)

where:

chan is the integer value of the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

dblk is the four word RT–11 file specification, in Radix–50, of the file whose date is being changed

idate is the integer date in RT–11 date format

Errors:

i = 0 Normal return.
= 1 Channel in use.
= 2 File not found.
= 3 Invalid operation.

Example:

This example changes the date of the file DY1:OLD23.DAT to July 4, 1976.

```
REAL*4 FILNAM(2)
DATA FILNAM /6RDY1OLD,6R23 DAT/
IDATE=7*1024 + 4*32 + (1976-1972)     !JULY 4, 1976
ICHAN = IGETC()                       !ALLOCATE CHANNEL
I = ISFDAT(ICHAN,FILNAM,IDATE)        !SET THE DATE
IF (I.NE.0) STOP 'ERROR DURING ISFDAT CALL'
.
.
.
END
```

3.54 ISLEEP

The ISLEEP function suspends the main program execution of a job for a specified amount of time. The specified time is the sum of hours, minutes, seconds, and ticks specified in the ISLEEP call. All completion routines continue to execute.

Form: i = ISLEEP (hrs,min,sec,tick)

where:

hrs is the integer number of hours

min is the integer number of minutes

sec is the integer number of seconds

tick is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks)

Notes:

1. SLEEP requires a queue element, which should be considered when the IQSET function (Section 3.42) is executed.
2. If the system is busy, the time in which execution is suspended may be later than that specified.

Errors:

i = 0 Normal return.
= 1 No queue element available.

Example:

```
.
.
.
CALL IQSET(2)
.
.
.
CALL ISLEEP(0,0,0,4)      !GIVE BACKGROUND JOB SOME TIME
```

3.55 ISPFN/ISPFNC/ISPFNF/ISPFNW

The functions ISPFN, ISPFNC, ISPFNF, and ISPFNW are used in conjunction with special functions to various handlers. They provide a means of doing device-dependent functions, such as rewind and backspace, to those devices. If ISPFN function calls are made to any other devices, the function call is ignored. For more information on programming for specific devices, see the *RT–11 Software Support Manual.*

To use these functions, the handler must be in memory, and a channel must be associated with a file via a non-file-structured LOOKUP call. These functions require a queue element; this should be considered when the IQSET function (Section 3.42) is executed.

ISPFN
The ISPFN function queues the specified operation and immediately returns control to the user program. The IWAIT function can be used to ensure completion of the operation.

Form: i = ISPFN (code,chan[,wcnt,buff,blk])

where:

code is the integer numeric code of the function to be performed (see Table 3–1)

chan is the integer specification for the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

wcnt is the integer number of data words in the operation. This parameter is optional with some ISPFN calls, depending on the particular function. Default value is 0. In magtape operations, it specifies the number of records to space forward or backward. For a backspace operation (*wcnt* = 0), the tape drive backspaces to a tape mark or to the beginning-of-tape. For a forward space operation (*wcnt* = 0), the tape drive forward spaces to a tape mark or the end-of-tape

buff is the array to be used as the data buffer. This parameter is optional with some ISPFN calls, depending on the particular function. Default value is 0

blk is the integer block number of the file to be operated upon. This parameter is optional with some ISPFN calls, depending on the particular function. Default value is 0

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to zero.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block, execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
      .
ERRADR = IADDR (ERRBLK)   !GET THE ADDRESS OF THE 4-WORD ERROR BLOCK
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

The three optional arguments (*wcnt, buff, blk*) are not individually optional. You must have all or none present.

Table 3–1: Functions and Function Codes (Octal)

Function	MS,MT,MM	CT	DX	DM	DY	DL	LD	DU
Read absolute			377	377	377	377		
Write absolute			376	376	376	376		
Write absolute with deleted data			375		375			
Forward to last file		377						
Forward to last block		376						
Forward to next file		375						
Forward to next block		374						
Rewind to load point	373	373						
Write file gap		372						
Write end-of-file	377							
Forward 1 block	376							
Backspace 1 block	375							
Initialize the bad block replacement table				374		374		
Write with extended record gap	374							
Offline	372							
Return volume size				373	373	373	373	373
Read/write translation table							372	372
Write variable size blocks	371							
Direct MSCP access								371
Read variable size blocks	370							
Stream at 100 ips (MS only)	367							

Errors:

i	= 0	Normal return.
	= 1	Attempt to read or write past end-of-file.
	= 2	Hardware error occurred on channel.
	= 3	Channel specified is not open.

Example:

```
CALL ISPFN("373,ICHAN)      !REWIND
```

ISPFNC

The ISPFNC function queues the specified operation and immediately returns control to the user program. When the operation is complete, the specified assembly language routine (*crtn*) is entered as an asynchronous completion routine.

Form: i = ISPFNC (code,chan,wcnt,buff,blk,crtn)

where:

code is the integer numeric code of the function to be performed (see Table 3–1)

chan is the integer specification for the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

wcnt is the integer number of data words in the operation; the default value for this argument is 0

buff is the array to be used as the data buffer; the default value for this argument is 0

blk is the integer block number of the file to be operated upon; this argument must be 0 if not required

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
!GET ADDRESS OF 4-WORD ERROR BLOCK
ERRADR = IADDR (ERRBLK)
ICODE = ISPFNC (CODE,ICHAN,WDCT,BUF,ERRADR)
```

crtn is the name of an assembly language routine to be activated on completion of the operation. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISPFNC call

Errors:

i = 0 Normal return.
= 1 Attempt to read or write past end-of-file.
= 2 Hardware error occurred on channel.
= 3 Channel specified is not open.

Example:

```
EXTERNAL SFCOMP          !NAME OF ASSEMBLY LANGUAGE COMPLETION RTN
.
.
.
ICODE = ISPFNC(CODE,ICHAN,WDCT,BUF,BLK,SFCOMP)
```

ISPFNF
The ISPFNF function queues the specified operation and immediately returns control to the user program. When the operation is complete, the specified FORTRAN subprogram (*crtn*) is entered as an asynchronous completion routine.

Form: i = ISPFNF (code,chan,wcnt,buff,blk,area,crtn)

where:

code is the integer numeric code of the function to be performed (see Table 3–1)

chan is the integer specification for the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

wcnt is the integer number of data words in the operation; this argument must be 0 if not required

buff is the array to be used as the data buffer; this argument must be 0 if not required

blk is the integer block number of the file to be operated upon; this argument must be 0 if not required

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block, execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
!GET THE ADDRESS OF THE 4-WORD ERROR BLOCK
ERRADR = IADDR (ERRBLK)
ICODE = ISPFNF (CODE,ICHAN,WDCT,BUF,ERRADR)
```

area — is a four-word area to be set aside for linkage information; this area must not be modified by the FORTRAN program, and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

crtn — is the name of a FORTRAN routine to be activated on completion of the operation. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the ISPFNF call (Section 1.2.1.2 describes completion routines)

Errors:

i = 0 Normal return.
= 1 Attempt to read or write past end-of-file.
= 2 Hardware error occurred on channel.
= 3 Channel specified is not open.

Example:

```
      REAL*4 MTNAME(2),AREA(2)
      DATA MTNAME/3RMT0,0./
      EXTERNAL DONSUB
      .
      .
      .
      I=IGETC()                  !ALLOCATE CHANNEL
      CALL IFETCH(MTNAME)        !FETCH MT HANDLER
      CALL LOOKUP(I,MTNAME)      !NON-FILE-STRUCTURED LOOKUP ON MT0
      IERR=ISPFNF("373,I,0,0,0,AREA,DONSUB)   !REWIND MAGTAPE
      .
      .
      .
      END
      SUBROUTINE DONSUB
C
C     RUNS WHEN MT0 HAS BEEN REWOUND
C
      .
      .
      .
      END
```

ISPFNW

The ISPFNW function queues the specified operation and returns control to the user program when the operation is complete.

Form: i = ISPFNW (code,chan[,wcnt,buff,blk])

where:

code is the integer numeric code of the function to be performed (see Table 3–1)

chan is the integer specification for the RT–11 channel to be used for the operation. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

wcnt is the integer number of data words in the operation. This parameter is optional with some ISPFNW calls, depending on the function

buff is the array to be used as the data buffer. This parameter is optional with some ISPFNW calls, depending on the function

blk is the integer block number of the file to be operated upon. This parameter is optional with some ISPFNW calls, depending on the function

When this argument is supplied by magtape, it is the address of a four-word error and status block used for returning the exception conditions. The four words must be initialized to 0.

The error and status block must always be mapped when running in the XM monitor, and the USR must not swap over it. To obtain the address of the error block execute the following instructions:

```
INTEGER*2       ERRADR, ERRBLK(4)
DATA ERRBLK     /0,0,0,0,/
      .
      .
      .
!GET THE ADDRESS OF THE 4-WORD ERROR BLOCK
ERRADR = IADDR (ERRBLK)
ICODE = ISPFN (CODE,ICHAN,WDCT,BUF,ERRADR)
```

Errors:

i = 0 Normal return.
= 1 Attempt to read or write past end-of-file.
= 2 Hardware error occurred on channel.
= 3 Channel specified is not open.

Example:

```
      INTEGER*2 BUF(65),TRACK,SECTOR,DBLK(4)
      DATA DBLK/3RDX0,0,0,0/
      .
      .
      .
      ICHAN=IGETC()
      IF(ICHAN.LT.0) STOP 'NO CHANNEL AVAILABLE'
      IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
      .
      .
      .
C     READ AN ABSOLUTE TRACK AND SECTOR FROM THE FLOPPY
C
```

```
      ICODE=ISPFNW("377,ICHAN,TRACK,BUF,SECTOR)
C
C     BUF(1) IS THE DELETED DATA FLAG
C     BUF(2-65) IS THE DATA
```

3.56 ISPY

The ISPY function returns the integer value of the word at a specified offset from the RT–11 resident monitor. This subroutine uses the .GVAL programmed request to return fixed monitor offsets. (See the *RT–11 Software Support Manual* for information on fixed offset references.)

Form: i = ISPY (ioff)

where:

ioff is the offset (from the base of RMON) to be examined

Function Result:

The function result (*i*) is set to the value of the word examined.

Example:

```
C
C     BRANCH TO 200 IF RUNNING UNDER FB MONITOR
C
      IF(ISPY("300),AND,1) GOTO 200
C
C     WORD AT OCTAL 300 FROM RMON IS
C     THE CONFIGURATION WORD.
```

3.57 ITIMER

The ITIMER function schedules a specified FORTRAN subroutine to be run as an asynchronous completion routine after a specified time interval has elapsed. This request is supported by SJ when the timer support special feature is included during system generation.

Form: i = ITIMER (hrs,min,sec,tick,area,id,crtn)

where:

hrs is the integer number of hours

min is the integer number of minutes

sec is the integer number of seconds

tick is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks)

area is a four-word area that must be provided for link information; this area must never be modified by the FORTRAN program, and the USR must never swap over it. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

id is the identification integer to be passed to the routine being scheduled

crtn is the name of the FORTRAN subroutine to be entered when the specified time interval elapses. This name must be speci-

fied in an EXTERNAL statement in the FORTRAN routine that references ITIMER. The subroutine has one argument. For example:

```
SUBROUTINE crtn(id)
INTEGER id
```

When the routine is entered, the value of the integer argument is the value specified for *id* in the appropriate ITIMER call

Notes:

1. This function can be canceled at a later time by an ICMKT function call.
2. If the system is busy, the actual time interval after which the completion routine is run can be longer than the time interval requested.
3. FORTRAN subroutines can periodically reschedule themselves by issuing ISCHED or ITIMER calls.
4. ITIMER requires a queue element, which should be considered when the IQSET function (Section 3.42) is executed.

For more information on scheduling completion routines, see Section 1.2.1.2 and the .MRKT programmed request, Section 2.45.

Errors:

i	= 0	Normal return.
	= 1	No queue elements available; unable to schedule request.

Example:

```
      INTEGER*2 AREA(4)
      EXTERNAL WATCHD
      .
      .
      .
C     IF THE CODE FOLLOWING ITIMER DOES NOT REACH THE ICMKT CALL
C     IN 12 MINUTES, WATCH DOG COMPLETION ROUTINE WILL BE
C     ENTERED WITH ID OF 3
C
      CALL ITIMER(0,12,0,0,AREA,3,WATCHD)
      .
      .
      .
      CALL ICMKT(3,AREA)
      .
      .
      .
      END
      SUBROUTINE WATCHD(ID)
C
C     THIS IS CALLED AFTER 12 MINUTES
      .
      .
      .
      RETURN
      END
```

3.58 ITLOCK (FB and XM Only)

The ITLOCK function is used in an FB or XM system to attempt to gain ownership of the USR. It is similar to LOCK (Section 3.78) in that, if successful, the user job returns with the USR in memory. However, if a job attempts to LOCK the USR while the other job is using it, the requesting job is suspended until the USR is free. With ITLOCK, if the USR is not available, control returns immediately and the lock failure is indicated. ITLOCK cannot be called from a completion or interrupt routine.

Form: i = ITLOCK()

For further information on gaining ownership of the USR, see the .TLOCK programmed request (Section 2.88).

Errors:

i	= 0	Normal return.
	= 1	USR is already in use.

Example:

```
IF(ITLOCK(),NE,0) GOTO 100      !GOTO 100 IF USR BUSY
```

3.59 ITTINR

The ITTINR function transfers a character from the console terminal to the user program. If no characters are available, system action is determined by the setting of bit 6 of the Job Status Word.

Form: i = ITTINR()

If the function result (*i*) is less than 0 when execution of the ITTINR function is complete, it indicates that no character was available. Under the FB or XM monitor, ITTINR does not return a result of less than zero unless bit 6 of the Job Status Word was on when the request was issued.

There are two modes of doing console terminal input, and they are governed by bit 12 of the Job Status Word (JSW). The JSW is at octal location 44. If bit 12 is 0, normal I/O is performed under the following conditions:

1. The monitor echoes all characters typed.
2. CTRL/U and RUBOUT perform line deletion and character deletion, respectively.
3. A carriage return, line feed, CTRL/Z, or CTRL/C must be struck before characters on the current line are available to the program. When one of these is typed, characters on the line typed are passed one by one to the user program.

If the console is in special mode (bit 12 set to 1), the following conditions apply:

1. The monitor does not echo characters typed except for CTRL/C and CTRL/O.
2. CTRL/U and RUBOUT do not perform special functions.

3. Characters are immediately available to the program.

4. No ALTMODE conversion is done.

In special mode, the user program must echo the characters desired. However, CTRL/C and CTRL/O are acted on by the monitor in the usual way.

Bit 12 in the JSW must be set by the user program if special console mode is desired. Bit 14 in the JSW must be set if lowercase characters are desired. These bits are cleared when control returns to RT–11.

Regardless of the setting of bit 12, when a carriage return is entered, both carriage return and line feed characters are passed to the program; if bit 12 is 0, these characters will be echoed.

Lowercase conversion is determined by the setting of bit 14. If bit 14 is 0, lowercase characters are converted to uppercase before being echoed (if bit 12 is 0) and passed to a program; if bit 14 is 1, lowercase characters are echoed (if bit 12 is 0) and passed as received. Bit 14 is cleared when the program terminates.

NOTE

To set and/or clear bits in the JSW, do an IPEEK and then an IPOKE (see example under IPOKE). In special terminal mode (JSW bit 12 set), normal FORTRAN formatted I/O from the console is undefined.

In the FB or XM monitor, CTRL/F and CTRL/B (and CTRL/X in monitors with the system job feature) are not affected by the setting of bit 12. The monitor always acts on these characters if the SET TT FB command is in effect.

Also under the FB or XM monitor, if a terminal input request is made and no character is available, job execution is normally suspended until a character is ready. If a program requires execution to continue and ITTINR to return a result of less than zero, it must turn on bit 6 of the JSW before the ITTINR. Bit 6 is cleared when a program terminates. The results of ITTINR must be stored in an INTEGER type variable for the purposes of error checking. Once it is known that the call did not have an error return, the result can be moved into a LOGICAL*1 variable or array element. Direct placement into a LOGICAL*1 variable will lead to incorrect results, because the negative flag (bit 15 set) is lost in conversion to a LOGICAL*1 variable.

Function Results:

i >0	Character read.
<0	No character available.

Example:

```
ICHAR=ITTINR()            !READ A CHARACTER FROM THE CONSOLE
IF(ICHAR.LT.0) GOTO 100   !CHARACTER NOT AVAILABLE
```

3.60 ITTOUR

The ITTOUR function transfers a character from the user program to the console terminal if there is room for the character in the monitor buffer. If it is not currently possible to output a character, an error flag is returned.

Form: i = ITTOUR (char)

where:

char is the character to be output, right-justified in the integer (can be LOGICAL*1 entity if desired)

If the function result (*i*) is 1 when execution of the ITTOUR function is complete, it indicates that there is no room in the buffer and that no character was output. Under the FB or XM monitor, ITTOUR normally does not return a result of 1. Instead, the job is blocked until room is available in the output buffer. If a job requires execution to continue and a result of 1 to be returned, it must turn on bit 6 of the JSW (location 44) before issuing the request.

NOTE

If a foreground job has characters in the TT output buffer, they are not output under the following conditions:

1. If a background job is doing output to the console TT, the foreground job cannot output characters from its buffer until the background job outputs a line feed character. This can be troublesome if the console device is a graphics terminal and the background job is doing graphic output without sending any line feeds.
2. If no background job is running (that is, KMON is in control of background), the foreground job cannot output its characters until the user types a carriage return or a line feed. In the former case, KMON gets control again and locks out foreground output as soon as the foreground output buffer is empty.

Note that the use of PRINT eliminates these problems.

Function Results:

i = 0 Character was output.
= 1 Ring buffer is full.

Example:

```
      DO 20 I=1,5
10    IF(ITTOUR("007),NE,0) GOTO 10      !RING BELL 5 TIMES
20    CONTINUE
```

3.61 ITWAIT (SYSGEN Option in SJ)

The ITWAIT function suspends the main program execution of the current job for a specified time interval. All completion routines continue to execute.

Form: i = ITWAIT (itime)

where:

itime	is the two-word internal format time interval
itime (1)	is the high-order time
itime (2)	is the low-order time

Notes:

1. WAIT requires a queue element, which should be considered when the IQSET function (Section 3.42) is executed.
2. If the system is busy, the actual time interval during which execution is suspended may be longer than the time interval specified.

Errors:

i = 0	Normal return.
= 1	No queue element available.

Example:

```
INTEGER*2 TIME(2)
.
.
.
CALL ITWAIT(TIME)      !WAIT FOR TIME
```

3.62 IUNTIL (SYSGEN Option in SJ)

The IUNTIL function suspends main program execution of the job until the time of day specified. All completion routines continue to run.

Form: i = IUNTIL (hrs,min,sec,tick)

where:

hrs	is the integer number of hours
min	is the integer number of minutes
sec	is the integer number of seconds
tick	is the integer number of ticks (1/60 of a second on 60-cycle clocks; 1/50 of a second on 50-cycle clocks)

Notes:

1. IUNTIL requires a queue element, which should be considered when the IQSET function (Section 3.39) is executed.
2. If the system is busy, the actual time of day that the program resumes execution may be later than that requested.

Errors:

i = 0	Normal return.
= 1	No queue element available.

Example:

```
C     TAKE A LUNCH BREAK
      CALL IUNTIL(13,0,0,0)      !START UP AGAIN AT 1 P.M.
```

3.63 IVERIF

See SYSLIB subroutine VERIFY, Section 3.113.

3.64 IWAIT

The IWAIT function suspends execution of the main program until all input/output operations on the specified channel are complete. This function is used with IREAD, IWRITE, and ISPFN calls. Completion routines continue to execute.

Form: i = IWAIT (chan)

where:

chan is the integer specification for the RT–11 channel to be used. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

For further information on suspending execution of the main program, see the .WAIT programmed request (Section 2.96).

Errors:

i = 0 Normal return.
= 1 Channel specified is not open.
= 2 Hardware error occurred during the previous I/O operation on this channel.

Example:

```
IF(IWAIT(ICHAN).NE.0) CALL IOERR(4)
```

3.65 IWRITE/IWRITC/IWRITF/IWRITW

The functions IWRITE, IWRITC, IWRITF, and IWRITW transfer a specified number of words from memory to the specified channel. The IWRITE functions require queue elements; this should be considered when the IQSET function (Section 3.42) is executed.

IWRITE
The IWRITE function transfers a specified number of words from memory to the specified channel. Control returns to the user program immediately after the request is queued. No special action is taken upon completion of the operation.

Form: i = IWRITE (wcnt,buff,blk,chan)

where:

wcnt is the integer number of words to be transferred

buff is the array to be used as the output buffer

blk is the integer block number of the file to be written. The user program normally updates *blk* before it is used again

chan is the integer specification for the RT–11 channel to be used. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

Errors:

i = n Normal return; *n* equals the number of words written, rounded to a multiple of 256 (0 for non-file-structured writes).

NOTE

If the word count returned is less than that requested, an implied end-of-file has occurred although the normal return is indicated.

= –1 Attempt to write past end-of-file; no more space is available in the file.
= –2 Hardware error occurred.
= –3 Channel specified is not open.

Example:

Refer to the example for IREAD.

IWRITC
The IWRITC function transfers a specified number of words from memory to the specified channel. The request is queued and control returns to the user program. When the transfer is complete, the specified assembly language routine (*crtn*) is entered as an asynchronous completion routine.

Form: i = IWRITC (wcnt,buff,blk,chan,crtn)

where:

wcnt is the relative integer number of words to be transferred

buff is the array to be used as the output buffer

blk is the relative integer block number of the file to be written. The user program normally updates *blk* before it is used again (for example, if the program is writing two blocks at a time, *blk* should be updated by 2)

chan is the relative integer specification for the RT–11 channel to be used. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

crtn is the name of the assembly language routine to be activated upon completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IWRITC call

Errors:

See the errors under IWRITE.

Example:

```
INTEGER*2 IBUF(256)
EXTERNAL CRTN
.
.
.
ICODE=IWRITC(256,IBUF,IBLK,ICHAN,CRTN)
```

IWRITF

The IWRITF function transfers a number of words from memory to the specified channel. The transfer request is queued and control returns to the user program. When the operation is complete, the specified FORTRAN subprogram (*crtn*) is entered as an asynchronous completion routine (see Section 1.2.1.2).

Form: i = IWRITF (wcnt,buff,blk,chan,area,crtnn)

where:

wcnt is the integer number of words to be transferred

buff is the array to be used as the output buffer

blk is the integer block number of the file to be written. The user program normally updates *blk* before it is used again

chan is the integer specification for the RT–11 channel to be used. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

area is a four-word area to be set aside for link information; this area must not be modified by the FORTRAN program, and the USR must not swap over it. This area can be reclaimed by other FORTRAN completion functions when *crtn* has been activated

crtn is the name of the FORTRAN routine to be activated upon completion of the transfer. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the IWRITF call (Section 1.2.1.2 describes completion routines)

Errors:

See the errors under IWRITE.

Example:

Refer to the example under IREADF, Section 3.45.

IWRITW

The IWRITW function transfers a specified number of words from memory to the specified channel. Control returns to the user program when the transfer is complete.

Form: i = IWRITW (wcnt,buff,blk,chan)

where:

wcnt	is the integer number of words to be transferred
buff	is the array to be used as the output buffer
blk	is the integer block number of the file to be written. The user program normally updates *blk* before it is used again
chan	is the integer specification for the RT–11 channel to be used. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

Errors:

See the errors under IWRITE.

Example:

Refer to the example under IREADW, Section 3.45.

3.66 JADD

The JADD function computes the sum of two INTEGER*4 values.

Form: i = JADD (jopr1,jopr2,jres)

where:

jopr1	is an INTEGER*4 variable
jopr2	is an INTEGER*4 variable
jres	is an INTEGER*4 variable that receives the sum of *jopr1* and *jopr2*

Function Results:

i = –1	Normal return; the result is negative.
= 0	Normal return; the result is zero.
= 1	Normal return; the result is positive.

Errors:

i = –2 An overflow occurred while computing the result.

Example:

```
INTEGER*4 JOP1,JOP2,JRES
.
.
.
IF(JADD(JOP1,JOP2,JRES).EQ.-2) GOTO 100
```

3.67 JAFIX

The JAFIX function converts a REAL*4 value to INTEGER*4.

Form: i = JAFIX (asrc,jres)

where:

asrc is a REAL*4 variable, constant, or expression to be converted to INTEGER*4

jres is an INTEGER*4 variable that is to contain the result of the conversion

Function Results:

i = –1 Normal return; the result is negative.
= 0 Normal return; the result is zero.
= 1 Normal return; the result is positive.

Errors:

i = –2 An overflow occurred while computing the result.

Example:

```
      INTEGER*4 JOP1
C     READ A LARGE INTEGER FROM THE TERMINAL
      ACCEPT 99,A
99    FORMAT (F15.0)
      IF(JAFIX(A,JOP1).EQ.-2) GOTO 100
      .
      .
      .
```

3.68 JCMP

The JCMP function compares two INTEGER*4 values and returns an INTEGER*2 value that reflects the signed comparison result.

Form: i = JCMP (jopr1,jopr2)

where:

jopr1 is the INTEGER*4 variable or array element that is the first operand in the comparison

jopr2 is the INTEGER*4 variable or array element that is the second operand in the comparison

Function Results:

i = –1 If *jopr1* is less than *jopr2*.
= 0 If *jopr1* is equal to *jopr2*.
= 1 If *jopr1* is greater than *jopr2*.

Errors:

None.

Example:

```
INTEGER*4 JOPX,JOPY
.
.
.
IF(JCMP(JOPX,JOPY)) 10,20,30
```

3.69 JDFIX

The JDFIX function converts a REAL*8 (DOUBLE PRECISION) value to INTEGER*4.

Form: i = JDFIX (dsrc,jres)

where:

dsrc is a REAL*8 variable, constant, or expression to be converted to INTEGER*4

jres is an INTEGER*4 variable to contain the conversion result

Function Results:

i = –1 Normal return; the result is negative.
= 0 Normal return; the result is zero.
= 1 Normal return; the result is positive.

Errors:

i = –2 An overflow occurred while computing the result.

Example:

```
      INTEGER*4 JNUM
      REAL*8 DPNUM
      .
      .
      .
20    TYPE 98
98    FORMAT(' ENTER POSITIVE INTEGER')
      ACCEPT 99,DPNUM
99    FORMAT(F20.0)
      IF(JDFIX(DPNNUM,JNUM).LT.0) GOTO 20
      .
      .
      .
```

3.70 JDIV

The JDIV function computes the quotient of two INTEGER*4 values.

Form: i = JDIV (jopr1,jopr2,jres[,jrem])

where:

jopr1 is an INTEGER*4 variable that is the dividend of the operation

jopr2 is an INTEGER*4 variable that is the divisor of *jopr1*

jres is an INTEGER*4 variable that receives the quotient of the operation (that is, *jres=jopr1/jopr2*)

jrem is an INTEGER*4 variable that receives the remainder of the operation. The sign is the same as that for *jopr1*

Function Results:

i	= –1	Normal return; the quotient is negative.
	= 0	Normal return; the quotient is 0.
	= 1	Normal return; the quotient is positive.

Errors:

i	= –3	An attempt was made to divide by 0.

Example:

```
INTEGER*4 JN1,JN2,JQUO
.
.
.
CALL JDIV(JN1,JN2,JQUO)
.
.
.
```

3.71 JICVT

The JICVT function converts a specified INTEGER*2 value to INTEGER*4.

Form: i = JICVT (isrc,jres)

where:

isrc is the INTEGER*2 quantity to be converted

jres is the INTEGER*4 variable or array element to receive the result

Function Results:

i	= –1	Normal return; the result is negative.
	= 0	Normal return; the result is 0.
	= 1	Normal return; the result is positive.

Errors:

None.

Example:

```
INTEGER*4 JVAL
CALL JICVT(478,JVAL)      !FORM A 32-BIT CONSTANT
```

3.72 JJCVT

The JJCVT function interchanges words of an INTEGER*4 value to form an internal format time or vice versa. This procedure is necessary when the INTEGER*4 variable is to be used as an argument in a timer-support function such as ITWAIT. When a two-word internal format time is specified to a function such as ITWAIT, it must have the high-order time as the first word and the low-order time as the second word.

Form: CALL JJCVT (jsrc)

where:

jsrc is the INTEGER*4 variable whose contents are to be interchanged

Errors:

None.

Example:

```
INTEGER*4 TIME
.
.
.
CALL GTIM(TIME)      !GET TIME OF DAY
CALL JJCVT(TIME)     !TURN IT INTO INTEGER*4 FORMAT
```

3.73 JMOV

The JMOV function assigns the value of an INTEGER*4 variable to another INTEGER*4 variable and returns the sign of the value moved.

Form: i = JMOV (jsrc,jdest)

where:

jsrc is the INTEGER*4 variable whose contents are to be moved

jdest is the INTEGER*4 variable that is the target of the assignment

Function Results:

The value of the function is an INTEGER*2 value that represents the sign of the result as follows:

i = –1 Normal return; the result is negative.
= 0 Normal return; the result is 0.
= 1 Normal return; the result is positive.

Errors:

None.

Example:

The JMOV function allows an INTEGER*4 quantity to be compared with 0 by using it in a logical IF statement. For example:

```
INTEGER*4 INT1
.
.
.
IF(JMOV(INT1,INT1),NE,0) GOTO 300 !GO TO STMT 300 IF INT1 NOT 0
```

3.74 JMUL

The JMUL function computes the product of two INTEGER*4 values.

Form: i = JMUL (jopr1,jopr2,jres)

where:

jopr1 is an INTEGER*4 variable that is the multiplicand

jopr2 is an INTEGER*4 variable that is the multiplier

jres is an INTEGER*4 variable that receives the product of the operation

Function Results:

i = –1 Normal return; the product is negative.
= 0 Normal return; the product is 0.
= 1 Normal return; the product is positive.

Errors:

i = –2 An overflow occurred while computing the result.

Example:

```
      INTEGER*4 J1,J2,JRES
      .
      .
      .
      IF(JMUL(J1,J2,JRES)+1) 100,10,20
C     GOTO 100 IF OVERFLOW
C     GOTO 10 IF RESULT IS NEGATIVE
C     GOTO 20 IF RESULT IS POSITIVE OR ZERO
```

3.75 JSUB

The JSUB function computes the difference between two INTEGER*4 values.

Form: i = JSUB (jopr1,jopr2,jres)

where:

jopr1 is an INTEGER*4 variable that is the minuend of the operation

jopr2 is an INTEGER*4 variable that is the subtrahend of the operation

jres is an INTEGER*4 variable that is to receive the difference between *jopr1* and *jopr2* (that is, *jres=jopr1–jopr2*)

Function Results:

i	= –1	Normal return; the result is negative.
	= 0	Normal return; the result is 0.
	= 1	Normal return; the result is positive.

Errors:

i	= –2	An overflow occurred while computing the result.

Example:

```
INTEGER*4 JOP1,JOP2,J3
.
.
.
CALL JSUB(JOP1,JOP2,J3)
```

3.76 JTIME

The JTIME subroutine converts the time specified to the internal two-word format time.

Form: CALL JTIME (hrs,min,sec,tick,time)

where:

hrs	is the integer number of hours
min	is the integer number of minutes
sec	is the integer number of seconds
tick	is the integer number of ticks (1/60 of a second for 60-cycle clocks; 1/50 of a second for 50-cycle clocks)
time	is the two-word area to receive the internal format time: time(1) is the high-order time; time(2) is the low-order time

Errors:

None.

Example:

```
      INTEGER*4 J1
      .
      .
      .
C     CONVERT 3 HRS, 7 MIN, 23 SECONDS TO INTEGER *4 VALUE
      CALL JTIME(3,7,23,0,J1)
      CALL JJCVT(J1)
```

3.77 LEN

The LEN function returns the number of characters currently in the string contained in a specified array. This number is computed as the number of characters preceding the first null byte encountered. If the specified array contains a null string, a value of 0 is returned.

Form: i = LEN (a)

where:

a specifies the array containing the string, which must be terminated by a null byte

Errors:

None.

Example:

```
      LOGICAL*1 STRNG(73)
      .
      .
      .
      TYPE 99,(STRNG(I),I=1,LEN(STRNG))
99    FORMAT('0',132A1)
```

3.78 LOCK

The LOCK subroutine keeps the USR in memory for a series of operations involving various RT–11 file management functions.

If all the conditions that cause swapping are satisfied, a portion of the user program is written out to the disk file SWAP.SYS and the USR is loaded. Otherwise, the USR in memory is used, and no swapping occurs. The USR is not released until an UNLOCK (see Section 3.112) is given. (Note that in an FB system, calling the CSI can also perform an implicit UNLOCK.) To save time in swapping, a program that has many USR requests to make can LOCK the USR in memory, make all the requests, and then UNLOCK the USR.

In an FB or XM environment, a LOCK inhibits another job from using the USR. Thus, the USR should be locked only for as long as necessary.

NOTE

If any job does a LOCK, it can cause the USR to be unavailable for other jobs for a considerable period of time. The USR is not reentrant and only one job has use of the USR at a time, which should be considered for systems requiring concurrent foreground and background jobs. This is particularly true when magtape and/or cassette are active.

File operations by the USR require a sequential search of the tape for magtape and cassette. This could lock out the foreground job for a long time while the background job does a tape operation. The programmer should keep this in mind when designing such systems. The FB and XM monitors supply the ITLOCK routine, which permits the foreground job to check for the availability of the USR.

Form: CALL LOCK

After a LOCK has been executed, the UNLOCK routine must be executed to release the USR from memory. The LOCK/UNLOCK routines are complementary and must be matched. That is, if three LOCKs are issued, at least three UNLOCKs must be done, otherwise the USR is not released. More UNLOCKs than LOCKs can occur without error; the extra UNLOCKs are ignored.

Notes:

1. It is vital that the LOCK call not come from within the area into which the USR will be swapped. If this should occur, the return from the USR request would not be to the user program, but to the USR itself, since the LOCK function causes part of the user program to be saved on disk and replaced in memory by the USR. Furthermore, subroutines, variables, and arrays in the area where the USR is swapping should not be referenced while the USR is locked in memory.
2. Once a LOCK has been performed, it is not advisable for the program to destroy the area the USR is in, even though no further use of the USR is required. This causes unpredictable results when an UNLOCK is done.
3. LOCK cannot be called from a completion or interrupt routine.
4. If a SET USR NOSWAP command has been issued, LOCK and UNLOCK do not cause the USR to swap. However, in FB, LOCK still inhibits the other job from using the USR, and UNLOCK allows the other job access to the USR.
5. The USR cannot accept argument lists, such as device file name specifications, located in the area into which it has been locked.

Errors:

None.

Example:

```
INTEGER*2 DBLK(4)
DATA DBLK /3RDK ,3RDT,3RFIL,3RF4 /
.
.
.
CALL LOCK                    !LOCK THE USR IN MEMORY
ICHN=GETC()                  !GET A CHANNEL TO USE
IF(LOOKUP(ICHN,DBLK).LT.0) STOP '?LOOKUP FAILED'
CALL UNLOCK                  !RELEASE THE USR
.
.
.
```

3.79 LOOKUP

The LOOKUP function associates a specified channel with a device and/or file for the purpose of performing I/O operations. The channel used is then busy until one of the following functions is executed.

CLOSEC or ICLOSE
ISAVES
PURGE

Form: i = LOOKUP (chan,dblk[,count,seqnum,])

i = LOOKUP (chan,jobdes)

where:

chan — is the integer specification for the RT–11 channel to be associated with the file. You must obtain this channel through an IGETC call, or you can use channel 16(decimal) or higher if you have done an ICDFN call

dblk — is the four-word area specifying the Radix-50 file descriptor. Note that unpredictable results occur if the USR swaps over this four-word area

ccount — is an optional argument used for the cassette handler; this argument defaults to 0

seqnum — is a file number. For cassette operations, 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 null. The sequence number can have the following values:

–1 Suppress rewind and search for the specified file name from the current tape position. If a file name is given, a file-structured lookup is performed (do not rewind). If the file name is null, a non-file-structured lookup is done (tape is not moved). You must specify a –1 and no other negative number.

0 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*.

jobdes — is an argument that allows communication between jobs in a system job environment. It is a pointer to a four-word job descriptor of the job to which messages will be sent or received. The syntax is

```
jobdes → .RAD50 /MQ/
         .ASCII /logical-job-name/
```

where the *logical-job-name* is six characters long. If the *logical-job-name* is zero, the channel will be opened only for .READ/C/W requests, and such requests will accept messages from any jobs.

If the *jobdes* argument is omitted, .LOOKUP operates as it did for Version 3B.

NOTE

The arguments of LOOKUP must be positioned so that the USR does not swap over them.

The handler for the selected device must be in memory for a LOOKUP. If the first word of the file name in *dblk* is 0 and the device is a file-structured device, absolute block 0 of the device is designated as the beginning of the file. This technique, called a non-file-structured lookup, allows I/O to any physical block on the device. If a file name is specified for a device that is not file structured (such as LP:FILE.TYP), the name is ignored.

NOTE

Since a non-file-structured lookup allows I/O to any physical block on the device, the user must be aware that, in this mode, it is possible to overwrite the RT–11 device directory, thus destroying all file information on the device.

Function Results:

i = n	Indicates a successful file-structured lookup on a random-access storage volume.
i = 0	Indicates a successful non-file-structured lookup on both random-access and non-file-structured volumes, or a successful file-structured lookup on magtape.

Errors:

i = –1	Channel specified is already open.
= –2	File specified was not found on the device.
= –3	Device in use.
= –4	Tape drive is not available.
= –5	Illegal argument error with a file-structured volume.
= –6	Illegal argument error with a non-file-structured volume.

Example:

```
INTEGER*2 DBLK(4)
DATA DBLK/3RDK0,3RFTN,3R44  ,3RDAT/
.
.
.
ICHAN=IGETC()
IF(ICHAN.LT.0) STOP 'NO CHANNEL'
IF(IFETCH(DBLK).NE.0) STOP 'BAD FETCH'
IF(LOOKUP(ICHAN,DBLK).LT.0) STOP 'BAD LOOKUP'
.
.
.
CALL ICLOSE(ICHAN,I)
I = ICLOSE()
CALL IFREEC(ICHAN)
.
.
.
```

or using LOOKUP with a system job

```
      LOGICAL*1 JNAM(6)
      DIMENSION JBLK(4)
      EQUIVALENCE (JNAM(1),JBLK(2))
      DATA JNAM /'Q','U','E','U','E',0/
      DATA JBLK(1) /3RMQ /
      .
      .
      .
C     OPEN A MESSAGE CHANNEL TO 'QUEUE'
      ICHN=GETC()
      IF(LOOKUP(ICHN,JBLK),LT,0) STOP 'QUEUE IS NOT RUNNING'
      .
      .
      .
```

3.80 MRKT (SYSGEN Option in SJ)

The MRKT function schedules an assembly language completion routine to be entered after a specified time interval has elapsed. Support for MRKT in SJ requires timer support.

Form: i = MRKT (id,crtn,time)

where:

id is an integer identification number to be passed to the routine being scheduled

crtn is the name of the assembly language routine to be entered when the time interval elapses. This name must be specified in an EXTERNAL statement in the FORTRAN routine that issues the MRKT call

time is the two-word, internal format time interval; when this interval elapses, the routine is entered. If considered as a two-element INTEGER*2 array:

time(1) is the high-order time.
time(2) is the low-order time.

Notes:

1. MRKT requires a queue element, which should be considered when the IQSET function (Section 3.42) is executed.
2. If the system is busy, the time interval that elapses before the completion routine is run can be greater than that requested.

For further information on scheduling completion routines, see the .MRKT programmed request (Section 2.45).

Errors:

i = 0 Normal return.
= 1 No queue element was available; unable to schedule request.

Example:

```
INTEGER*2 TINT(2)
EXTERNAL ARTN
.
.
.
CALL MRKT(4,ARTN,TINT)
```

3.81 MTATCH (Special Feature)

The MTATCH subroutine attaches a terminal for exclusive use by the requesting job. This operation must be performed before any job can use a terminal with multiterminal programmed requests.

Form: i = MTATCH (unit[,addr][,jobnum])

where:

unit — is the logical unit number (*lun*) of the terminal

addr — is the optional address of an asynchronous terminal status word. Omit this argument if the asynchronous terminal status word is not required by specifying a comma. For example:

I = MTATCH (unit,,jobnum)

jobnum — is the job number associated with the terminal if the terminal is not available

Errors:

i = 0 Normal return.
= 3 Nonexistent unit number.
= 5 Unit attached by another job (job number returned in *jobnum*)
= 6 In XM monitor, the optional status word address is not in a valid user virtual address space

Example:

```
C     TEST SYSLIB MULTITERMINAL ROUTINES

      INTEGER*2 UNIT,SBLOK(4),STAT(8),ASW,STRING(41),PROMT(8)
      LOGICAL*1 TEND(11)
      REAL*4 TESTM(9)
      DATA PROMT/'EN','TE','R ','ST','RI','NG','>',"200/
      DATA TEND/'*','E','N','D',' ','T','E','S','T','*',0/
      DATA TESTM/'STAT','ATCH','GET','SET','','','','','DTCH'/

C     USE MTSTAT TO GET & DISPLAY NO. OF UNITS

      TYPE 106                                          ! ANNOUNCE TEST
      L=1                                               ! L = FUNC CODE
      IF(MTSTAT(STAT).NE.0)GOTO 999                     ! GET MTTY STATUS
5     TYPE 99,STAT(3)                                   ! DISPLAY # UNITS

C     GET UNIT # TO TEST
```

```
      TYPE 100                                              ! TYPE PROMPT
      ACCEPT 101,UNIT                                       ! GET UNIT #
      IF(UNIT.EQ.99) STOP 'END OF MULTITERMINAL TEST'! UNIT #99 STOPS TEST

C     ATTACH UNIT TO THIS JOB THEN GET TCB STATUS WORDS

      TYPE 110                                              ! SEE IF ASW TEST
      ACCEPT 111,IASW                                       ! IS TO BE DONE
      IF(IASW.EQ.'Y')IER=MTATCH(UNIT,ASW,JOB)               ! ATT W/ ASW IF YES
      IF(IASW.NE.'Y')IER=MTATCH(UNIT,0,JOB)                 ! ATT W/O ASW IF NO
      L=2
      IF(IER)GOTO 999                                       ! REPORT ERROR IF ANY
      L=3
      IF(MTGET(UNIT,SBLOK(1)),NE,0)GOTO 999                 ! GET TCB WORDS
      TYPE 102,UNIT,SBLOK                                   ! DISPLAY CONTENTS

C     GET NEW STATUS, PUT IT IN TCB, THEN DISPLAY IT

      CALL SETUP(SBLOK,UNIT)                                ! GET CHANGES IF ANY
      L=4
      IF(MTSET(UNIT,SBLOK(1)),NE,0)GOTO 999                 ! GET NEW TCB STATUS
      TYPE 102,UNIT,SBLOK                                   ! THEN DISPLAY IT...

C     PERFORM TEST - FIRST ECHO INPUT THEN REPEAT IT USING MTIN & MTOUT

20    TYPE 103                                              ! ANNOUNCE RULES OF
      TYPE 104                                              ! THE TEST...
      TYPE 105
30    CALL MTIN(UNIT,J)                                     ! GET LINE OF INPUT
      CALL MTOUT(UNIT,J)                                    ! REPEAT 1ST/ECHO 2ND
      IF(J.NE.10)GOTO 30                                    ! LF = END OF LINE
      CALL MTRCTO(UNIT)                                     ! RESET CTRL/O

C     NOW TEST W/ TTSPC$ BIT ON - ECHO INPUT WITH MTOUT (DON'T REPEAT)
C     THEN TURN TTSPC$ BIT OFF...

      IF(SBLOK(1).AND."10000)GOTO 40                        ! IF TTSPC$ ON BRANCH
      SBLOK(1)=SBLOK(1).OR."10000                           ! SET TTSPC$ BIT
      IF(MTSET(UNIT,SBLOK(1)),NE,0)GOTO 999                 ! UPDATE TCB
      GOTO 30                                               ! GO DO 2ND TEST
40    SBLOK(1)=SBLOK(1).AND..NOT."10000                     ! TURN OFF TTSPC$ BIT
      IF(MTSET(UNIT,SBLOK(1)),NE,0)GOTO 999                 ! UPDATE TCB
      IF(IASW.NE.'Y')GOTO 60                                ! SKIP ASW TEST?

C     ASYNCHRONOUS STATUS WORD TEST - "POLL" TERMINAL UNTIL INPUT
C     AVAILABLE - ECHO INPUT THEN REPEAT IT ON NEXT LINE

      TYPE 109                                              ! ANNOUNCE TEST
50    IF(.NOT.ASW.AND."40000)GOTO 50                        ! WAIT FOR INPUT
55    CALL MTIN(UNIT,J)                                     ! GET CHAR
      CALL MTOUT(UNIT,J)                                    ! OUTPUT CHAR
      IF(J.NE.10)GOTO 55                                    ! END ON LINE FEED
      CALL MTRCTO(UNIT)                                     ! RESET CTRL/O

C     TEST MTPRNT BY OUTPUTTING 2 STRINGS, 1 FROM USER & 1 INTERNAL

60    CALL GTLIN(STRING,PROMT)                              ! GET STRING VIA GTLIN
      CALL MTPRNT(UNIT,STRING)                              ! OUTPUT TO TERMINAL
      CALL MTPRNT(UNIT,TEND)                                ! ANNOUNCE END OF TEST

C     DETACH UNIT FROM JOB AND START OVER

      L=9
      TYPE 108,UNIT                                         ! DETATCH UNIT
      IF(MTDTCH(UNIT).EQ.0)GOTO 5                           ! FROM JOB THEN LOOP
```

```
C     ERROR REPORTING

999   TYPE 909,TESTM(L),IER                                  ! ANNOUNCE ERROR
      GOTO 5                                                 ! THEN START OVER

99    FORMAT('0THERE ARE',I3,' UNITS ON THIS SYSTEM')
100   FORMAT('$UNIT # TO BE TESTED?')
101   FORMAT(I2)
102   FORMAT('0UNIT',I3,' STATUS =',4O8)
103   FORMAT('0GO TO TERMINAL BEING TESTED...ENTER 2 LINES + (RET)')
104   FORMAT(' 1ST LINE: INPUT WILL BE ECHOED THEN REPEATED')
105   FORMAT(' 2ND LINE: TEST TTSPC$ ON - INPUT ECHOED VIA MTOUT'/)
106   FORMAT('1     SYSLIB MULTITERMINAL ROUTINE TEST PROGRAM')
108   FORMAT(' ABOUT TO DETACH UNIT # ',I2)
109   FORMAT(' TEST ASW - INPUT WILL BE ECHOED, THEN REPEATED'/)
110   FORMAT('$TEST ASYNCH STATUS WORD FUNCTION?')
111   FORMAT(A1)
909   FORMAT('0MT',A4,'  ERROR CODE =',I3)
      END

C     SUBROUTINE TO GET NEW STATUS WORD VALUES

      SUBROUTINE SETUP(SBLOK,UNIT)

      INTEGER SBLOK(4),UNIT

      TYPE 100                                               ! PROMPT FOR NEW CONFIG WORD
      ACCEPT 101,J                                           ! ACCEPT INPUT
      IF(J)SBLOK(1)=J                                        ! UPDATE IF ANY INPUT
      TYPE 102                                               ! ASK FOR FILL CHAR
      ACCEPT 101,J                                           ! ACCEPT IT
      TYPE 103                                               ! ASK FOR # OF FILL CHARS
      ACCEPT 101,I                                           ! ACCEPT IT TOO
      IF(I.OR.J)SBLOK(3)=I*256+J                             ! PUT IN PROPER BYTES
5     TYPE 104                                               ! ASK FOR CARRIAGE WIDTH
      ACCEPT 105,I                                           ! ACCEPT IT
      IF(I)SBLOK(4)=SBLOK(4)/256*256+I                       ! SET BUT DON'T MESS WITH
      RETURN                                                 ! STATE WORD ... RETURN

100   FORMAT('$CONFIG BIT MASK:')
101   FORMAT(O6)
102   FORMAT('$CHAR REQUIRING FILLER:')
103   FORMAT('$# OF FILL CHARS:')
104   FORMAT('$CARRIAGE WIDTH:')
105   FORMAT(I3)
      END
```

3.82 MTDTCH (Special Feature)

The MTDTCH subroutine is the complement of the MTATCH subroutine. Its function is to detach a terminal from a particular job and make it available for other jobs.

Form: i = MTDTCH(unit)

where:

unit is the logical unit number (*lun*) of the terminal to be detached

Errors:

i = 0 Normal return.
= 2 Invalid unit number; terminal is not attached.
= 3 Nonexistent unit number.

Example:

Refer to the example under MTATCH.

3.83 MTGET (Special Feature)

The MTGET subroutine furnishes the user with information about a specific terminal in a multiterminal system. You do not need to do an MTATCH before using MTGET.

Form: i = MTGET (unit,addr[,jobnum])

where:

unit	is the unit number of the line and terminal whose status is desired
addr	is the four-word area to receive the status information. The area is a four-element INTEGER*2 array (see the .MTSET programmed request, Section 2.54, for area format)
jobnum	is the job number associated with the terminal if the terminal is not available

Status information including bit definitions for the terminal configuration words and the terminal state byte are described in detail under the .MTGET programmed request.

Errors:

i = 0	Normal return.
= 2	Unit not attached.
= 3	Nonexistent unit number.
= 4	Unit attached by another job (job number returned in *jobnum*).
= 6	In XM monitor, the address of the terminal buffer is outside the valid program limits.

Example:

Refer to the example under MTATCH.

3.84 MTIN (Special Feature)

The MTIN subroutine transfers characters from a specified terminal to the user program. This subroutine is a multiterminal form of ITTINR. If no characters are available, an error flag is set to indicate an error upon return from the subroutine. If no character count argument is specified, one character is transferred.

Form: i = MTIN (unit,char[,chrcnt][,ocnt])

where:

unit	is the unit number of the terminal
char	is the variable to contain the characters read in from the terminal indicated by the unit number

chrcnt is an optional argument that indicates the number of characters to be read

ocnt is an optional argument that indicates the number of characters actually transferred

When a request for a multiple-character transfer is requested, if the optional fourth argument (*ocnt*) is specified and bit 6 of the M.TSTS word is set, the variable specified as the argument will have a value equal to the actual number of characters transferred upon return from the subroutine.

Errors:

i = 0 Normal return.
= 1 No input available.
= 2 Unit not attached.
= 3 Nonexistent unit number.

Example:

Refer to the example under MTATCH.

3.85 MTOUT (Special Feature)

The MTOUT subroutine transfers characters to a specified terminal. This subroutine is a multiterminal form of ITTOUR. If no room is available in the output ring buffer, an error flag is set to indicate an error upon return from the subroutine. If no character count argument is specified, one character is transferred.

Form: i = MTOUT (unit,char[,chrcnt][,ocnt])

where:

unit is the unit number of the terminal

char is the variable or array containing the characters to be output, right-justified in the integer (can be LOGICAL*1 if desired)

chrcnt is an optional argument that indicates the number of characters to be output

ocnt is an optional argument that indicates the number of characters actually transferred

When a request for a multiple-character transfer is requested, if the optional fourth argument (*ocnt*) is specified and bit 6 of the M.TSTS word is set, the variable specified as the argument will have a value equal to the actual number of characters transferred upon return from the subroutine.

Errors:

i = 0 Normal return.
= 1 No room in output ring buffer.
= 2 Unit not attached.
= 3 Nonexistent unit number.
= 5 In the XM monitor, the address of the user buffer is outside the valid program limits.

Example:

Refer to the example under MTATCH.

3.86 MTPRNT (Special Feature)

The MTPRNT subroutine allows output to be printed at any terminal in a multiterminal environment. This subroutine has the same effect as the PRINT subroutine (Section 3.91).

Form: i = MTPRNT (unit,string)

where:

unit — is the unnit number associated with the terminal

string — is the character string to be printed. Note that all quoted literals used in FORTRAN subroutine calls are in ASCIZ format, which ends in zero for a CR/LF or a 200 if no action is to be taken

Errors:

i = 0 Normal return.
= 2 Unit not attached.
= 3 Nonexistent unit number.
= 5 In the XM monitor, the address of the character string is outside the valid program limits.

3.87 MTRCTO (Special Feature)

The MTRCTO subroutine resets the CTRL/O command typed at the specified terminal in a multiterminal environment. This subroutine has the same effect as the .MTRCTO programmed request (Section 2.53).

Form: i = MTRCTO(unit)

where:

unit — is the unit number associated with the terminal

Errors:

i = 0 Normal return.
= 2 Unit not attached.
= 3 Nonexistent unit number.

Example:

Refer to the example under MTATCH.

3.88 MTSET (Special Feature)

The MTSET subroutine sets terminal and line characteristics. The set conditions remain in effect until the system is booted or the terminal and line characteristics are reset. See the .MTSET programmed request (Section 2.54) for more details.

Form: i = MTSET (unit,addr)

where:

unit — is the unit number of the line and terminal whose characteristics are to be changed

addr — is a four-word area to pass the status information. The area is a four-element INTEGER*2 array

Errors:

i = 0 — Normal return.
= 2 — Unit not attached.
= 3 — Nonexistent unit number.
= 6 — In the XM monitor, the address of the status block is outside the valid program limits.

Example:

Refer to the example under MTATCH.

3.89 MTSTAT (Special Feature)

The MTSTAT subroutine returns multiterminal system status in an eight-word status block.

Form: i = MTSTAT (addr)

where:

addr — is the address of an eight-word array where multiterminal status information is returned. The status block contains the following information:

	Contents
addr(1)	Offset from the base of the resident monitor to the first Terminal Control Block (TCB).
addr(2)	Offset from the base of the resident monitor to the terminal control block of the console terminal for the program.
addr(3)	The value (0–16 decimal) of the highest logical unit number (LUN) built into the system.
addr(4)	The size of the terminal control block in bytes.
addr(5)–(8)	Reserved.

Errors:

i = 0 — Normal return.
= 5 — In the XM monitor, the address of the status block is not in valid user address space.

Example:

Refer to the example under MTATCH.

3.90 MWAIT (FB and XM Only)

The MWAIT subroutine suspends main program execution of the current job until all messages sent to or from the other job have been transmitted or received. It provides a means for ensuring that a required message has been processed. MWAIT is used primarily in conjunction with the IRCVD and ISDAT calls, where no action is taken when a message transmission is completed. This subroutine requires a queue element, which should be considered when the IQSET function (Section 3.42) is executed.

Form: CALL MWAIT

Errors:

None.

Example:

Refer to the example under ISDAT, Section 3.51.

3.91 PRINT

The PRINT subroutine prints output from a specified string at the console terminal. This routine can be used to print messages from completion routines without using the FORTRAN formatted I/O system. Control returns to the user program after all characters have been placed in the output buffer.

The string to be printed can be terminated with either a null (0) byte or a 200(octal) byte. If the null (ASCIZ) format is used, the output is automatically followed by a carriage return/line feed pair (octal 15 and 12). If a 200 byte terminates the string, no carriage return/line feed pair is generated.

In the FB monitor, a change in the job that is controlling terminal output is indicated by a B> or F>. Any text following the message has been printed by the job indicated (foreground or background) until another B> or F> is printed. When PRINT is used by the foreground job, the message appears immediately, regardless of the state of the background job. Thus, for urgent messages, PRINT should be used rather than ITTOUR.

Form: CALL PRINT (string)

where:

string is the string to be printed. Note that all quoted literals used in FORTRAN subroutine calls are in ASCIZ format, as are all strings produced by the SYSLIB string-handling package (The CONCAT routine can be used to append an octal 200 to an ASCIZ string; see example.)

Errors:

None.

Example:

```
CALL PRINT ('THE COFFEE IS READY')

or

BYTE QUESTION(80)
!APPEND BYTE 200
CALL CONCAT('WHAT IS YOUR NAME?,"200,QUESTION)
CALL PRINT(QUESTION) !QUESTION PRINTS WITHOUT CR,LF
```

3.92 PURGE

The PURGE subroutine deactivates a channel without performing an ISAVES, CLOSEC, or ICLOSE. Any tentative file currently associated with the channel is not made permanent. This subroutine prevents entered (IENTER or .ENTER) files from becoming permanent directory entries.

Form: CALL PURGE (chan)

where:

chan is the integer specification for the RT–11 channel to be deactivated

Errors:

None.

Example:

Refer to the example under IENTER, Section 3.25.

3.93 PUTSTR

The PUTSTR subroutine writes a variable-length character string to a specified FORTRAN logical unit. PUTSTR can be used in main program routines or in completion routines but not in both in the same program at the same time. If PUTSTR is used in a completion routine, it must not be the first I/O operation on the specified logical unit.

Form: CALL PUTSTR (lun,in,char,err)

where:

lun is the integer specification of the FORTRAN logical unit number to which the string is to be written

in is the array containing the string to be written

char is an ASCII character that is appended to the beginning of the string before it is output. If 0, no extra character is output. This character is used primarily for carriage control purposes

err is a LOGICAL*1 variable that is .TRUE. for an error condition and .FALSE. for a no-error condition

Errors:

err =	–1	End-of-file for write operation.
	–2	Hardware error for write operation.

Example:

```
LOGICAL*1 STRNG(81),ERR
.
.
.
!OUTPUT STRING WITH DOUBLE SPACING
CALL PUTSTR(7,STRNG,'0',ERR)
```

3.94 R50ASC

The R50ASC subroutine converts a specified number of Radix–50 characters to ASCII.

Form: CALL R50ASC (icnt,input,output)

where:

icnt	is the integer number of ASCII characters to be produced
input	is the area from which words of Radix–50 values to be converted are taken. Note that (*icnt*+2)/3 words are read for conversion
output	is the area into which the ASCII characters are stored

Errors:

If an input word contains illegal Radix–50 codes — that is, if the input word is greater (unsigned) than 174777(octal) — the routine outputs question marks for the value.

Example:

```
REAL*8 NAME
LOGICAL*1 OUTP(12)
.
.
.
CALL R50ASC(12,NAME,OUTP)
```

3.95 RAD50

The RAD50 function provides a method of encoding RT–11 file descriptors in Radix–50 notation. The RAD50 function converts six ASCII characters from the specified area, returning a REAL*4 result that is the two-word Radix–50 value.

Form: a = RAD50 (input)

where:

input	is the area from which the ASCII input characters are taken

The RAD50 call:

```
A = RAD50 (LINE)
```

is exactly equivalent to the IRAD50 call:

```
CALL IRAD50 (6,LINE,A)
```

Function Results:

The two-word Radix–50 value is returned as the function result.

3.96 RCHAIN

The RCHAIN subroutine allows a program to determine whether it has been chained to and to access variables passed across a chain. If RCHAIN is used, it must be used in the first executable FORTRAN statement in a program.

Form: CALL RCHAIN (flag,var,wcnt)

where:

flag — is an integer variable that RCHAIN will set to –1 (true) if the program has been chained to; otherwise, it is 0 (false)

var — is the first variable in a sequence of variables with increasing memory addresses to receive the information passed across the chain (see Section 3.2)

wcnt — is the number of words to be moved from the chain parameter area to the area specified by *var*. RCHAIN moves *wcnt* words into the area beginning at *var*

Errors:

None.

Example:

```
INTEGER*2 PARMS(50)
CALL RCHAIN(IFLAG,PARMS,50)
IF(IFLAG) GOTO 10     !GOTO 10 IF CHAINED TO
.
.
.
```

3.97 RCTRLO

The RCTRLO subroutine resets the effect of any console terminal CTRL/O command that was typed. After an RCTRLO call, any output directed to the console terminal prints until another CTRL/O is typed.

Form: CALL RCTRLO

Errors:

None.

Example:

```
CALL RCTRLO
CALL PRINT ('PRINT UNTIL ANOTHER CTRL/O TYPED')
```

3.98 REPEAT

The REPEAT subroutine concatenates a specified string with itself to produce the indicated number of copies. REPEAT places the resulting string in a specified array.

Form: CALL REPEAT (in,out,i[,len[,err]])

where:

in — is the array containing the string to be repeated; it must be terminated with a null byte

out — is the array into which the resultant string is placed. This array must be at least one element longer than the value of *len*, if *len* is specified. It also must be terminated with a null byte if *len* is specified

i — is the integer number of times to repeat the string

len — is the integer number representing the maximum length of the output string

err — is the logical error flag set if the output string is truncated to the length specified by *len*

Input and output strings can specify the same array only if the repeat count (*i*) is 1 or 0. When the repeat count is 1, this routine is the equivalent of SCOPY; when the repeat count is 0, *out* is replaced by a null string. The old contents of *out* are lost when this routine is called.

Errors:

Error conditions are indicated by *err*, if specified. If *err* is given and the output string would have been longer than *len* characters, then *err* is set to .TRUE.; otherwise, *err* is unchanged.

Example:

```
LOGICAL*1 SIN(21),SOUT(101)
.
.
.
CALL REPEAT(SIN,SOUT,5)
```

3.99 RESUME (FB and XM Only)

The RESUME subroutine allows a job to resume execution of the main program. A RESUME call is normally issued from an asynchronous FORTRAN routine entered on I/O completion or because of a schedule request (see the SUSPND subroutine, Section 3.107, for more information).

Form: CALL RESUME

Errors:

None.

Example:

Refer to the example under SUSPND.

3.100 SCCA

The SCCA subroutine provides a CTRL/C intercept to:

1. Inhibit a CTRL/C abort
2. Indicate that a CTRL/C command is active
3. Distinguish between single and double CTRL/C commands

Form: CALL SCCA [(iflag)]

where:

iflag is an integer terminal status word that must be tested and cleared to determine if two CTRL/Cs were typed at the console terminal; the *iflag* must be an INTEGER*2 variable (not LOGICAL*1)

When a CTRL/C is typed, the SCCA subroutine places it in the input ring buffer. While residing in the buffer, the character can be read by the program. The program must test and clear the *iflag* to determine if two CTRL/C commands were typed consecutively. The *iflag* is set to non-zero when two CTRL/Cs are typed together. It is the responsibility of the program to abort itself, if appropriate, on an input of CTRL/C from the terminal. The SCCA subroutine with no argument disables the CTRL/C intercept. A CTRL/C from indirect command files is not intercepted by SCCA.

Errors:

None.

Example:

```
      PROGRAM SCCA
C     SCCA.FOR SYSLIB TEST FOR SCCA
C
      CALL PRINT ('PROGRAM HAS STARTED, TYPE')
      IFLAG=0
      CALL SCCA (IFLAG)
10    I = ITTINR()              !GET A CHARACTER
      IF (I .NE. 3) GOTO 10
C     A CTRL/C WAS TYPED
      CALL PRINT ('A CTRL/C WAS TYPED')
      IF (IFLAG .EQ. 0) GOTO 10
      CALL PRINT ('A DOUBLE CTRL/C WAS TYPED')
      TYPE 19,IFLAG
19    FORMAT (' IFLAG = ',O6,/)
      CALL SCCA                 !DISABLE CTRL/C INTERCEPT
      CALL PRINT ('TYPE A CTRL/C TO EXIT')
20    GOTO 20                   !LOOP UNTIL CTRL/C TYPED
      END
```

3.101 SCOMP/ISCOMP

The SCOMP routine compares two character strings and returns the integer result of the comparison.

Form: CALL SCOMP (a,b,i)

or

i = ISCOMP (a,b)

where:

a is the array containing the first string; it must be terminated with a null byte

b is the array containing the second string; it must be terminated with a null byte

i is the integer variable that receives the result of the comparison

The strings are compared from left to right, one character at a time, using the collating sequence specified by the ASCII codes for each character. If the two strings are not equal, the absolute value of variable *i* (or the result of the function ISCOMP) is the character position of the first inequality found. Strings are terminated by a null (0) character.

If the strings are not the same length, the shorter one is treated as if it were padded on the right with blanks to the length of the other string. A null string argument is equivalent to a string containing only blanks.

Function Results:

i <0 If *a* is less than *b*.
=0 If *a* is equal to *b*.
>0 If *a* is greater than *b*.

Example:

```
LOGICAL*1 INSTR(81)
.
.
.
CALL GETSTR(5,INSTR,80)
CALL SCOMP('YES',INSTR,IVAL)
IF(IVAL.NE.0) GOTO 10   !IF INPUT STRING IS NOT YES GOTO 10
```

3.102 SCOPY

The SCOPY routine copies a character string from one array to another. Copying stops either when a null (0) character is encountered or when a specified number of characters have been moved.

Form: CALL SCOPY (in,out[,len[,err]])

where:

in is the array containing the string to be copied; it must be terminated with a null byte if *len* is not specified, or if the string is shorter than *len*

out is the array to receive the copied string. This array must be at least one element longer than the value of *len*, if *len* is specified. It also must be terminated with a null byte if *len* is specified

len is the integer number representing the maximum length of the output string. The effect of *len* is to truncate the output string to a given length

err is a logical variable that receives the error indication if the output string was truncated to the length specified by *len*

The input (*in*) and output (*out*) arguments can specify the same array. The string previously contained in the output array is lost when this subroutine is called.

Errors:

Error conditions are indicated by *err*, if specified. If *err* is given and the output string was truncated to the length specified by *len*, then *err* is set to .TRUE.; otherwise, *err* is unchanged.

Example:

SCOPY is useful for initializing strings to a constant value, for example:

```
LOGICAL*1 STRING(80)
CALL SCOPY('THIS IS THE INITIAL VALUE',STRING)
```

3.103 SECNDS

The SECNDS function returns the current system time, in seconds past midnight, minus the value of a specified argument. Thus, SECNDS can be used to calculate elapsed time. The value returned is single-precision floating point (REAL*4).

Form: a = SECNDS (atime)

where:

atime is a REAL*4 variable, constant, or expression whose value is subtracted from the current time of day to form the result

Notes:

This function does floating-point arithmetic. Elapsed time can also be calculated by using the GTIM call and the INTEGER*4 support functions.

Function Result:

The function result (*a*) is the REAL*4 value returned.

Errors:

None.

Example:

```
C     START OF TIMED SEQUENCE
      T1=SECNDS(0.)
C
C     CODE TO BE TIMED GOES HERE
C
      DELTA=SECNDS(T1)     !DELTA IS ELAPSED TIME
```

3.104 SETCMD

The SETCMD routine allows a user program to pass a command line to the keyboard monitor to be executed after the program exits. The command lines are passed to the chain information area (500–777 octal) and stored beginning at location 512(octal). No check is made to determine if the string extends into the stack space. For this reason, the command line should be short and the subroutine call should be made in the main program unit near the end of the program just before completion. When several commands are involved, an indirect command file that contains several command lines should be used.

The monitor commands REENTER, START, and CLOSE are not allowed if the SETCMD feature is used.

Form: CALL SETCMD (string)

where:

string is a keyboard monitor command line in ASCIZ format with no embedded carriage returns or line feeds

Errors:

None.

Example:

```
LOGICAL*1,INPUT(134),PROMPT(8)
DATA PROMPT/'P','R','O','M','P','T','>',"200/
CALL GTLIN (INPUT,PROMPT)
CALL SETCMD (INPUT)
END
```

NOTE

Set USR NOSWAP, or specify /NOSWAP with the COMPILE, FORTRAN, or EXECUTE command to control the swapping state of the USR. A LOCK would inhibit another job from using the USR.

A STOP or CALL EXIT must also be issued after the SETCMD to cause an exit.

3.105 STRPAD

The STRPAD routine pads a character string with rightmost blanks until that string is a specified length. This padding is done in place; the result

string is contained in its original array. If the present length of the string is greater than or equal to the specified length, no padding occurs.

Form: CALL STRPAD (a,len[,err])

where:

a is the array containing the string to be padded. This array must be one element longer than the value of *len* if *len* is specified. It will be terminated by a null byte

len is the integer length of the desired result string

err is the logical error flag that is set to .TRUE. if the string specified by *a* exceeds the value of *i* in length

Errors:

Error conditions are indicated by *err*, if specified. If *err* is given and the string indicated is longer than *i* characters, *err* is set to .TRUE.; otherwise, the value of *err* is unchanged.

Example:

This routine is especially useful for preparing strings to be output in A-type FORMAT fields. For example:

```
      LOGICAL*1 STR(81)
      .
      .
      .
      CALL STRPAD(STR,80)        !ASSURE 80 VALID CHARACTERS
      PRINT 100,(STR(I),I=1,80) !PRINT STRING OF 80 CHARACTERS
100   FORMAT(80A1)
```

3.106 SUBSTR

The SUBSTR routine copies a substring from a specified position in a character string. If desired, the substring can then be placed in the same array as the string from which it was taken.

Form: CALL SUBSTR (in,out,i[,len])

where:

in is the array from which the substring is taken; it is terminated by a null byte

out is the array to contain the substring result. This array must be one element longer than *len*, if *len* is specified. It also is terminated by a null byte if *len* is specified

i is the integer character position in the input string of the first character of the desired substring

len is the integer number of characters representing the maximum length of the substring

If a maximum length (*len*) is not given, the substring contains all characters to the right of character position *i* in array *in* and is not terminated by a null byte. If *len* is given, the string is copied and terminated with a null byte. If *len* is equal to zero, *out* is replaced by the null string. The old contents of array *out* are lost when this routine is called.

Errors:

None.

3.107 SUSPND (FB and XM Only)

The SUSPND subroutine suspends main program execution of the current job and allows only completion routines (for I/O and scheduling requests) to run.

Form: CALL SUSPND

Notes:

1. The monitor maintains a suspension counter for each job. This count is decremented by SUSPND and incremented by RESUME (see Section 3.99). A job will actually be suspended only if this counter is negative. Thus, if a RESUME is issued before a SUSPND, the latter routine will return immediately.
2. A program must issue an equal number of SUSPND and RESUME calls.
3. A SUSPND subroutine call from a completion routine decrements the suspension counter but does not suspend the main program. If a completion routine does a SUSPND, the main program continues until it also issues a SUSPND, at which time it is suspended. Two RESUME calls are then required to proceed.
4. Because SUSPND and RESUME are used to simulate an ITWAIT (see Section 3.61) in the monitor, a RESUME issued from a completion routine and not matched by a previously executed SUSPND can cause the main program execution to continue past a timed wait before the entire time interval has elapsed.

For further information on suspending main program execution of the current job, see the .SPND programmed request (Section 2.84).

Errors:

None.

Example:

```
INTEGER IAREA(4)
COMMON /RDBLK/ IBUF(256)
EXTERNAL RDFIN
.
.
.
```

```
      IF(IREADF(256,IBUF,IBLK,ICHAN,IAREA,RDFIN),NE,0) GOTO 1000
C     GOTO 1000 FOR ANY TYPE OF ERROR
C
C     DO OVERLAPPED PROCESSING
      .
      .
      .
      CALL SUSPND      !SYNCHRONIZE WITH COMPLETION ROUTINE
      .
      .
      .
      END
      SUBROUTINE RDFIN(IARG1,IARG2)
      COMMON /RDBLK/ IBUF(256)
      .
      .
      .
      CALL RESUME      !CONTINUE MAIN PROGRAM
      .
      .
      .
      END
```

3.108 TIMASC

The TIMASC subroutine converts a two-word internal format time into an ASCII string of the form:

hh:mm:ss

where:

hh is the two-digit hours indication

mm is the two-digit minutes indication

ss is the two-digit seconds indication

Form: CALL TIMASC (itime,strng)

where:

itime is the two-word internal format time to be converted. itime(1) is the high-order time, itime(2) is the low-order time

strng is the eight-element array to contain the ASCII time

Errors:

None.

Example:

The following example determines the amount of time from the time the program is run until 5 p.m. and prints it.

```
      INTEGER*4 J1,J2,J3
      LOGICAL*1 STRNG(8)
      .
      .
      .
```

```
      CALL JTIME(17,0,0,0,J1)
      CALL GTIM(J2)
      CALL JJCVT(J1)
      CALL JJCVT(J2)
      CALL JSUB(J1,J2,J3)
      CALL JJCVT(J3)
      CALL TIMASC(J3,STRNG)
      TYPE 99,(STRNG(I),I=1,8)
99    FORMAT(' IT IS ',8A1,' TILL 5 P.M.')
      .
      .
      .
```

3.109 TIME

The TIME subroutine returns the current system time of day as an eight-character ASCII string of the form:

hh:mm:ss

where:

hh is the two-digit hours indication

mm is the two-digit minutes indication

ss is the two-digit seconds indication

Form: CALL TIME (strng)

where:

strng is the eight-element array to receive the ASCII time

Notes:

A 24-hour clock is used (for example, 1:00 p.m. is represented as 13:00:00).

Errors:

None.

Example:

```
      LOGICAL*1 STRNG(8)
      .
      .
      .
      CALL TIME(STRNG)
      TYPE 99,(STRNG(I),I=1,8)
99    FORMAT (' IT IS NOW ',8A1)
```

3.110 TRANSL

The TRANSL routine performs character translation on a specified string and requires approximately 64(decimal) words on the R6 stack for its execution. This space should be considered when allocating stack space.

Form: CALL TRANSL (in,out,r[,p])

where:

in — is the array containing the input string; it is terminated by a null byte

out — is the array to receive the translated string; it is not terminated by a null byte

r — is the array containing the replacement string; it is terminated by a null byte

p — is the array containing the characters in *in* to be translated; it is terminated by a null byte

The string specified by array *out* is replaced by the string specified by array *in*, modified by the character translation process specified by arrays *r* and *p*. If any character position in *in* contains a character that appears in the string specified by *p*, it is replaced in *out* by the corresponding character from string *r*. If the array *p* is omitted, it is assumed to be the 127 seven-bit ASCII characters arranged in ascending order, beginning with the character whose ASCII code is 001. If strings *r* and *p* are given and differ in length, the longer string is truncated to the length of the shorter. If a character appears more than once in string *p*, only the last occurrence is significant. A character can appear any number of times in string *r*.

Errors:

None.

Examples:

The following example causes the string in array A to be copied to array B. All periods within A become minus signs, and all question marks become exclamation points.

```
CALL TRANSL(A,B,'-!',',?')
```

The following is an example of TRANSL being used to format character data.

```
      LOGICAL*1 STRING(27),RESULT(27),PATRN(27)
C     SET UP THE STRING TO BE REFORMATTED
C
      CALL SCOPY('THE HORN BLOWS AT MIDNIGHT',STRING)
C     SET UP NUMBER-CHARACTER DATA RELATIONSHIP
C
C     000000000111111111122222222
C     123456789012345678901234567
C     THE HORN BLOWS AT MIDNIGHT
C     NOW SET UP PATRN TO CONTAIN THE FOLLOWING PATTERN:
C     16,17,18,19,20,21,22,23,24,25,26,15,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0
C
      DO 10 I=16,26
10    PATRN(I-15)=I
      PATRN(12)=15
      DO 20 I=1,14
20    PATRN(I+12)=I
      PATRN(27)=0
```

```
C
C   THE FOLLOWING CALL TO TRANSL REARRANGES THE CHARACTERS OF
C   THE INPUT STRING TO THE ORDER SPECIFIED BY PATRN:
C
    CALL TRANSL(PATRN,RESULT,STRING)
C
C   RESULT NOW CONTAINS THE STRING 'AT MIDNIGHT THE HORN BLOWS'
C   IN GENERAL, THIS METHOD CAN BE USED TO FORMAT INPUT STRINGS
C   OF UP TO 127 CHARACTERS. THE RESULTANT STRING WILL BE
C   AS LONG AS THE PATTERN STRING (AS IN THE ABOVE EXAMPLE).
```

3.111 TRIM

The TRIM routine shortens a specified character string by removing all trailing blanks. A trailing blank is a blank that has no non-blanks to its right. If the specified string contains all blank characters, it is replaced by the null string. If the specified string has no trailing blanks, it is unchanged.

Form: CALL TRIM (a)

where:

a is the array containing the string to be trimmed; it is terminated by a null byte on input and output

Errors:

None.

Example:

```
      LOGICAL*1 STRING(81)
      ACCEPT 100,(STRING(I),I=1,80)
100   FORMAT(80A1)
      CALL SCOPY(STRING,STRING,80)        !MAKE ASCIZ
      CALL TRIM(STRING)                   !TRIM TRAILING BLANKS
```

3.112 UNLOCK

The UNLOCK subroutine releases the User Service Routine (USR) from memory if it was placed there by the LOCK routine. If the LOCK required a swap, the UNLOCK loads the user program back into memory. If the USR does not require swapping, the UNLOCK involves no I/O. The USR is always resident in XM.

Form: CALL UNLOCK

Notes:

1. It is important that at least as many UNLOCK calls are given as LOCK calls. If more LOCK calls were done, the USR remains locked in memory. Extra UNLOCK calls are ignored.
2. When running two jobs in the FB system, use the LOCK/UNLOCK pairs only when absolutely necessary. If one job locks the USR, the other job cannot use the USR until it is unlocked.

3. In an FB system, calling the CSI (ICSI) with input coming from the console terminal performs a temporary implicit UNLOCK.

For further information on releasing the USR from memory, see the .LOCK/.UNLOCK programmed requests (Section 2.41).

Errors:

None.

Example:

```
      .
      .
      .
C     GET READY TO DO MANY USR OPERATIONS
      CALL LOCK      !DISABLE USR SWAPPING
C     PERFORM THE USR CALLS
      .
      .
      .
C     FREE THE USR
      CALL UNLOCK
      .
      .
      .
```

3.113 VERIFY

The VERIFY routine checks that a given string is composed entirely of characters from a second string. If a character does not exist in the string being examined, VERIFY returns the position of the first character in the string being examined that is not in the source string. If all characters exist, VERIFY returns a 0.

Form: CALL VERIFY (a,b,i)

or

i = IVERIF (a,b)

where:

a is the array containing the string to be scanned; it is terminated by a null byte

b is the array containing the string of characters to be accepted in *a*; it is terminated by a null byte

Function Result:

i = 0 If all characters of *a* exist in *b*; also if *a* is a null string.
= n Where *n* is the character position of the first character in array *a* that does not appear in array *b*; if *b* is a null string and *a* is not, *i* equals 1.

Example:

The following example accepts a one- to five-digit unsigned decimal number and returns its value.

```
LOGICAL*1 INSTR(81)
.
.
.
CALL VERIFY(INSTR,'0123456789',I)
IF(I.EQ.1) STOP 'NUMBER MISSING'
IF(I.EQ.0) I=LEN(INSTR)
IF(I.GT.5) STOP 'TOO MANY DIGITS'
NUM=IVALUE(INSTR,I)
.
.
.
END
```

Appendix A
Display File Handler

This appendix describes the assembly language support provided under RT-11 for the VT11 graphic display hardware systems.

The following manuals are suggested for additional reference:

GT40/GT42 User's Guide
EK-GT40-OP-002

GT44 User's Guide
EK-GT44-OP-001

VT11 Graphic Display Processor
EK-VT11-TM-001

DECGRAPHIC-11 GT Series Reference Card
EH-02784-73

DECGraphic-11 FORTRAN Reference Manual
DEC-11-GFRMA-A-D

BASIC-11 Graphics Extensions User's Guide
DEC-11-LBGEA-A-D

A.1 Description

The graphics display terminals have hardware configurations that include a display processor and CRT (cathode ray tube) display. All systems are equipped with light pens and hardware character and vector generators, and are capable of high-quality graphics. The Display File Handler supports this graphics hardware at the assembly language level under the RT-11 monitor.

A.1.1 Assembly Language Display Support

The Display File Handler is not an RT-11 device handler, since it does not use the I/O structure of the RT-11 monitor. For example, it is not possible to use a utility program to transfer a text file to the display through the Display File Handler. Rather, the Display File Handler provides the graphics programmer the means for the display of graphics files and the easy management of the display processor. Included in its capabilities are such services as interrupt handling, light pen support, tracking object, and starting and stopping of the display processor.

The Display File Handler manages the display processor by means of a base segment (called VTBASE) which contains interrupt handlers, an internal display file and some pointers and flags. The display processor cycles through the internal display file; any user graphics files to be displayed are accessed

by display subroutine calls from the Handler's display file. In this way, the Display File Handler exerts control over the display processor, relieving the assembly language user of the task.

Through the Display File Handler, the programmer can insert and remove calls to display files from the Handler's internal display file. Up to two user files may be inserted at one time, and that number may be increased by re-assembling the Handler. Any user file inserted for display may be blanked (the subroutine call to it bypassed) and unblanked by macro calls to the Display File Handler.

Since the Handler treats all user display files as graphics subroutines to its internal display file, a display processor subroutine call is required. This is implemented with software, using the display stop instruction, and is available for user programs. This instruction and several other extended instructions implemented with the display stop instruction are described in Section A.3.

The facilities of the Display File Handler are accessed through a file of macro definitions (VTMAC) which generate calls to a set of subroutines in VTLIB. VTMAC's call protocol is similar to that of the RT–11 macros. The expansion of the macros is shown in Section A.6. VTMAC also contains, for convenience in programming, the set of recommended display processor instruction mnemonics and their values. The mnemonics are listed in Section A.7 and are used in the examples throughout this appendix.

VTCAL1 through VTCAL4 are the set of subroutines which service the VTMAC calls. They include functions for display file and display processor management. These are described in detail in Section A.2. VTCAL1 through VTCAL4 are distributed, along with the base segment VTBASE, as a file of five object modules called VTHDLR.OBJ. VTHDLR is built into the graphics library VTLIB by using the monitor LIBRARY command. VTHDLR only supports VT11 hardware. Section A.4.2 shows an example.

A.1.2 Monitor Display Support

The RT–11 monitor, under Version 03 and later, directly supports the display as a console device. A keyboard monitor command, GT ON (GT OFF) permits the selection of the display as console device. Selection results in the allocation of approximately 1.25K words of memory for text buffer and code. The buffer holds approximately 2000 characters.

The text display includes a blinking cursor to indicate the position in the text where a character is added. The cursor initially appears at the top left corner of the text area. As lines are added to the text the cursor moves down the screen. When the maximum number of lines are on the screen, the top line is deleted from the text buffer when the line feed terminating a new line is received. This causes the appearance of "scrolling," as the text disappears off the top of the display.

When the maximum number of characters have been inserted in the text buffer, the scroller logic deletes a line from the top of the screen to make room

for additional characters. Text may appear to move (scroll) off the top of the screen while the cursor is in the middle of a line.

The Display File Handler can operate simultaneously with the scroller program, permitting graphic displays and monitor dialogue to appear on the screen at the same time. It does this by inserting its internal display file into the display processor loop through the text buffer. However, the following should be noted. Under the SJ Monitor, if a program using the display for graphics is running with the scroller in use (that is, GT ON is in effect), and the program does a soft exit (.EXIT with R0 not equal to 0) with the display stopped, the display remains stopped until a CTRL/C is typed at the keyboard.

This can be recognized by failure of the monitor to echo on the screen when expected. If the scroller text display disappears after a program exit, always type CTRL/C to restore. If CTRL/C fails to restore the display, the running program probably has an error.

Four scroller control characters provide the user with the capability of halting the scroller, advancing the scrolling in page sections, and printing hard copy from the scroller.

NOTE

The scroller logic does not limit the length of a line, but the length of text lines affects the number of lines which may be displayed, since the text buffer is finite. As text lines become longer, the scroller logic may delete extra lines to make room for new text, temporarily decreasing the number of lines displayed.

A.2 Description of Graphics Macros

The facilities of the Display File Handler are accessed through a set of macros, contained in VTMAC, which generate assembly language calls to the Handler at assembly time. The calls take the form of subroutine calls to the subroutines in VTLIB. Arguments are passed to the subroutines through register 0 and, in the case of the .TRACK call, through both register 0 and the stack.

This call convention is similar to Version 1 RT-11 I/O macro calls, except that the subroutine call instruction is used instead of the EMT instruction. If a macro requires an argument but none is specified, it is assumed that the address of the argument has already been placed in register 0. The programmer should not assume that R0 is preserved through the call.

A.2.1 .BLANK

The .BLANK request temporarily blanks the user display file specified in the request. It does this by bypassing the call to the user display file, which prevents the display processor from cycling through the user file, effectively

blanking it. This effect can later be canceled by the .RESTR request, which restores the user file. When the call returns, the user is assured the display processor is not in the file that was blanked.

Macro Call: .BLANK faddr

where:

faddr is the address of the user display file to be blanked

Errors:

No error is returned. If the file specified was not found in the Handler file or has already been blanked, the request is ignored.

A.2.2 .CLEAR

The .CLEAR request initializes the Display File Handler, clearing out any calls to user display files and resetting all of the internal flags and pointers.

After initialization with .LNKRT (Section A.2.4), the .CLEAR request can be used any time in a program to clear the display and to reset pointers. All calls to user files are deleted and all pointers to status buffers are reset. They must be re-inserted if they are to be used again.

Macro Call: .CLEAR

Errors:

None.

Example:

This example uses a .CLEAR request to initialize the Handler then later uses the .CLEAR to re-initialize the display. The first .CLEAR is used for the case when a program may be restarted after a CTRL C or other exit.

```
            BR RSTRT

EX1:        BIS #20000,@#44     ;SET REENTER BIT IN JSW
RSTRT:      .UNLNK              ;CLEARS LINK FLAG FOR RESTART
            .LNKRT              ;SET UP VECTORS, START DISPLAY
            .CLEAR              ;INITIALIZE HANDLER
            .INSRT #FILE1       ;DISPLAY A PICTURE
1$:         .TTYIN              ;WAIT FOR A KEY STRIKE
            CMPB #12,R0         ;LINE FEED?
            BNE 1$              ;NO, LOOP
            .CLEAR              ;YES, CLEAR DISPLAY
            .INSRT #FILE2       ;DISPLAY NEW PICTURE
                 .
                 .
                 .
FILE1:      POINT           ;AT POINT (0,500)
            0
            500
            LONGV           ;DRAW A LINE
            500!INTX        ;TO (500,500)
            0
            DRET
            0
```

```
FILE2:  POINT           ;AT POINT (500,0)
        500
        0
        LONGV           ;DRAW A LINE
        0!INTX          ;TO (500,500)
        500
        DRET
        0

        .END EX1
```

A.2.3 .INSRT

The .INSRT request inserts a call to the user display file specified in the request into the Display File Handler's internal display file. .INSRT causes the display processor to cycle through the user file as a subroutine to the internal file. The handler permits two user files at one time. The call inserted in the handler looks like the following:

```
DJSR            ;DISPLAY SUBROUTINE
.+4             ;RETURN ADDRESS
.faddr          ;SUBROUTINE ADDRESS
```

The call to the user file is removed by replacing its address with the address of a null display file. The user file is blanked by replacing the DJSR with a DJMP instruction, bypassing the user file.

Macro Call: .INSRT faddr

where:

faddr is the address of the user display file to be inserted

Errors:

The .INSRT request returns with the C bit set if there was an error in processing the request. An error occurs only when the Handler's display file is full and cannot accept another file. If the user file specified exists, the request is not processed. Two display files with the same starting address cannot be inserted.

Example:

See the examples in Sections A.2.2 and A.2.4.

A.2.4 .LNKRT

The .LNKRT request sets up the display interrupt vectors and possibly links the Display File Handler to the scroll text buffer in the RT-11 monitor. It must be the first call to the Handler, and is used whether or not the RT-11 monitor is using the display for console output (that is, the KMON command GT ON has been entered).

The .LNKRT request used with Version 03 and later RT-11 monitors enables a display application program to determine the environment in which it is operating. Error codes are provided for the situations where there is no display

hardware present on the system or the display hardware is already being used by another task (for example, a foreground job in the foreground/background version).

The existence of the monitor scroller and the size of the Handler's subpicture stack are also returned to the caller. If a previous call to .LNKRT was made without a subsequent .UNLNK, the .LNKRT call is ignored and an error code is returned.

Macro Call: .LNKRT

Errors:

Error codes are returned in R0, with the N condition bit set.

Code	Meaning
-1	No VT11 display hardware is present on this system.
-2	VT11 hardware is presently in use.
-3	Handler has already been linked.

On completion of a successful .LNKRT request, R0 will contain the display subroutine stack size, indicating the depth to which display subroutines may be nested. The N bit will be zero.

If the RT-11 monitor scroll text buffer was not in memory at the time of the .LNKRT, the C bit will be returned set. The KMON commands GT ON and GT OFF cannot be issued while a task is using the display.

Example:

```
START:  .LNKRT                  ;LINK TO MONITOR
        BMI     ERROR           ;ERROR DOING LINK
        BCS     CONT            ;NO SCROLL IF C SET
        .SCROL  #SBUF           ;ADJUST SCROLL PARAMETERS
CONT:   .INSRT  #FILE1          ;DISPLAY A PICTURE
1$:     .TTYIN                  ;WAIT FOR KEY STRIKE
        CMPB    #12,R0          ;LINE FEED?
        BNE     1$              ;NO, LOOP
        .UNLNK                  ;YES, UNLINK AND EXIT
        .EXIT

SBUF:   .BYTE   5               ;LINE COUNT OF 5
        .BYTE   7               ;INTENSITY 7 (SCALE OF 1-8)
        .WORD   1000            ;POSITION OF TOP LINE

FILE1:  POINT                   ;AT POINT (500,500)
        500
        500
        CHAR                    ;DISPLAY SOME TEXT
        .ASCII  /FILE1 THIS IS FILE1. TYPE CR TO EXIT/
        .EVEN
        DRET
        0

ERROR:  Error routine
```

A.2.5 .LPEN

The .LPEN request transfers the address of a light pen status data buffer to VTBASE. Once the buffer pointer has been passed to the Handler, the light pen interrupt handler in VTBASE will transfer display processor status data to the buffer, depending on the state of the buffer flag.

The buffer must have seven contiguous words of storage. The first word is the buffer flag, and it is initially cleared (set to zero) by the .LPEN request. When a light pen interrupt occurs, the interrupt handler transfers status data to the buffer and then sets the buffer flag non-zero. The program can loop on the buffer flag when waiting for a light pen hit (although doing this will tie up the processor; in a foreground/background environment, timed waits would be more desirable). No further data transfers take place, despite the occurrence of numerous light pen interrupts, until the buffer flag is again cleared to zero. This permits the program to process the data before it is destroyed by another interrupt.

The buffer structure looks like this:

Buffer Flag
Name
Subpicture Tag
Display Program Counter (DPC)
Display Status Register (DSR)
X Status Register (XSR)
Y Status Register (YSR)

The Name value is the contents of the software Name Register (described in A.3.5) at the time of interrupt. The Tag value is the tag of the subpicture being displayed at the time of interrupt. The last four data items are the contents of the display processor status registers at the time of interrupt. They are described in detail in Table A–1.

Macro Call: .LPEN baddr

where:

baddr is the address of the 7-word light pen status data buffer

Errors:

None.

If a .LPEN was already issued and a buffer specified, the new buffer address replaces the previous buffer address. Only one light pen buffer can be in use at a time.

Example:

```
            .INSRT    #LFILE     ;DISPLAY LFILE
            .LPEN     #LBUF      ;SET UP LPEN BUFFER
LOOP:       TST       LBUF       ;TEST LBUF FLAG, WHICH
            BEQ       LOOP       ;WILL BE SET NON-ZERO
                                 ;ON LIGHT PEN HIT.
;PROCESS DATA IN LBUF HERE.
```

```
                                    ;DATA IN LBUF
        CLR         LBUF            ;CLEAR THE BUFFER FLAG
                                    ;PERMITTING ANOTHER "HIT"
        BR          LOOP            ;GO WAIT FOR IT
LBUF:   .BLKW       7               ;SEVEN WORD LPEN BUFFER
LFILE:
                .
                .
```

Table A–1: Description of Display Status Words

Bits	Significance
Display Program Counter (DPC=172000)	
0–15	Address of display processor program counter at time of interrupt.
Display Status Register (DSR=172002)	
0–1	Line Type
2	Spare
3	Blink
4	Italics
5	Edge Indicator
6	Shift Out
7	Light Pen Flag
8–10	Intensity
11–14	Mode
15	Stop Flag
X Status Register (XSR=172004)	
0–9	X Position
10–15	Graphplot Increment
Y Status Register (YSR=172006)	
0–9	Y Position
10–15	Character Register

A.2.6 .NAME

The .NAME request has been added to the Version 03 and later Display File Handler. The contents of the name register are now stacked when a subpicture call is made. When a light pen interrupt occurs, the contents of the name register stack may be recovered if the user program has supplied the address of a buffer through the .NAME request.

The buffer must have a size equal to the stack depth (default is 10) plus one word for the flag. When the .NAME request is entered, the address of the buffer is passed to the Handler and the first word (the flag word) is cleared. When a light pen hit occurs, the stack's contents are transferred and the flag is set non-zero.

Macro Call: .NAME baddr

where:

baddr is the address of the name register buffer

Errors:

None.

If a .NAME request has been previously issued, the new buffer address replaces the previous buffer address.

A.2.7 .REMOV

The .REMOV request removes the call to a user display file previously inserted in the handler's display file by the .INSRT request. All reference to the user file is removed, unlike the .BLANK request, which merely bypasses the call while leaving it intact.

Macro Call: .REMOV faddr

where:

faddr is the address of the display file to be removed

Errors:

No errors are returned. If the file address given cannot be found, the request is ignored.

A.2.8 .RESTR

The .RESTR request restores a user display file that was previously blanked by a .BLANK request. It removes the by-pass of the call to the user file, so that the display processor once again cycles through the user file.

Macro Call: .RESTR faddr

where:

faddr is the address of the user file that is to be restored to view

Errors:

No errors are returned. If the file specified cannot be found, the request is ignored.

A.2.9 .SCROL

This request is used to modify the appearance of the Display Monitor's text display. The .SCROL request permits the programmer to change the maximum line count, intensity and the position of the top line of text of the scroller. The request passes the address of a two-word buffer which contains the parameter specifications. The first byte is the line count, the second byte is the intensity, and the second word is the Y position. Line count, intensity and Y position must all be octal numbers. The intensity may be any number from 0 to 7, ranging from dimmest to brightest. (If an intensity of 0 is specified, the scroller text will be almost unnoticeable at a BRIGHTNESS knob setting less than one-half.) The scroller parameter change is temporary, since an .UNLNK or CTRL/C restores the previous values.

Macro Call: .SCROL baddr

where:

baddr is the address of the two-word scroll parameters buffer

Errors:

No errors are returned. No checking is done on the values of the parameters. A zero argument is interpreted to mean that the parameter value is not to be changed. A negative argument causes the default parameter value to be restored.

Example:

```
        .SCROL    #SCBUF      ;ADJUST SCROLL PARAMETERS
        .
        .
        .

SCBUF:  .BYTE 5               ;DECREASE #LINES TO 5.
        .BYTE 0               ;LEAVE INTENSITY UNCHANGED.
        .WORD 300             ;TOP LINE AT Y=300.
```

A.2.10 .START

The .START request starts the display processor if it was stopped by a .STOP directive. If the display processor is running, it is stopped first, then restarted. In either case, the subpicture stack is cleared and the display processor is started at the top of the handler's internal display file.

Macro Call: .START

Errors:

None.

A.2.11 .STAT

The .STAT request transfers the address of a seven-word status buffer to the display stop interrupt routine in VTBASE. Once the transfer has been made, display processor status data is transferred to the buffer by the display stop interrupt routine in VTBASE whenever a .DSTAT or .DHALT instruction is encountered (see Sections A.3.3 and A.3.4). The transfer is made only when the buffer flag is clear (zero). After the transfer is made, the buffer flag is set non-zero and the .DSTAT or .DHALT instruction is replaced by a .DNOP (Display NOP) instruction.

The status buffer must be a seven-word, contiguous block of memory. Its contents are the same as the light pen status buffer. For a detailed description of the buffer and an explanation of the status words, see Section A.2.5 and Table A–1.

Macro Call: .STAT baddr

where:

baddr is the address of the status buffer receiving the data

Errors:

No errors are indicated. If a buffer was previously set up, the new buffer address is replaced as the old buffer address.

A.2.12 .STOP

The .STOP request "stops" the display processor. It actually effects a stop by preventing the DPU from cycling through any user display files. It is useful for stopping the display during modification of a display file, a risky task when the display processor is running. However, a .BLANK could be equally useful for this purpose, since the .BLANK request does not return until the display processor has been removed from the user display file being blanked.

Macro Call: .STOP

Errors:

None.

NOTE

Since the display processor must cycle through the text buffer in the Display Monitor in order for console output to be processed, the text buffer remains visible after a .STOP request is processed, but all user files disappear.

A.2.13 .SYNC/.NOSYN

The .SYNC and .NOSYN requests provide program access to the power line synchronization feature of the display processor. The .SYNC request enables synchronization and the .NOSYN request disables it (the default case).

Synchronization is achieved by stopping the display and restarting it when the power line frequency reaches a trigger point, e.g., a peak or zero-crossing. Synchronization has the effect of fixing the display refresh time. This may be useful in some cases where small amounts of material are displayed but the amount frequently changes, causing changes in intensity. In most cases, however, using synchronization increases flicker.

Macro Calls: .SYNC
.NOSYN

Errors:

None.

A.2.14 .TRACK

The .TRACK request causes the tracking object to appear on the display CRT at the position specified in the request. The tracking object is a diamond-shaped display figure which is light-pen sensitive. If the light pen is placed over the tracking object and then moved, the tracking object follows the light pen, trying to center itself on the pen.

The tracking object first appears at a position specified in a two-word buffer whose address was supplied with the .TRACK request. As the tracking object moves to keep centered on the light pen, the new center position is returned to the buffer. A new set of X and Y values is returned for each light pen interrupt.

The tracking object cannot be lost by moving it off the visible portion of the display CRT. When the edge flag is set, indicating a side of the tracking object is crossing the edge of the display area, the tracking object stops until moved toward the center. To remove the tracking object from the screen, repeat the .TRACK request without arguments.

The .TRACK request may also include the address of a completion routine as the second argument. If a .TRACK completion routine is specified, the light pen interrupt handler passes control to the completion routine at interrupt level. The completion routine is called as a subroutine and the exit statement must be an RTS PC. The completion routine must also preserve any registers it may use.

Macro Call: .TRACK baddr, croutine

where:

baddr	is the address of the two-word buffer containing the X and Y position for the track object
croutine	is the address of the completion routine

Errors:

None.

Example:

See Section A.10.

A.2.15 .UNLNK

The .UNLNK request is used before exiting from a program. In the case where the scroller is present, .UNLNK breaks the link, established by .LNKRT, between the Display File Handler's internal display file and the scroll file in the Display Monitor. The display processor is started cycling in the scroll text buffer, and no further graphics may be done until the link is established again. In the case where no scroller exists, the display processor is simply left stopped.

Macro Call: .UNLNK

Errors:

No errors are returned. An internal link flag is checked to determine if the link exists. If it does not exist, the request is ignored.

A.3 Extended Display Instructions

The Display File Handler offers the assembly language graphics programmer an extended display processor instruction set, implemented in software through the use of the Load Status Register A (LSRA) instruction. The extended instruction set includes: subroutine call, subroutine return, display status return, display halt, and load name register.

A.3.1 DJSR Subroutine Call Instruction

The DJSR instruction (octal code is 173400) simulates a display subroutine call instruction by using the display stop instruction (LSRA instruction with interrupt bits set). The display stop interrupt handler interprets the non-zero word following the DJSR as the subroutine return address, and the second word following the DJSR as the address of the subroutine to be called. The instruction sequence is:

DJSR
Return address
Subroutine address

Example:

To call a subroutine SQUARE:

```
POINT           ;POSITION BEAM
100             ;AT (100,100)
100
DJSR            ;THEN CALL SUBROUTINE
.+4
SQUARE          ;TO DRAW A SQUARE
DRET
0
```

The use of the return address preceding the subroutine address offers several advantages. For example, the BASIC–11 graphics software uses the return address to branch around subpicture tag data stored following the subpicture address. This structure is described in Section A.5.3. In addition, a subroutine may be temporarily bypassed by replacing the DJSR code with a DJMP instruction, without the need to stop the display processor to make the by-pass.

The address of the return address is stacked by the display stop interrupt handler on an internal subpicture stack. The stack depth is conditionalized and has a default depth of 10. If the stack bottom is reached, the display stop interrupt handler attempts to protect the system by rejecting additional subroutine calls. In that case, the portions of the display exceeding the legal stack depth will not be displayed.

A.3.2 DRET Subroutine Return Instruction

The DRET instruction provides the means for returning from a display file subroutine. It uses the same octal code as DJSR, but with a single argument of zero. The DRET instruction causes the display stop interrupt handler to pop its subpicture stack and fetch the subroutine return address.

Example:

```
SQUARE:  LONGV            ;DRAW A SQUARE
         100!INTX
         0
         0!INTX
         100
         100!INTX!MINUSX
         0
         0!INTX
         100!MINUSX
         DRET             ;RETURN FROM SUBPICTURE
         0
```

A.3.3 DSTAT Display Status Instruction

The DSTAT instruction (octal code is 173420) uses the LSRA instruction to produce a display stop interrupt, causing the display stop interrupt handler to return display status data to a seven-word user status buffer. The status buffer must first have been set up with a .STAT macro call (if not, the DSTAT is ignored and the display is resumed). The first word of the buffer is set non-zero to indicate the transfer has taken place, and the DSTAT is replaced with a DNOP (display NOP). The first word is the buffer flag and the next six words contain name register contents, current subpicture tag, display program counter, display status register, display X register, and display Y register. After transfer of status data, the display is resumed.

A.3.4 DHALT Display Halt Instruction

The DHALT instruction (octal code is 173500) operates similarly to the DSTAT instruction. The difference between the two instructions is that the DHALT instruction leaves the display processor stopped when exiting from the interrupt. A status data transfer takes place provided the buffer was initialized with a .STAT call. If not, the DHALT is ignored.

Example:

```
         .STAT     #SBUF           ;INIT BUFFER
         MOV       #DHALT,STPLOC   ;INSERT DHALT
         .INSRT    #DFILE          ;DISPLAY THE PICTURE
1$:      TST       SBUF            ;DHALT PROCESSED?
         BEQ       1$              ;NO, WAIT
          .
          .
SBUF:    .BLKW 7                   ;STATUS BUFFER
DFILE:   POINT                     ;POSITION NEAR TOP OF 12" TUBE
         .WORD     500,1350
         LONGV                     ;DRAW A LINE, MAYBE OVER EDGE
         .WORD     0,400           ;IF IT IS A 12" SCOPE.
STPLOC:  DNOP                      ;STATUS WILL BE RETURNED AT
         DRET                      ;THIS POINT
         0
```

A.3.5 DNAME Load Name Register Instruction

The Display File Handler provides a name register capability through the use of the display stop interrupt. When a DNAME instruction (octal code is 173520) is encountered, a display stop interrupt is generated. The display stop handler stores the argument following the DNAME instruction in an internal

software register called the "name register." The current name register contents are returned whenever a DSTAT or DHALT is encountered, and more importantly, whenever a light pen interrupt occurs. The use of a "name" (with a valid range from 1 to 77777) enables the programmer to label each element of the display file with a unique name, permitting the easy identification of the particular display element selected by the light pen.

The name register contents are stacked on a subpicture call and restored on return from the subpicture.

Example:

The SQUARE subroutine with "named" sides.

```
SQUARE:  DNAME                  ;NAME IS
         10                     ;10
         LONGV                  ;DRAW A SIDE
         100!INTX
         0
         DNAME                  ;THIS SIDE IS NAMED
         11                     ;11
         0!INTX                 ;STILL IS LONG VECTOR MODE
         100
         DNAME
         12
         100!INTX!MINUSX
         0
         DNAME
         13
         0!INTX
         100!MINUSX
         DRET                   ;RETURN FROM SUBPICTURE
         0
```

A.4 Using the Display File Handler

Graphics programs which intend to use the Display File Handler for display processor management can be written in MACRO assembly language. The display code portions of the program may use the mnemonics described in Section A.7. Calls to the Handler should have the format described in Section A.6.

The Display File Handler is supplied in two pieces, a file of MACRO definitions and a library containing the Display File Handler modules.

MACRO Definition File: VTMAC.MAC

Display File Handler: VTLIB.OBJ (consisting of:)

VTBASE.OBJ
VTCAL1.OBJ
VTCAL2.OBJ
VTCAL3.OBJ
VTCAL4.OBJ

A.4.1 Assembling Graphics Programs

To assemble a graphics program using the display processor mnemonics or the Display Handler macro calls, the file VTMAC.MAC must be assembled with the program, and must precede the program in the assembler command string.

Example:

Assume PICTUR.MAC is a user graphics program to be assembled. An assembler command string would look like this:

```
MACRO VTMAC+PICTUR/OBJECT
```

A.4.2 Linking Graphics Programs

Once assembled with VTMAC, the graphics program must be linked with the Display File Handler, which is supplied as a single concatenated object module, VTHDLR.OBJ. The Handler may optionally be built as a library, following the directions in A.8.5. The advantage of using the library when linking is that the Linker will select from the library only those modules actually used. Linking with VTHDLR.OBJ results in all modules being included in the link.

To link a user program called PICTUR.OBJ using the concatenated object module supplied with RT-11:

```
LINK PICTUR,VTHDLR
```

To link a program called PICTUR.OBJ using the VTLIB library built by following the directions in A.8.5, be sure to use the Version 03 Linker:

```
LINK PICTUR,VTLIB
```

VTLIB (Handler Modules):

Module	CSECT	Contains	Globals
VTCAL1	$GT1	.CLEAR	$VINIT
		.START	$VSTRT
	.STOP	$VSTOP	
		.INSRT	$VNSRT
		.REMOV	$VRMOV
VTCAL2	$GT2	.BLANK	$VBLNK
		.RESTR	$VRSTR
VTCAL3	$GT3	.LPEN	$VLPEN
		.NAME	$NAME
		.STAT	$VSTPM
		.SYNC	$SYNC
		.NOSYN	$NOSYN
		.TRACK	$VTRAK
VTCAL4	$GT4	.LNKRT	$VRTLK
		.UNLNK	$VUNLK
		.SCROL	$VSCRL
VTBASE	$GTB	Interrupt handlers and internal display file	$DFILE

The file modules in VTHDLR can be used in three different ways. When space is not critical, the most straightforward way is to link VTHDLR directly with a display program. The following command is an example.

```
LINK PICTUR,VTHDLR
```

It is often necessary to conserve space, however, and selective loading of modules is possible by first creating an indexed object module library from VTHDLR and then by making global calls within the display program. The following command creates an indexed object module library.

```
LIBRARY/CREATE VTLIB VTHDLR
```

To further conserve space with overlays, it is also possible to extract individual object modules from a library and create separate object module files. For example, to link a display program using overlays, the following statements are a typical sequence of creating, extracting and linking commands. (NOTE: the modules VTCAL1 and VTCAL2 must be in the same overlay if any global in either one is used.)

```
                .
                .
                .
.LIBRARY/CREATE VTLIB VTHDLR
                .
                .
                .
.LIBRARY/EXTRACT VTLIB VTCAL1
GLOBAL? $VSTRT !moves entire module with $VSTRT to VTCAL1
GLOBAL? !Terminates prompting sequence
.LIBRARY/EXTRACT VTLIB VTCAL2
GLOBAL? $VBLNK !Moves the entire module to VTCAL2
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTCAL3
GLOBAL? $VLPEN !Moves the entire module
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTCAL4
GLOBAL? $VRTLK !Moves the entire module
GLOBAL?
.LIBRARY/EXTRACT VTLIB VTBASE
GLOBAL? $DFILE !Moves the entire module
GLOBAL?
                .
                .
                .
.LINK/PROMPT PICTUR,VTBASE
*VTCAL1,VTCAL2,VTCAL3/O:1
*VTCAL4/O:1
*//
                .
                .
                .
```

A.5 Display File Structure

The Display File Handler supports a variety of display file structures, takes over the job of display processor management for the programmer, and may be used for both assembly language graphics programming and for systems program development. For example, the Handler supports the tagged subpicture file structure used by the BASIC–11 graphics software, as well as simple file structures. These are discussed in this section.

A.5.1 Subroutine Calls

A subroutine call instruction, with the mnemonic DJSR, is implemented using the display stop (DSTOP) instruction with an interrupt. The display stop interrupt routine in the Display File Handler simulates the DJSR instruction, and this allows great flexibility in choosing the characteristics of the DJSR instruction.

The DJSR instruction stops the display processor and requests an interrupt. The DJSR instruction may be followed by two or more words, and in this implementation the exact number may be varied by the programmer at any time. The basic subroutine call has this form:

```
DJSR
Return Address
Subroutine Address
```

In practice, simple calls to subroutines could look like:

```
DJSR
.WORD       .+4
.WORD       SUB
```

where SUB is the address of the subroutine. Control will return to the display instruction following the last word of the subroutine call. This structure permits a call to the subroutine to be easily by-passed without stopping the display processor, by replacing the DJSR with a display jump (DJMP) instruction:

```
DJMP
.WORD       .+4
.WORD       SUB
```

A more complex display file structure is possible if the return address is generalized:

```
.DJSR
.WORD       NEXT
.WORD       SUB
```

where NEXT is the generalized return address. This is equivalent to the sequence:

```
DJSR
.WORD       .+4
.WORD       SUB
DJMP
.WORD       NEXT
```

It is also possible to store non-graphic data such as tags and pointers in the subroutine call sequence, such as is done in the tagged subpicture display file structure of the BASIC–11 graphics software. This technique looks like:

```
            DJSR
            .WORD       NEXT
            .WORD       SUB
            DATA
NEXT:         .
              .
              .
```

For simple applications where the flexibility of the DJSR instruction described above is not needed and the resultant overhead is not desired, the

Display File Handler (VTBASE.MAC and VTCALL.MAC) can be conditionally re-assembled to produce a simple DJSR call. If NOTAG is defined during the assembly, the Handler will be configured to support this simple DJSR call:

```
DJSR
.WORD      SUB
```

where SUB is the address of the subroutine. Defining NOTAG will eliminate the subpicture tag capability, and with it the tracking object, which uses the tag feature to identify itself to the light pen interrupt handler.

Whatever the DJSR format used, all subroutines and the user main file must be terminated with a subroutine return instruction. This is implemented as a display stop instruction (given the mnemonic DRET) with an argument of zero. A subroutine then has the form:

```
SUB:    Display Code
        DRET
        .WORD 0
```

A.5.2 Main File/Subroutine Structure

A common method of structuring display files is to have a main file which calls a series of display subroutines. Each subroutine will produce a picture element and may be called many times to build up a picture, producing economy of code. If the following macros are defined:

```
.MACRO     CALL <ARG>
DJSR
.WORD      .+4
.WORD      ARG
.ENDM
.MACRO     RETURN
DRET
.WORD      0
.ENDM
```

then a main file/subroutine file structure would look like:

```
;MAIN DISPLAY FILE
;
MAIN:      Display Code
           CALL SUB1        ;CALL SUBROUTINE 1
           Display Code
           CALL SUB2        ;CALL SUBROUTINE 2
             .              ;ETC
             .
             .
           RETURN
;
;DISPLAY   SUBROUTINES
;
SUB1:      Display Code     ;SUBROUTINE 1
           RETURN
;
SUB2:      Display Code     ;SUBROUTINE 2
           RETURN
             .              ;ETC.
             .
             .
```

A.5.3 BASIC–11 Graphic Software Subroutine Structure

An example of another method of structuring display files is the tagged subpicture structure used by BASIC-11 graphic software. The display file is divided into distinguishable elements called subpictures, each of which has its own unique tag.

The subpicture is constructed as a subroutine call followed by the subroutine. It is essentially a merger of the main file/subroutine structure into an in-line sequence of calls and subroutines. As such, it facilitates the construction of display files in real time, one of the important advantages of BASIC-11 graphic software.

The following is an example of the subpicture structure. Each subpicture has a call to a subroutine with the return address set to be the address of the next subpicture. The subroutine called may either immediately follow the call, or may be a subroutine defined as part of a subpicture created earlier in the display file. This permits a subroutine to be used by several subpictures without duplication of code. Each subpicture has a tag to identify it, and it is this tag which is returned by the light pen interrupt routine. To facilitate finding subpictures in the display file, they are made into a linked list by inserting a forward pointer to the next tag.

```
SUB1:   DJSR                    ;START OF SUBPICTURE 1
        .WORD   SUB2            ;NEXT SUBPICTURE
        .WORD   SUB1+12         ;JUMP TO THIS SUBPICTURE
        .WORD   1               ;TAG = 1
        .WORD   SUB2+6          ;POINTER TO NEXT TAG

;BODY OF SUBPICTURE 1

        DRET                    ;RETURN FROM
        0                       ;SUBPICTURE 1

SUB2:   DJSR                    ;START SUBPICTURE 2
        .WORD   SUB3            ;NEXT SUBPICTURE
        .WORD   SUB2+12         ;JUMP TO THIS SUBPICTURE
        .WORD   2               ;TAG 2
        .WORD   SUB3+6          ;PTR TO NEXT TAG
;BODY OF SUBPICTURE 2

        DRET                    ;RETURN FROM
        .WORD   0               ;SUBPICTURE 2

SUB3:   DJSR                    ;START SUBPICTURE 3
        .WORD   SUB4            ;NEXT SUBPICTURE
        .WORD   SUB1+12         ;COPY SUBPICTURE 1
                                ;FOR THIS SUBPICTURE
        .WORD   3               ;BUT TAG IT 3.
        .WORD   SUB4+6          ;PTR TO NEXT TAG

SUB4:   DJSR                    ;START SUBPICTURE 4
        .                       ;ETC.
        .
        .
```

A.6 Summary of Graphics MACRO Calls

Mnemonic	Function	MACRO Call (see Note 1)	Assembly Language Expansion (see Note 2)
.BLANK	Temporarily blanks a user display file.	.BLANK faddr	.GLOBL $VBLNK .IF NB, faddr MOV faddr, ^100 .ENDC JSR ^O7, $VBLNK
.CLEAR	Initializes handler.	.CLEAR	.GLOBL $VINIT JSR ^O7, $VINIT
.INSRT	Inserts a call to user display file in handler's master display file.	.INSRT faddr	.GLOBL $VNSRT .IF NB, faddr MOV faddr, ^O0 .ENDC JSR ^O7, $VNSRT
.LNKRT	Sets up vectors and links display file handler to RT-11 scroller.	.LNKRT	.GLOBL $VRTLK JSR ^O7, $VRTLK
.LPEN	Sets up light pen status buffer.	.LPEN baddr	.GLOBL $VLPEN .IF NB, baddr MOV baddr, ^O0 .ENDC JSR ^O7, $VLPEN
.NAME	Sets up buffer to receive name register stack contents.	.NAME \baddr	.GLOBL $NAME .IF NB, baddr MOV .BEDDR, ^O0 .ENDC JSR ^O7, $NAME
.NOSYN	Disables power line synchronization.	.NOSYN	.GLOBL $NOSYN JSR ^O7, $NOSYN
.REMOV	Removes the call to a user display file.	.REMOV faddr	.GLOBL $VRMOV .IF NB, faddr MOV faddr, ^O0 .ENDC JSR ^O7, $VRMOV
.RESTR	Unblanks the user display file.	.RESTR faddr	.GLOBL $VRSTR IF NB, faddr MOV faddr, ^O0 ENDC JSR ^O7, $VRSTR
.SCROL	Adjusts monitor scroller parameters.	.SCROL baddr	.GLOBL $VSCRL .IF NB, baddr MOV baddr, ^O0 .ENDC JSR ^O7, $VSCRL

Mnemonic	Function	MACRO Call (see Note 1)	Assembly Language Expansion (see Note 2)
.START	Starts the display.	.START	.GLOBL $VSTRT JSR ^O7, $VSTRT
.STAT	Sets up status buffer.	.STAT baddr	.GLOBL $VSTPM .IF NB, baddr MOV baddr, ^O0 .ENDC JSR ^O7, $VSTPM
.STOP	Stops the display.	.STOP	.GLOBL $VSTOP JSR ^O7, $VSTOP
.SYNC	Enables power line synchronization.	.SYNC	.GLOBL $SYNC JSR ^O7, $SYNC

.TRACK	Enables the track object.	.TRACK baddr, croutine	.GLOBL $VTRAK
			IF NB, baddr
			MOV baddr, ^O0
			.ENDC
			.IF NB, croutine
			MOV croutine,-(^O6)
			.IFF
			CLR-(^O6)
			.ENDC
			.NARG T
			.IF EQ, T
			CLR ^O0
			.ENDC
			JSR ^O7, $VTRAK

.UNLNK	Unlinks display handler from RT-11 if linked (otherwise leaves display stopped).	.UNLNK	.GLOBL $VUNLK JSR ^O7, $VUNLK

NOTE 1

baddr	Address of data buffer.
faddr	Address of start of user display file.
croutine	Address of .TRACK completion routine.

NOTE 2

The lines preceded by a dot will not be assembled. The code they enclose may or may not be assembled depending on the conditionals.

A.7 Display Processor Mnemonics

Mnemonic		Value	Function
CHAR	=	100000	Character Mode
SHORTV	=	104000	Short Vector Mode
LONGV	=	110000	Long Vector Mode
POINT	=	114000	Point Mode
GRAPHX	=	120000	Graphplot X Mode
GRAPHY	=	124000	Graphplot Y Mode
RELATV	=	130000	Relative Point Mode
INT0	=	2000	Intensity 0 (Dim)
INT1	=	2200	Intensity 1
INT2	=	2400	Intensity 2
INT3	=	2600	Intensity 3
INT4	=	3000	Intensity 4
INT5	=	3200	Intensity 5
INT6	=	3400	Intensity 6
INT7	=	3600	Intensity 7 (Bright)
LPOFF	=	100	Light Pen Off
LPON	=	140	Light Pen On
BLKOFF	=	20	Blink Off
BLKON	=	30	Blink On
LINE0	=	4	Solid Line
LINE1	=	5	Long Dash
LINE2	=	6	Short Dash
LINE3	=	7	Dot Dash
DJMP	=	160000	Display Jump
DNOP	=	164000	Display No Operation
STATSA	=	170000	Load Status A Instruction
LPLITE	=	200	Light Pen Hit On
LPDARK	=	300	Light Pen Hit Off
ITAL0	=	40	Italics Off
ITAL1	=	60	Italics On
SYNC	=	4	Halt and Resume Synchronized
STATSB	=	174000	Load Status B Instruction
INCR	=	100	Graphplot Increment
Vector/Point Mode			
INTX	=	40000	Intensity Vector or Point
MAXX	=	1777	Maximum X Component
MAXY	=	1377	Maximum Y Component
MINUSX	=	20000	Negative X Component
MINUSY	=	20000	Negative Y Component

Mnemonic		Value	Function
Short Vector Mode			
SHIFTX	=	200	
MAXSX	=	17600	Maximum X Component
MAXSY	=	77	Maximum Y Component
MISVX	=	20000	Negative X Component
MISVY	=	100	Negative Y Component

A.8 Assembly Instructions

A.8.1 General Instructions

All programs can be assembled in 16K, using RT-11 MACRO. All assemblies and all links should be error free. The following conventions are assumed:

1. Default file types are not explicitly typed. These are .MAC for source files, .OBJ for assembler output, and .SAV for Linker output.
2. The default device (DK) is used for all files in the example command strings.
3. Listings and link maps are not generated in the example command strings.

A.8.2 VTBASE

To assemble VTBASE with RT-11 link-up capability.

```
MACRO VTBASE
```

A.8.3 VTCAL1 - VTCAL4

To assemble the modules VTCAL1 through VTCAL4:

```
MACRO VTCAL1,VTCAL2,VTCAL3,VTCAL4
```

A.8.4 VTHDLR

To create the concatenated handler module:

```
COPY/BINARY VTCAL1.OBJ,VTCAL2.OBJ,VTCAL3.OBJ,-
VTCAL4.OBJ,VTBASE.OBJ VTHDLR.OBJ
```

A.8.5 Building VTLIB.OBJ

To build the VTLIB library:

```
LIBRARY/CREATE VTLIB VTHDLR
```

A.9 VTMAC

```
        .TITLE  VTMAC
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
; OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
;
; COPYRIGHT (C) 1978, DIGITAL EQUIPMENT CORPORATION.
;
; VTMAC IS A LIBRARY OF MACRO CALLS AND MNEMONIC DEFINITIIONS WHICH
; PROVIDE SUPPORT OF THE VT11 DISPLAY PROCESSOR. THE MACROS PRODUCE
; CALLS TO THE VT11 DEVICE SUPPORT PACKAGE, USING GLOBAL REFERENCES.

; MACRO TO GENERATE A MACRO WITH ZERO ARGUMENTS.
.MACRO MAC0 NAME,CALL
                .MACRO NAME
                .GLOBL  CALL
        JSR     PC,CALL
                .ENDM
.ENDM

; MACRO TO GENERATE A MACRO WITH ONE ARGUMENT

.MACRO MAC1 NAME,CALL
        .MACRO NAME ARG
        .IF NB,ARG
        MOV     ARG,%^O0
        .ENDC
        .GLOBL  CALL
        JSR     PC,CALL
        .ENDM
.ENDM

; MACRO TO GENERATE A MACRO WITH TWO OPTIONAL ARGUMENTS

.MACRO MAC2 NAME,CALL
        .MACRO NAME ARG1,ARG2
        .GLOBL  CALL
        .IF NB,ARG1



        MOV     ARG1,%^O0
        .ENDC
        .IF NB,ARG2
        MOV     ARG2,-(SP)
        .IFF
        CLR     -(SP)
        .NARG   T
        .IF EQ,T
        CLR     %^O0
        .ENDC
        .ENDC
        JSR     PC,CALL
        .ENDM
.ENDM

; MACRO LIBRARY FOR VT11:

MAC0    <.CLEAR>,<$VINIT>
MAC0    <.STOP>,<$VSTOP>
MAC0    <.START>,<$VSTRT>
MAC1    <.INSRT>,<$VNSRT>
MAC1    <.REMOV>,<$VRMOV>
MAC1    <.BLANK>,<$VBLNK>
MAC1    <.RESTR>,<$VRSTR>
MAC1    <.STAT>,<$VSTPM>
MAC1    <.LPEN>,<$VLPEN>
```

```
MAC1    <.SCROL>,<$VSCRL>
MAC2    <.TRACK>,<$VTRAK>
MAC0    <.LNKRT>,<$VRTLK>
MAC0    <.UNLNK>,<$VUNLK>

; MNEMONIC DEFINITIONS FOR THE VT11 DISPLAY PROCESSOR

DJMP=160000      ;DISPLAY JUMP
DNOP=164000      ;DISPLAY NOP
DJSR=173400      ;DISPLAY SUBROUTINE CALL
DRET=173400      ;DISPLAY SUBROUTINE RETURN
DNAME=173520     ;SET NAME REGISTER
DSTAT=173420     ;RETURN STATUS DATA
DHALT=173500     ;STOP DISPLAY AND RETURN STATUS DATA

CHAR=100000      ;CHARACTER MODE
SHORTV=104000    ;SHORT VECTOR MODE
LONGV=110000     ;LONG VECTOR MODE
POINT=114000     ;POINT MODE
GRAPHX=120000    ;GRAPH X MODE
GRAPHY=124000    ;GRAPH Y MODE
RELATV=130000    ;RELATIVE VECTOR MODE

INT0=2000        ;INTENSITY 0
INT1=2200
INT2=2400
INT3=2600
INT4=3000
INT5=3200
INT6=3400
INT7=3600

LPOFF=100        ;LIGHT PEN OFF
LPON=140         ;LIGHT PEN ON
BLKOFF=20        ;BLINK OFF
BLKON=30         ;BLINK ON
LINE0=4          ;SOLID LINE
LINE1=5          ;LONG DASH
LINE2=6          ;SHORT DASH
LINE3=7          ;DOT DASH

STATSA=170000    ;LOAD STATUS REG A
LPLITE=200       ;INTENSIFY ON LPEN HIT
LPDARK=300       ;DON'T INTENSIFY
ITAL0=40         ;ITALICS OFF
ITAL1=60         ;ITALICS ON
SYNC=4           ;POWER LINE SYNC

STATSB=174000    ;LOAD STATUS REG B
INCR=100         ;GRAPH PLOT INCREMENT
INTX=40000       ;INTENSIFY VECTOR OR POINT
MAXX=1777        ;MAXIMUM X INCR. = LONGV
MAXY=1377        ;MAXIMUM Y INCR. = LONGV
MINUSX=20000     ;NEGATIVE X INCREMENT
MINUSY=20000     ;NEGATIVE Y INCREMENT
MAXSX=17600      ;MAXIMUM X INCR. = SHORTV
MAXSY=77         ;MAXIMUM Y INCR. = SHORTV
MISVX=20000      ;NEGATIVE X INCR. = SHORTV
MISVY=100        ;NEGATIVE Y INCR. = SHORTV
```

A.10 Examples Using GTON

```
EXAMPLE #1      MACRO X03.04 18-MAY-77 14:49:44 PAGE 5

                                                .TITLE EXAMPLE #1
                                              ;
                                              ; THIS EXAMPLE USES THE .LPEN STATUS BUFFER AND THE
                                              ; NAME REGISTER TO MODIFY A DISPLAY FILE WITH THE LIGHT PEN.
                                              ;
                                                R0=%0
                                                R1=%1
                                                PC=%7
                                                JSW=44                  ;JOB STATUS WORD

                                                .MCALL  .TTINR,.EXIT,.PRINT
000000                                  START:  .LNKRT                  ;LINK TO MONITOR
000004  100004                                 BPL     1$              ;LINK UP ERROR?
000006                                          .PRINT  #EMSG           ;YES, PRINT MESSAGE
000014                                          .EXIT                   ;AND EXIT.
000016                                  1$:     .SCROL  #SCBUF          ;ADJUST SCROLL
000026                                          .PRINT  #MSG
000034                                          .INSRT  #DFILE          ;INSERT DISPLAY FILE
000044                                          .LPEN   #LBUF           ;SET UP LPEN BUFFER
000054  052737  000100  000044                  BIS     #100,@#JSW      ;SET JSW FOR TTINR
000062  005767  000072                  LTST:   TST     LBUF            ;LIGHT PEN HIT?
000066  001003                                  BNE     1$              ;YES
000070                                          .TTINR                  ;NO, ANY TT INPUT?
000072  103023                                  BCC     EXIT            ;YES, EXIT
000074  000772                                  BR      LTST            ;NO, LOOP AGAIN
000076  016777  000074  000102          1$:     MOV     I2,@IPTR        ;RESTORE PREVIOUS CODE
000104  016701  000050                          MOV     LBUF+2,R1       ;GET NAME VALUE
000110  005301                                  DEC     R1              ;SUBTRACT ONE
000112  006301                                  ASL     R1              ;MULTIPLY BY TWO
000114  060701                                  ADD     PC,R1           ;USE TO INDEX
000116  062701  000062                          ADD     #DTABL-.,R1     ;OFF TABLE DTABL.
000122  011167  000060                          MOV     (R1),IPTR       ;MOVE ADDR INTO IPTR
000126  016777  000042  000052                  MOV     I1,*IPTR        ;MODIFY THAT CODE
000134  005067  000016                          CLR     LBUF            ;CLEAR BUFFER FLAG TO
                                                                        ;ENABLE ANOTHER LP HIT.
                                                BR      LTST            ;LOOP AGAIN
000140  000750                          EXIT:   CMP     #12,R0          ;LINE FEED?
000142  022700  000012                          BNE     LTST            ;NO, GET ANOTHER
000146  001345                                  .UNLNK                  ;UNLINK FROM MONITOR
000150                                          .EXIT
000154                                  LBUF:   .BLKW   7               ;LPEN STATUS BUFFER
000156                                  I1:     .WORD   CHAR!INTS!BLKON!LPON
000174  103370                          I2:     .WORD   CHAR!INT4!BLKOFF!LPON
000176  103160
```

```
    44 000200  000252' 000272' 000312' DTABL:  .WORD   D1,D2,D3          ;TABLE OF DISPLAY FILE
    45                                                                   ;LOCATIONS TO BE MODIFIED
    46 000206  000252'                 IPTR:   .WORD   D1                ;PREVIOUS LOCATION MODIFIED
    47 000210  000002                  SCBUF:  .WORD   2                 ;SCROLL LINE COUNT
    48 000212  000100                          .WORD   100               ;SCROLL TOP Y POS.
    49 000214     041     105     122  EMSG:   .ASCIZ  /!ERROR!/         ;ERROR MESSAGE
       000217     122     117     122
       000222     041     000
    50                                 .EVEN
    51 000224     105     130     101  MSG:    .ASCIZ  /EXAMPLE #1/      ;I.D. MESSAGE
       000227     115     120     114
       000232     105     040     043
       000235     061     000
    52                                 .EVEN


EXAMPLE #1     MACRO X03.04 18-MAY-77 14:49:44 PAGE 5-1

    53                                 ;
    54                                 ; DISPLAY FILE FOR EXAMPLE #1
    55                                 ;
    56 000240  114000                  DFILE:  POINT
    57 000242  000100                          100
    58 000244  000500                          500
    59 000246  173520                          DNAME
    60 000250  000001                          1
    61 000252  103160                  D1:     CHAR!BLKOFF!INT4!LPON
    62 000254     117     116     105           .ASCII  /ONE./
       000257     056
    63 000260  114000                          POINT
    64 000262  000100                          100
    65 000264  000300                          300
    66 000266  173520                          DNAME
    67 000270  000002                          2
    68 000272  103160                  D2:     CHAR!BLKOFF!INT4!LPON
    69 000274     124     127     117           .ASCII  /TWO./
       000277     056
    70 000300  114000                          POINT
    71 000302  000100                          100
    72 000304  000100                          100
    73 000306  173520                          DNAME
    74 000310  000003                          3
    75 000312  103160                  D3:     CHAR!BLKOFF!INT4!LPON
    76 000314     124     110     122           .ASCII  /THREE./
       000317     105     105     056
    77 000322  173400                          DRET
```

```
    78 000324  000000                          0
    79         000000'                         .END    START


EXAMPLE #1      MACRO X03.04 18-MAY-77 14:49:44 PAGE 5-2
SYMBOL TABLE

D2      000272R         INTX   = 040000         INT4   = 003000         ITAL1 = 000060         DFILE   000240R
EXIT    000142R         INT2   = 002400         LPON   = 000140         LTST    000062R        INT7   = 003600
INT0  = 002000          LINE1  = 000005         MINUSX= 020000          INT5   = 003200        MAXY   = 001377
MAXSX = 017600          DJMP   = 160000         MINUSY= 020000          I2      000176R        SHORTV= 104000
D3      000312R         LONGV  = 110000         POINT  = 114000         IPTR    000206R        D1      000252R
MAXSY = 000077          LPDARK= 000300          EMSG    000214R         DSTAT  = 173424        STATSA= 170000
MISVY = 000100          LINE2  = 000006         ITAL0  = 000040         SCRUF   000210R        STATSB= 174000
MSG     000224R         INT3   = 002600         I1      0001/4R         SYNC   = 000004        $VSCRL= ****** G
INT1  = 002200          RELATV= 130000          $VLPEN= ****** G        INT6   = 003400        GRAPHX= 120000
BLKON = 000030          DRET   = 173400         HLKOFF= 000020          JSW    = 000044        GRAPHY= 120000
DHALT = 173500          LINE3  = 000007         DJSR   = 173400         MAYX   = 001777        $VNSRT= ****** G
LINE0 = 100004          LBUF    000156R         $VRTLK= ****** G        START   000000R        LPOFF  = 000100
CHAR  = 100000          INCR   = 000100         DNAME  = 173520         $VINLK= ****** G        MISVX  = 020000
DTABL   000200R         LPLITE= 000200          DNOP   = 164000

. ABS.  000000      000
        000326      001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  3564 WORDS  ( 14 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  64 PAGES
.LP:=VTMAC.MANEX1


EXAMPLE #2      MACRO X03.04 18-MAY-77 14:49:57 PAGE 5

     1                                          .TITLE  EXAMPLE #2
     2                                  ;
     3                                  ; THIS EXAMPLE USES THE TRACKING OBJECT AND THE TRACK
     4                                  ; COMPLETION ROUTINE TO CAUSE A VECTOR TO FOLLOW
     5                                  ; THE LIGHT PEN FROM A SET POINT AT (500,500).
     6                                  ;
     7          000000                          R0=%0
     8          000001                          R1=%1
     9          000006                          SP=%6
    10          000007                          PC=%7
    11                                          .MCALL  .EXIT,.TTYIN,.PRINT
    12 000000                           START:  .LNKRT                  ;LINK TO MONITOR
    13 000004  100004                           BPL     1$              ;LINK UP ERROR?
```

```
14 000006                                .PRINT  #EMSG           ;YES, INFORM USER
15 000014                                .EXIT                   ;AND EXIT
16 000016                          1$:   .INSRT  #DFILE          ;INSERT DISPLAY FILE
17 000026                                .TRACK  #IBUF,#TCOM     ;DISPLAY TRACK OBJECT
18 000042  004767  000006             JSR     PC,WAIT         ;WAIT FOR <CR>
19 000046                                .UNLNK                  ;UNLINK FROM MONITOR
20 000052                                .EXIT
21 000054                          WAIT: .TTYIN                  ;GET CHAR. FROM TTY
22 000060  022700  000012             CMP     #12,R0          ;LINE FEED?
23 000064  001373                     BNE     WAIT            ;NO, GET ANOTHER
24 000066  000207                     RTS     PC
25 000070  000500  000500          TBUF: .WORD   500,500         ;TRACK BUFFER INITED TO
26                                                               ;START TRACK AT (500,500)
27                                 ;
28                                 ; TRACK COMPLETION ROUTINE ENTERED AT INTERRUPT LEVEL
29                                 ; FROM DISPLAY FILE HANDLER WITH DISPLAY STOPPED.
30                                 ; USED TO UPDATE DISPLAY FILE WITH DATA FROM TBUF.
31                                 ;
32 000074  010146                  TCOM:   MOV     R1,-(SP)        ;SAVE R1
33 000076  016701  177766              MOV     TBUF,R1         ;NEW X
34 000102  166701  000052              SUB     OX,R1           ;NEW X - OLD X
35 000106  100003                      BPL     1$              ;POSITIVE DIFFERENCE?
36 000110  005401                      NEG     R1              ;NO, SO MAKE POSITIVE
37 000112  052701  020000              BIS     #MINUSX,R1      ;BUT SET MINUS BIT
38 000116  052701  040000          1$: BIS     #INTX,R1        ;ALSO SET INTENSIFY BIT
39 000122  010167  000040              MOV     R1,DX           ;THEN STORE IN DFILE.
40 000126  016701  177740              MOV     TBUF+2,R1       ;NEW Y
41 000132  166701  000024              SUB     OY,R1           ;NEW Y - OLD Y
42 000136  100003                      BPL     2$              ;POSITIVE DIFFERENCE?
43 000140  005401                      NEG     R1              ;NO, SO MAKE POSITIVE
44 000142  052701  020000              BIS     #MINUSX,R1      ;AND SET MINUS BIT
45 000146  010167  000016          2$: MOV     R1,DY           ;THEN STORE IN DFILE
46 000152  012601                      MOV     (SP)+,R1        ;RESTORE R1
47 000154  000207                      RTS     PC              ;EXIT FROM COMPLETION ROUTINE
48                                 ;
49                                 ; DISPLAY FILE FOR EXAMPLE #2
50                                 ;
51 000156  114000                  DFILE:  POINT                   ;SET POINT AT
52 000160  000500                  OX:     500
53 000162  000500                  OY:     500                     ;(500,500)
54 000164  113000                          LONGV!INT4              ;DRAW A VECTOR
55 000166  000000                  DX:     .WORD   0               ;INITIALLY NOWHERE
56 000170  000000                  DY:     .WORD   0
57 000172  173400                          DRET                    ;DISPLAY FILE END
```

```
EXAMPLE #2     MACRO X03.04 18-MAY-77 14:49:57 PAGE 5-1

                                                    0
     58 000174  000000                                  
     59 000176     123      117      122  EMSG:   .ASCIZ  /SORRY, THERE SEEMS TO BE A PROBLEM/
        000201     122      131      054
        000204     040      124      110
        000207     105      122      105
        000212     040      123      105
        000215     105      115      123
        000220     040      124      117
        000223     040      102      105
        000226     040      101      040
        000231     120      122      117
        000234     102      114      105
        000237     115      000
     60                                 .EVEN
     61          000000'                .END    START
```

```
EXAMPLE #2     MACRO X03.04 18-MAY-77 14:49:57 PAGE 5-2
SYMBOL TABLE

INT0  = 002000          LONGV = 110000          LPLITE= 000200          $VRTLK= ****** G        $VUNLK= ****** G
MAXSX = 017600          LPDARK= 000300          WAIT    000054R         DNAME = 173520          DFILE   000156R
MAXSY = 000077          LINE2 = 000006          $XTRAK= ****** G        DNOP  = 164000          INT7  = 003600
MISVY = 000100          DX      000166R         INT4  = 003000          ITAL1 = 000060          MAXY  = 001377
INT1  = 002200          INT3  = 002600          LPON  = 000140          INT5  = 003200          SHORTV= 104000
BLKON = 000030          RELATV= 130000          MINUSX= 020000          DSTAT = 173420          STATSA= 170000

DHALT = 173500          TCOM    000074R         MINUSY= 020000          SYNC  = 000004          STATSB= 174400
LINE0 = 000004          DRET  = 173400          POINT = 114000          INT6  = 003400          GRAPHX= 120000
CHAR  = 100000          LINE3 = 000007          EMSG    000176R         MAXX  = 001777          GRAPHY= 124000
INTX  = 040000          DY      000170R         ITAL0 = 000040          OX      000160R         $VNSRT= ****** G
INT2  = 002400          TBUF    000070R         BLKOFF= 000020          START   000000R         LPOFF = 000100
LINE1 = 000005          INCR  = 000100          DJSR  = 173400          OY      000162R         MISVX = 020000
DJMP  = 160000

. ABS.  000000     000
        000242     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  3717 WORDS  ( 15 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  64 PAGES
.LP:=VTMAC.MANEX2
```

Appendix B
System Macro Library

This appendix contains the listing of the system macro library (SYSMAC.SML) for Version 5 of RT–11. This library is stored on the system device. It is used by MACRO when expanding the programmed requests and calling the subroutines that are described in Chapter 2.

```
.MCALL .MODULE
.MODULE SYSMAC,RELEASE=Y05,VERSION=11,COMMENT=<System macro library>,LIB=YES
;
;                       COPYRIGHT (c) 1982, 1983 BY
;              DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
;                          ALL RIGHTS RESERVED
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
; COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
; OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
; TRANSFERRED.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
; CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.


;++
;       RESERVED SYMBOL NAMES
;       ---------------------
;
;       ...V1 and ...V5 are "global" values defined in macros
;
;       ...V1 -- controls .MCalling of ...CM* and support of V1, V2 and
;                 V3 versions of the expansions.
;       ...V5 -- controls generation of .Audit information.
;
;       ...V* -- are reserved for more local and global values
;
;       ...V2, ...V3, and ...V4 are in use currently (Y05.06) as
;                 local symbols (reusable in each macro definition).
;
;       RESERVED MACRO NAMES
;       --------------------
;
;       ..V1..
;       ..V2..
;       .MACS
;       ...CM*
;--
;++
;       Error Messages
;
;?SYSMAC-W-Invalid argument, use #0, not 0
;
```

```
;       General message -- Macro arguments of "0" are almost always
;       errors, (Which specify an address of 0, not a value of 0.)
;
;?SYSMAC-E-Odd or invalid vector specified
;
;       .DRBEG & .DRVTB -- the 2 lower bits of the vector address must
;       be 0.
;
;?SYSMAC-E-Invalid c o n t r o l value - control
;
;       .DRBOT -- an invalid controller type was specified. See definition
;       of .DRBOT for valid type codes.
;
;?SYSMAC-E-Invalid s i d e s value - sides
;
;       .DRBOT -- an invalid number of sides was specified, use 1 or 2.
;
;?SYSMAC-E-Primary boot too large
;
;       .DREND -- the primary boot code overflowed the area available to it.
;
;?SYSMAC-E-V A L Must not be 0, VAL;
;
;       .DRSET -- the second argument to the macro must not be 0, if it
;       is, the rest of the options in the table are ignored.
;
;?SYSMAC-E-Invalid parameter x;
;
;       .DRSET -- the last argument must be blank or one (or more) of
;       the following: NO,NUM,OCT
;--
;++
;       .,V1,,
;
;       .MCall the support routines and set the version to 1
;--

.MACRO .,V1,,
.MCALL  .,.,CM0,.,.,CM1,.,.,CM2,.,.,CM3,.,.,CM4,.,.,CM5,.,.,CM6,.,.,CM7
.,.,V1=1
.ENDM

;++
;       .,.,V2,,
;
;       .MCall the support routines and set the version to 2
;--

.MACRO .,V2,,
.MCALL  .,.,CM0,.,.,CM1,.,.,CM2,.,.,CM3,.,.,CM4,.,.,CM5,.,.,CM6,.,.,CM7
.,.,V1=2,
.ENDM

;++
;       .MACS
;
;       .MCall the support routines and set the version to [>=]3
;--

.MACRO .MACS
.MCALL  .,.,CM0,.,.,CM1,.,.,CM2,.,.,CM3,.,.,CM4,.,.,CM5,.,.,CM6,.,.,CM7
.,.,V1=3,
.ENDM
```

```
;++
;       ...CM0
;
;       Move a word to the stack. If argument blank or #0
;       Put a 0 on the stack. If second argument present,
;       generate an EMT with that value
;--

.MACRO ...CM0 STARG,INS
.IF B <STARG>
        CLR     -(SP)
.IFF
.IF IDN <STARG>,#0
        CLR     -(SP)
.IFF
.IIF IDN <STARG> <0> .ERROR ;?SYSMAC-W-Invalid argument, use #0, not 0;
        MOV     STARG,-(SP)
.ENDC
.ENDC
.IIF NB <INS>   EMT     ^o<INS>
.ENDM

;++
;       ...CM1
;
;       Setup R0 to point to AREA, set the CHAN and IC (subcode)
;       value in first word. This macro optimises number of
;       instructions to set up first word.
;       IC is forced to decimal.
;--

.MACRO ...CM1   AREA,IC,CHAN,FLAG,ARG,INS,CSET,BB
...CM5  <AREA>
...V2=0
.IF B <FLAG>
.IIF B <AREA>, ...V2=1
.IFF
.IIF DIF <FLAG>,SET, ...V2=1
.ENDC
.IF NE ...V2
.IF IDN <CHAN>,<#0>
        CLRB    @R0
.IFF
.IIF NB <CHAN>  MOVB    CHAN,@R0
.ENDC
.IFF
.IF B <CHAN>
        MOVB    #IC',,1(R0)
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^O27
        MOV     CHAN+<IC',*^O400>,@R0
.IFF
        MOV     #IC',*^O400,@R0
        MOVB    CHAN,@R0
.ENDC
.ENDC
.ENDC
.IIF IDN <CHAN> <0> .ERROR ;?SYSMAC-W-Invalid argument, use #0, not 0;
...CM2  <ARG>,2,INS,CSET,BB
.ENDM

;++
;       ...CM2
```

```
;
;       Move an argument value to OFFSET(R0),
;       Use a CLR_ if the value is #0,
;       Offset is forced to decimal,
;       BB is blank or B for byte operations
;       If INS NB generate an EMT 375
;--

.MACRO ...CM2    ARG,OFFSE,INS,CSET,BB
.IF B <ARG>
.IF NB <CSET>
.IIF NE ...V1-3, CLR'BB  OFFSE'.(R0)
.ENDC
.IFF
.IF IDN <ARG>,#0
        CLR'BB   OFFSE'.(R0)
.IFF
.IIF IDN <ARG> <0> .ERROR ;?SYSMAC-W-Invalid argument, use #0, not 0;
        MOV'BB   ARG,OFFSE'.(R0)
.ENDC
.ENDC
.IIF NB <INS>    EMT      ^0375
.ENDM

;++
;       ...CM3
;
;       Move a channel and code to R0,
;       Follow this with an EMT 374,
;       This macro optimises the instructions used
;       to load R0,
;--

.MACRO ...CM3    CHAN,IC
.IF B <CHAN>
        MOV      #IC*^0400,R0
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^027
        MOV      CHAN+<IC*^0400>,R0
.IFF
        MOV      #IC*^0400,R0
        BISB     CHAN,R0
.ENDC
.ENDC
        EMT      ^0374
.ENDM

;++
;       ...CM4
;
;       Setup for a SDAT/RCVD EMT block
;--

.MACRO ...CM4    AREA,BUF,WCNT,CRTN,IC,CODE
.IIF IDN <CODE>,NOSET ...CM1 <AREA>,IC,,<CODE>
.IIF DIF <CODE>,NOSET ...CM1 <AREA>,IC,#0,<CODE>
...CM2 <BUF>,4
...CM2 <WCNT>,6
...CM2 <CRTN>,8,E
.ENDM

;++
;       ...CM5
;
```

```
;       Move a (byte) value to R0 unless the src is
;       blank or R0. If so, then generate nothing.
;       BB is blank for word operations or B for byte.
;       If second argument present, generate an EMT with
;       that code value.
;--

.MACRO ...CM5    SRC,INS,BB
.IF NB <SRC>
.IIF DIF <SRC>,R0          MOV'BB  SRC,R0
.ENDC
.IIF NB <INS>    EMT       ^o<INS>
.ENDM

;++
;       ...CM6
;
;       Move a code and channel to R0. This macro
;       optimises the instructions needed to load R0.
;       Do the first ...CM2 also.
;       IC and CHAN are forced to decimal.
;--

.MACRO ...CM6    AREA,IC,CHAN,FLAG,ARG,INS,CSET,BB
...CM5  <AREA>
.IF B <FLAG>
.IIF NB <AREA>   MOV       #IC'.*^O400+CHAN'.,@R0
.IFF
.IIF IDN <FLAG>,SET        MOV    #IC'.*^O400+CHAN'.,@R0
.ENDC
...CM2  <ARG>,2,INS,CSET,BB
.ENDM

;++
;       ...CM7
;
;       Generate READ_/WRIT_ requests
;--

.MACRO ...CM7    AREA,CHAN,BUF,WCNT,BLK,CRTN,IC,CODE,V1
.IF EQ ...V1-1
...CM5  <WCNT>
...CM0  <CRTN>
...CM0  <BUF>
...CM0  <CHAN>,<V1+AREA>
.IFF
...CM1 <AREA>,IC,<CHAN>,<CODE>,<BLK>
...CM2 <BUF>,4
...CM2 <WCNT>,6
...CM2 <CRTN>,8,E
.ENDC
.ENDM

.MACRO .ABTIO    CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM3  <CHAN>,11.
.ENDM

;++
;       .AUDIT
;
;       macro to generate list of versions starting at abs 110
```

```
;
;       up to 26 names
;
;       First reference generates a Rad50 value for 110 of release
;--

.MACRO .AUDIT     q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m
.Save
.Asect
.IIf    NDF       ...V5     ...V5=^o110
.=...V5
.If     EQ        .-^o110
        .Globl    .Audit
.List
        .Word     .Audit
.NList
.EndC
.Irp    ...V2 <q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m>
.If     NB        <...V2>
        .Globl    '...V2'
.List
        .Word     '...V2'
.NList
.EndC
.EndR
...V5=.
.List
        .Word     -1
.NList
.Restore
.ENDM   .AUDIT

.MACRO .CDFN      AREA,ADDR,NUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,13,0,<CODE>,<ADDR>
...CM2  <NUM>,4,E
.ENDM

.MACRO  .CHAIN
         MOV      #^O4000,R0
         EMT      ^O374
.ENDM

.MACRO .CHCOP     AREA,CHAN,OCHAN,JOBBLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM1  <AREA>,11,<CHAN>,<CODE>,<OCHAN>
.IF NB <JOBBLK>
...CM2  <JOBBLK>,4,E
.IFF
...CM2  #0,4,E
.ENDC
.ENDM

.MACRO .CLOSE     CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
        EMT       ^O<160+CHAN>
```

```
.IFF
...CM3 <CHAN>,6,
.ENDC
.ENDM

.MACRO .CMKT      AREA,ID,TIME=#0,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,19,0,<CODE>,<ID>
...CM2  <TIME>,4,E,C
.ENDM

.MACRO .CNTXS     AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,27,0,<CODE>,<ADDR>,E
.ENDM

.MACRO .CRAW      AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30,2,<CODE>,<ADDR>,E
.ENDM

.MACRO .CRRG      AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30,0,<CODE>,<ADDR>,E
.ENDM

.MACRO .CSIGE     DEVSPC,DEFEXT,CSTRNG,LINBUF
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF NB <LINBUF>
...CMO  <LINBUF>
.NTYPE  ...V2,DEVSPC
.IF EQ ...V2-^O27
...CMO  <DEVSPC'+1>
.IFF
...CMO  <DEVSPC>
        INC     (SP)
.ENDC
.IFF
...CMO  <DEVSPC>
.ENDC
...CMO  <DEFEXT>
...CMO  <CSTRNG>,344
.ENDM

.MACRO .CSISP     OUTSPC,DEFEXT,CSTRNG,LINBUF
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF NB <LINBUF>
...CMO  <LINBUF>
```

```
.NTYPE  ...V2,OUTSPC
.IF EQ ...V2-^O27
...CM0  <OUTSPC'+1>
.IFF
...CM0  <OUTSPC>
        INC     (SP)
.ENDC
.IFF
...CM0  <OUTSPC>
.ENDC
...CM0  <DEFEXT>
...CM0  <CSTRNG>,345
.ENDM

.MACRO .CSTAT   AREA,CHAN,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM1  <AREA>,23,<CHAN>,<CODE>,<ADDR>,E
.ENDM

.MACRO .CTIMI   TBK
        JSR     R5,@$TIMIT
        .WORD   TBK-.
        .WORD   1
.ENDM

.MACRO .DATE
        MOV     #^O5000,R0
        EMT     ^O374
.ENDM

.MACRO .DELET   AREA,CHAN,DBLK,SEQNUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>,<AREA>
.IFF
...CM5  <AREA>
.IF IDN <CHAN>,#0
        CLR     @R0
.IFF
.IIF IDN <CHAN> <0> .ERROR ;?SYSMAC-W-Invalid argument, use #0, not 0;
...V2=0
.IF B <CODE>
.IIF B <AREA>, ...V2=1
.IFF
.IIF DIF <CODE>,SET, ...V2=1
.ENDC
.IF NE ...V2
.IIF NB <CHAN>  MOVB    CHAN,@R0
.IFF
.IF B <CHAN>
        CLRB    1(R0)
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^O27
        MOV     CHAN,@R0
.IFF
        CLR     @R0
        MOVB    CHAN,@R0
.ENDC
```

```
.ENDC
.ENDC
.ENDC
...CM2  <DBLK>,2
...CM2  <SEQNUM>,4,E,C
.ENDC
.ENDM

.MACRO .DEVIC    AREA,ADDR,LINK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF B LINK
...CM6  <AREA>,12,0,<CODE>,<ADDR>,E
.IFF
...CM6  <AREA>,12,1,<CODE>,<ADDR>,E
.ENDC
.ENDM

.MACRO .DRAST    NAME,PRI,ABT
.GLOBL  $INPTR
.IF B <ABT>
        RETURN
.IFF
        BR      ABT
.ENDC
NAME'INT:: JSR   R5,@$INPTR
        .WORD   ^C<PRI*^O40>&^O340
.ENDM

.MACRO .DRBEG    NAME,VEC,DSIZ,DSTS,VTBL
.ASECT
.=^O52
.GLOBL  NAME'END,NAME'INT
        .WORD   <NAME'END-NAME'STRT>
.IF B   <DSIZ>
        .WORD   NAME'DSIZE
.IFF
        .WORD   DSIZ
.ENDC
.IF B   <DSTS>
        .WORD   NAME'STS
.IFF
        .WORD   DSTS
.ENDC
        .WORD   ^O<ERL$G+<MMG$T*2>+<TIM$IT*4>+<RTE$M*10>>
.=^O176
.IIF DF NAME'$CSR, .WORD  NAME'$CSR
.PSECT  NAME'DVR
NAME'STRT::
.IF NB  VTBL
.GLOBL  VTBL
        .WORD   <VTBL-.>/2.-1+^O100000
.IFF
.IF NB  <VEC>
.IIF NE VEC&3. .ERROR     VEC ;?SYSMAC-E-Odd or invalid vector specified;
        .WORD   VEC&^C3.
.IFF
.IF DF  NAME'$VTB
.GLOBL  NAME'$VTB
        .WORD   <NAME'$VTB-.>/2.-1+^O100000
.IFF
.IIF NE NAME'$VEC&3. .ERROR     NAME'$VEC ;SYSMAC-E-Odd or invalid vector
                                ;specified;
```

```
        .WORD   NAME'$VEC&^C3.
.ENDC
.ENDC
.ENDC
        .WORD   NAME'INT-.,^O340
NAME'SYS::
NAME'LQE::      .WORD   0
NAME'CQE::      .WORD   0
.ENDM

;++
;       .DRBOT
;
;       CONTROL= is used to generate the controller description bits
;       in the boot block. The default value is correct for nearly
;       all RT supported devices. As many options may be specified as
;       are supported by the boot code:
;
;       <UBUS>    Unibus device
;       <QBUS>    Q-Bus device
;       <CBUS>    PC-Bus device
;       <UMSCP>   Unibus MSCP device
;       <QMSCP>   Q-Bus MSCP device
;       <CMSCP>   PC-Bus MSCP device
;
;       SIDES= is used to indicate the number of sides supported in
;       floppy disk drive. Valid values are 1 and 2. Hard media
;       sidedness is not coded.
;
;       NOTE: the definition of a code does NOT imply any present
;       or future product committment.
;--
.MACRO  .DRBOT  NAME,ENTRY,READ,CONTROL=<UBUS,QBUS>,SIDES=1
        .DREND  NAME
.IIF NDF TPS, TPS=^O177564
.IIF NDF TPB, TPB=^O177566
LF=^O12
CR=^O15
B$BOOT=^O1000
B$DEVN=^O4716
B$DEVU=^O4722
B$READ=^O4730
.IF EQ  MMG$T
B$DNAM=^R'NAME
.IFF
B$DNAM=^R'NAME'X
.ENDC
.ASECT
.=^O62
        .WORD   NAME'BOOT,NAME'BEND-NAME'BOOT,READ-NAME'BOOT
.PSECT  NAME'BOOT
NAME'BOOT::NOP
        BR      ENTRY-2.
...V2=^O100
.IRP    X       <CONTROL>
...V3=0
.IIF    IDN     <X>     <UBUS>  ...V3=1.
.IIF    IDN     <X>     <QBUS>  ...V3=2.
.IIF    IDN     <X>     <CBUS>  ...V3=4.
.IIF    IDN     <X>     <QMSCP> ...V3=^O10
.IIF    IDN     <X>     <UMSCP> ...V3=^O20
.IIF    IDN     <X>     <CMSCP> ...V3=^O40
.IIF    EQ      ...V3   .ERROR  ;?SYSMAC-E-Invalid c o n t r o l value -
                                ;control;
...V2=...V2!...V3
```

```
.ENDR
.=ENTRY-6.
        .BYTE     ^o20,...V2,^o20,^o^c<20+...V2+20>
.IF     EQ        <SIDES-1>
        BR        ENTRY
.IFF
.IF     EQ        <SIDES-2.>
        BMI       ENTRY
.IFF
        .ERROR ;?SYSMAC-E-Invalid s i d e s value - sides;
.ENDC
.ENDC
.ENDM

.MACRO .DRDEF     NAME,CODE,STAT,SIZE,CSR,VEC
.MCALL  .DRAST,.DRBEG,.DRBOT,.DREND,.DRFIN,.DRSET,.DRVTB,.FORK,.QELDF
.IIF NDF RTE$M RTE$M=0
.IIF NE RTE$M RTE$M=1
.IIF NDF TIM$IT TIM$IT=0
.IIF NE TIM$IT TIM$IT=1
.IIF NDF MMG$T MMG$T=0
.IIF NE MMG$T MMG$T=1
.IIF NDF ERL$G ERL$G=0
.IIF NE ERL$G ERL$G=1
.IIF NE TIM$IT .MCALL      .TIMIO,.CTIMI
.QELDF
HDERR$=1
EOF$=^o20000
VARSZ$=^o400
ABTIO$=^o1000
SPFUN$=^o2000
HNDLR$=^o4000
SPECL$=^o10000
WONLY$=^o20000
RONLY$=^o40000
FILST$=^o100000
NAME'DSIZ=SIZE
NAME'$COD=CODE
NAME'STS=<CODE>!<STAT>
.IIF NDF NAME'$CSR, NAME'$CSR=CSR
.IIF NDF NAME'$VEC, NAME'$VEC=VEC
.GLOBL  NAME'$CSR,NAME'$VEC
.ENDM

;++
;       .DREND
;
;       FORCE is used to force the generation of the vector table
;       assigning a value to FORCE causes the associated sysgen
;       bit value to "forced" on for purposes of generating the table.
;
;       FORCE=1 will force generation of the error logging vector
;       for instance.
;--
.MACRO .DREND     NAME,FORCE=0
.PSECT  NAME'DVR
.IIF NDF NAME'$END, NAME'$END:
.IF EQ  .-NAME'$END
NAME'$END::
.IF NE MMG$T!<FORCE&2.>
$RLPTR::.WORD     0
$MPPTR::.WORD     0
$GTBYT::.WORD     0
$PTBYT::.WORD     0
$PTWRD::.WORD     0
```

```
.ENDC
.IIF NE ERL$G!<FORCE&1> $ELPTR::.WORD   0
.IIF NE TIM$IT!<FORCE&4.> $TIMIT::.WORD 0
$INPTR::.WORD    0
$FKPTR::.WORD    0
.GLOBL  NAME'STRT
NAME'END==.
.IFF
.PSECT  NAME'BOOT

.IIF LT <NAME'BOOT-.+^o664>, .ERROR ;?SYSMAC-E-Primary boot too large;

.=NAME'BOOT+^o664
BIOERR: JSR       R1,REPORT
        .WORD     IOERR-NAME'BOOT

REPORT: MOV       #BOOTF-NAME'BOOT,R0
        JSR       R1,REP
        MOV       (R1)R0
        JSR       R1,REP
        MOV       #CRLFLF-NAME'BOOT,R0
        JSR       R1,REP
        RESET
        HALT
        BR        .-2.

REPOR:  MOVB      (R0)+,@#TPB
REP:    TSTB      @#TPS
        BPL       REP
        TSTB      @R0
        BNE       REPOR
        RTS       R1

BOOTF:  .ASCIZ    <CR><LF>"?BOOT-U-"<^o200>
CRLFLF: .ASCIZ    <CR><LF><LF>
IOERR:  .ASCIZ    "I/O error"
        .EVEN
NAME'BEND::
.ENDC
.ENDM

.MACRO  .DRFIN    NAME
.GLOBL  NAME'CQE
        MOV       PC,R4
        ADD       #NAME'CQE-.,R4
        MOV       @#^O54,R5
        JMP       @^O270(R5)
.ENDM

.MACRO  .DRSET    OPTION,VAL,RTN,MODE
.ASECT
.IF LE  .-^o400
.=^o400
.IFF
.=.-2.
.ENDC
...V2=0
.IRP X,<VAL>
.IIF EQ ...V2 .IIF EQ <X> .ERROR ;?SYSMAC-E-V A L Must not be 0, VAL;
...V2=...V2+1
.ENDR
        VAL
...V2=.
        .RAD50    \OPTION\
.=...V2+4.
        .BYTE     <RTN-^o400>/2.
```

```
.,.V2=0
.IRP X,<MODE>
.IF IDN <X>,<NUM>
.,.V2=.,.V2!^o100
.IFF
.IF IDN <X>,<NO>
.,.V2=.,.V2!^o200
.IFF
.IF IDN <X>,<OCT>
.,.V2=.,.V2!^o140
.IFF
.ERROR  ;?SYSMAC-E-Invalid parameter x;
.ENDC
.ENDC
.ENDC
.ENDR
        .BYTE     .,.V2
        .WORD     0
.ENDM

.MACRO .DRVTB     NAME,VEC,INT,PS=0
.IF NB  NAME
NAME'$VTB::
.IFF
.=.-2.
.ENDC
.IIF NE VEC&3,    .ERROR   VEC ;?SYSMAC-E-Odd or invalid vector specified;
        .WORD     VEC&^C3.,INT-.,^o340!PS,0
.ENDM

.MACRO .DSTAT     RETSPC,DNAM
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM5  <DNAM>
.,.CM0  <RETSPC>,342
.ENDM

.MACRO .ELAW      AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM6  <AREA>,30,3,<CODE>,<ADDR>,E
.ENDM

.MACRO .ELRG      AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM6  <AREA>,30,1,<CODE>,<ADDR>,E
.ENDM

.MACRO .ENTER     AREA,CHAN,DBLK,LEN,SEQNUM,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ .,.V1-1
.,.CM5  <CHAN>
.,.CM0  <DBLK>,<40+AREA>
.IFF
.,.CM1  <AREA>,2,<CHAN>,<CODE>,<DBLK>
.,.CM2  <LEN>,4,,C
```

```
.,.CM2  <SEQNUM>,6,E,C
.ENDC
.ENDM

.MACRO .EXIT
        EMT       ^O350
.ENDM

.MACRO .FETCH     ADDR,DNAM
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM5  <DNAM>
.,.CM0  <ADDR>,343
.ENDM

.MACRO .FORK      FKBLK
        JSR       R5,@$FKPTR
        .WORD     FKBLK - .
.ENDM

.MACRO .FPROT     AREA,CHAN,DBLK,PROT=#1,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM1  <AREA>,35,<CHAN>,<CODE>,<DBLK>
.,.CM2  <PROT>,4,E,,B
.ENDM

.MACRO .GMCX      AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM6  <AREA>,30,6,<CODE>,<ADDR>,E
.ENDM

.MACRO .GTIM      AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM6  <AREA>,17,0,<CODE>,<ADDR>,E
.ENDM

.MACRO .GTJB      AREA,ADDR,JOBBLK,CODE
.IF NDF .,.V1
.MCALL .MACS
.MACS
.ENDC
.,.CM6  <AREA>,16,1,<CODE>,<ADDR>
.IF NB <JOBBLK>
.IF IDN <JOBBLK>,<ME>
.,.CM2  #-1,4,E
.IFF
.,.CM2  <JOBBLK>,4,E
.ENDC
.IFF
.,.CM2  #-3.,4,E
.ENDC
.ENDM

.MACRO .GTLIN     LINBUF,PROMPT,TYPE
```

```
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM0  <LINBUF>
.IF     NB      <TYPE>
...CM0  #3.
.IFF
...CM0  #1
.ENDC
...CM0  <PROMPT>
...CM0  ,345
.ENDM

.MACRO .GVAL    AREA,OFFSE,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,28,0,<CODE>,<OFFSE>,E
.ENDM

.MACRO .HERR
        MOV     #^O2400,R0
        EMT     ^O374
.ENDM

.MACRO .HRESE
        EMT     ^O357
.ENDM

.MACRO .INTEN   PRIO,PIC
.IF B PIC
        JSR     R5,@^O54
.IFF
        MOV     @#^O54,-(SP)
        JSR     R5,@(SP)+
.ENDC
        .WORD   ^C<PRIO*32.>&224.
.ENDM

.MACRO .LOCK
        EMT     ^O346
.ENDM

.MACRO .LOOKU   AREA,CHAN,DBLK,SEQNUM,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
.IF EQ ...V1-1
...CM5  <CHAN>,<20+AREA>
.IFF
...CM1  <AREA>,1,<CHAN>,<CODE>,<DBLK>
...CM2  <SEQNUM>,4,E,C
.ENDC
.ENDM

.MACRO .MAP     AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,30,4,<CODE>,<ADDR>,E
.ENDM
```

```
.MACRO .MFPS     ADDR
       MOV       @#^O54,-(SP)
       ADD       #^O362,(SP)
       CALL      @(SP)+
.IIF NB <ADDR>   MOVB      (SP)+,ADDR
.ENDM

;++
;       .MODULE
;
;       Macro to define a standard identification for all
;       modules.
;
;       Inputs:
;
;       Module 1-5 character symbol name         (KEDIO)
;       Release 3 char release identification    (X05)
;       Version 2 char version number            (09)
;       Comment n char title string              <I/O Module>
;
;       TITLE=YES generate .Title                (default)
;       TITLE=NO do not generate .Title
;
;       IDENT=YES generate .Ident                (default)
;       IDENT=NO do not generate .Ident
;
;       AUDIT=NO generate .Audit call            (default)
;       AUDIT=YES generate .Audit call
;
;       LIB=NO  generate .Audit global value     (default)
;       LIB=YES do not generate .Audit
;
;       GLOBAL    not specified                  (default)
;       GLOBAL=gname substitutes gname for .'Module'
;
;       Outputs:
;
;       .Title  'Module' - 'Comment'             title for module
;       .Ident  "'Release','Version'"            ident for module
;       .'Module' ==:   'Version',               version value symbol Binary
;       .Audit  ==:   ^r'Release'                release value symbol Rad50
;
;       .MCall .Audit                            get .Audit definition
;       .Audit .Audit  .'Module'                 generate audit information
;
;
;       definition of .NLCSI macro               generate program ID string
;
;       .NLCSI    TYPE=,PART=
;       TYPE=Z generate .AsciZ (default)         .AsciZ "KEDIO X05.09 "
;       TYPE=I generate .AsciI                   .AsciI "KEDIO X05.09 "
;
;       PART=ALL  generate std ID (default)      .Asciz "KEDIO X05.09 "
;       PART=NAME generate name                  .Asciz "KEDIO"
;       PART=RLSVER generate release & version   .Asciz "X05.09"
;       PART=PREFIX generate message prefix      .Asciz "?KEDIO-"
;
;       definition of .RModule macro             generate Rad50 for 'Module'
;
;       .RModule
;--

.MACRO  .MODULE Module          Release Version Comment
TITLE=YES,IDENT=YES,AUDIT=NO,GLOBAL,LIB=NO
.MCall  .Audit
```

```
.IIf     IDN     <Title> <YES>    .Title 'Module'  - 'Comment'
.IIf     IDN     <Ident> <YES>    .Ident "'Release','Version'"
.IIf     IDN     <Lib> <NO>       .Audit==:^r'Release'
.If      NB      <GLOBAL>
'Global'==:'Version',
.IIf     IDN     <Audit> <YES>    .Audit  'Global'
.IfF
.'Module'==:'Version',
.IIf     IDN     <Audit> <YES>    .Audit  .'Module'
.EndC
.Macro   .NLCSI TYPE=Z,PART=ALL
.IIf     IDN     <PART> <ALL>     .Asci'Type'    "'Module' 'Release','Version' "
.IIf     IDN     <PART> <NAME>    .Asci'Type'    "'Module' "
.IIf     IDN     <PART> <RLSVER>  .Asci'Type'    "'Release','Version' "
.IIf     IDN     <PART> <PREFIX>  .Asci'Type'    "?'Module'-"
.EndM
.Macro   .RModule
         .Rad50 "'Module'"
.EndM
...V5=^o110
.ENDM

.MACRO .MRKT    AREA,TIME,CRTN,ID,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,18,0,<CODE>,<TIME>
...CM2  <CRTN>,4
...CM2  <ID>,6,E
.ENDM

.MACRO .MTATC    AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,5,<CODE>,<ADDR>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTDTC    AREA,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,6,<CODE>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTGET    AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,1,<CODE>,<ADDR>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTIN     AREA,ADDR,UNIT,CHRCNT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,2,<CODE>,<ADDR>
```

```
...CM2  <UNIT>,4,,,B
...CM2  <CHRCNT>,5,E,,B
.ENDM

.MACRO .MTOUT   AREA,ADDR,UNIT,CHRCNT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,3,<CODE>,<ADDR>
...CM2  <UNIT>,4,,,B
...CM2  <CHRCNT>,5,E,,B
.ENDM

.MACRO .MTPRN   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,7,<CODE>,<ADDR>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTPS    ADDR
.IIF NB <ADDR>  CLR     -(SP)
.IIF NB <ADDR>  MOVB    ADDR,(SP)
        MOV     @#^O54,-(SP)
        ADD     #^O360,(SP)
        CALL    @(SP)+
.ENDM

.MACRO .MTRCT   AREA,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,4,<CODE>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTSET   AREA,ADDR,UNIT,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,0,<CODE>,<ADDR>
...CM2  <UNIT>,4,E,,B
.ENDM

.MACRO .MTSTA   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,31,8,<CODE>,<ADDR>
...CM2  #0,4,E
.ENDM

.MACRO .MWAIT
        MOV     #^O4400,R0
        EMT     ^O374
.ENDM

.MACRO  .PEEK   AREA,ADDR,CODE
.IF NDF ...V1
```

```
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,28,1,<CODE>,<ADDR>,E
.ENDM

.MACRO  .POKE   AREA,ADDR,VALUE,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,28,3,<CODE>,<ADDR>
...CM2  <VALUE>,4,E
.ENDM

.MACRO .PRINT   ADDR
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <ADDR>,351
.ENDM

.MACRO .PROTE   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,25,0,<CODE>,<ADDR>,E
.ENDM

.MACRO .PURGE   CHAN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM3  <CHAN>,3,
.ENDM

.MACRO  .PVAL   AREA,OFFSE,VALUE,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM6  <AREA>,28,2,<CODE>,<OFFSE>
...CM2  <VALUE>,4,E
.ENDM

.MACRO .QELDF
.IIF NDF MMG$T, MMG$T=1
.IIF NE MMG$T, MMG$T=1
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
.IRP    X,<LINK,CSW,BLKN,FUNC,JNUM,UNIT,BUFF,WCNT,COMP>
Q$'X=Q.'X-4
.ENDR
.IF EQ MMG$T
Q.ELGH=^O16
```

```
.IFF
Q.PAR=^O16
Q$PAR=^O12
Q.ELGH=^O24
.ENDC
.ENDM

.MACRO .QSET     ADDR,LEN
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM5  <LEN>
...CM0  <ADDR>,353
.ENDM

.MACRO  .RCTRL
        EMT     ^O355
.ENDM

.MACRO  .RCVD   AREA,BUF,WCNT,CRTN=#1,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM4  <AREA>,<BUF>,<WCNT>,<CRTN>,22,<CODE>
.ENDM

.MACRO  .RCVDC  AREA,BUF,WCNT,CRTN,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM4  <AREA>,<BUF>,<WCNT>,<CRTN>,22,<CODE>
.ENDM

.MACRO  .RCVDW  AREA,BUF,WCNT,CRTN=#0,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM4  <AREA>,<BUF>,<WCNT>,<CRTN>,22,<CODE>
.ENDM

.MACRO  .RDBBK  RGSIZ
.MCALL  .RDBDF
.RDBDF
        .WORD
        .WORD   RGSIZ
        .WORD
.ENDM

.MACRO  .RDBDF
R.GID=0
R.GSIZ=2.
R.GSTS=4.
R.GLGH=6.
RS.CRR=^O100000
RS.UNM=^O40000
RS.NAL=^O20000
.ENDM

.MACRO  .READ   AREA,CHAN,BUF,WCNT,BLK,CRTN=#1,CODE
.IF NDF ...V1
.MCALL  .MACS
```

```
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8,<CODE>,200
.ENDM

.MACRO  .READC  AREA,CHAN,BUF,WCNT,CRTN,BLK,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8,<CODE>,200
.ENDM

.MACRO  .READW  AREA,CHAN,BUF,WCNT,BLK,CRTN=#0,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,8,<CODE>,200
.ENDM

.MACRO  .REGDEF
.ENDM

.MACRO  .RELEA  DNAM
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM5  <DNAM>
...CM0  ,343
.ENDM

.MACRO  .RENAM  AREA,CHAN,DBLK,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
.IF EQ  ...V1-1
...CM5  <CHAN>,<100+AREA>
.IFF
...CM1  <AREA>,4,<CHAN>,<CODE>,<DBLK>,E
.ENDC
.ENDM

.MACRO  .REOPE  AREA,CHAN,CBLK,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
.IF EQ  ...V1-1
...CM5  <CHAN>,<140+AREA>
.IFF
...CM1  <AREA>,6,<CHAN>,<CODE>,<CBLK>,E
.ENDC
.ENDM

.MACRO  .RSUM
        MOV     #^O1000,R0
        EMT     ^O374
.ENDM

.MACRO  .SAVES  AREA,CHAN,CBLK,CODE
.IF NDF ...V1
.MCALL  .MACS
```

```
.MACS
.ENDC
.IF EQ  .,.V1-1
...CM5  <CHAN>,<120+AREA>
.IFF
...CM1  <AREA>,5,<CHAN>,<CODE>,<CBLK>,E
.ENDC
.ENDM

.MACRO  .SCCA   AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,29,0,<CODE>,<ADDR>,E
.ENDM

.MACRO  .SDAT   AREA,BUF,WCNT,CRTN=#1,CODE
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM4 <AREA>,<BUF>,<WCNT>,<CRTN>,21,<CODE>
.ENDM

.MACRO  .SDATC  AREA,BUF,WCNT,CRTN,CODE
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM4 <AREA>,<BUF>,<WCNT>,<CRTN>,21,<CODE>
.ENDM

.MACRO  .SDATW  AREA,BUF,WCNT,CRTN=#0,CODE
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM4 <AREA>,<BUF>,<WCNT>,<CRTN>,21,<CODE>
.ENDM

.MACRO  .SDTTM  AREA,ADDR,CODE
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,32,0,<CODE>,<ADDR>,E
.ENDM

.MACRO  .SERR
        MOV     #^O2000,R0
        EMT     ^O374
.ENDM

.MACRO  .SETTO  ADDR
.IF NDF .,.V1
.MCALL  .MACS
.MACS
.ENDC
...CM5  <ADDR>,354
.ENDM

.MACRO  .SFDAT  AREA,CHAN,DBLK,DATE=#0,CODE
.IF NDF .,.V1
```

```
.MCALL  .MACS
.MACS
.ENDC
...CM1  <AREA>,34,<CHAN>,<CODE>,<DBLK>
...CM2  <DATE>,4,E
.ENDM

.MACRO  .SFPA   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,24,0,<CODE>,<ADDR>,E
.ENDM

.MACRO  .SPCPS  AREA,ADDR,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM6  <AREA>,33,0,<CODE>,<ADDR>,E
.ENDM

.MACRO  .SPFUN  AREA,CHAN,FUNC,BUF,WCNT,BLK,CRTN=#0,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM1  <AREA>,26,<CHAN>,<CODE>,<BLK>
...CM2  <BUF>,4
...CM2  <WCNT>,6
.IF NB FUNC
.NTYPE  ...V2,FUNC
.IF NE  ...V2-^O27
.IIF DIF <CODE>,NOSET,...CM2       #^O377,8,,,B
...CM2  <FUNC>,9,,,B
.IFF
...CM2  <FUNC'*^O4000377>,8
.ENDC
.ENDC
...CM2  <CRTN>,10,E,C
.ENDM

.MACRO  .SPND
        MOV     #^O400,R0
        EMT     ^O374
.ENDM

.MACRO  .SRESE
        EMT     ^O352
.ENDM

.MACRO  .SYNCH  AREA,PIC
.IF B PIC
.IIF NB <AREA>  MOV     AREA,R4
.IFF
.IF NB AREA
        MOV     PC,R4
        ADD     #AREA-.,R4
.ENDC
.ENDC
        MOV     @#^O54,R5
        JSR     R5,@^O324(R5)
.ENDM
```

```
.MACRO  .TIMIO  TBK,HI,LO
        JSR     R5,@$TIMIT
        .WORD   TBK-.
        .WORD   0
        .WORD   HI
        .WORD   LO
.ENDM

.MACRO .TLOCK
        MOV     #^O3400,R0
        EMT     ^O374
.ENDM

.MACRO .TRPSE   AREA,ADDR,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CMG  <AREA>,3,0,<CODE>,<ADDR>,E
.ENDM

.MACRO .TTINR
        EMT     ^O340
.ENDM

.MACRO .TTOUT
        EMT     ^O341
.ENDM

.MACRO .TTYIN   CHAR
        EMT     ^O340
        BCS     .-2.
.IF NB <CHAR>
.IIF DIF <CHAR>,R0      MOVB    R0,CHAR
.ENDC
.ENDM

.MACRO  .TTYOU  CHAR
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CM5  <CHAR>,341,B
        BCS     .-2.
.ENDM

.MACRO  .TWAIT  AREA,TIME,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CMG  <AREA>,20,0,<CODE>,<TIME>,E
.ENDM

.MACRO  .UNLOC
        EMT     ^O347
.ENDM

.MACRO  .UNMAP  AREA,ADDR,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CMG  <AREA>,30,5,<CODE>,<ADDR>,E
.ENDM
```

```
.MACRO  .UNPRO  AREA,ADDR,CODE
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
...CMG  <AREA>,25,1,<CODE>,<ADDR>,E
.ENDM

.MACRO  .WAIT   CHAN
.IF NDF ...V1
.MCALL  .MACS
.MACS
.ENDC
.IF EQ  ...V1-1
        EMT     ^O<240+CHAN>
.IFF
.IF B   <CHAN>
        CLR     R0
.IFF
.NTYPE ...V2,CHAN
.IF EQ ...V2-^O27
.IF IDN <CHAN>,#0
        CLR     R0
.IFF
.IIF IDN <CHAN> <0> .ERROR ;?SYSMAC-W-Invalid argument, use #0, not 0;
        MOV     CHAN,R0
.ENDC
.IFF
        CLR     R0
        BISB    CHAN,R0
.ENDC
.ENDC
        EMT     ^O374
.ENDC
.ENDM

.MACRO .WDBBK   WNAPR,WNSIZ,WNRID,WNOFF,WNLEN,WNSTS
.MCALL  .WDBDF
.WDBDF
        .BYTE
        .BYTE   WNAPR
        .WORD
        .WORD   WNSIZ
        .WORD   WNRID
        .WORD   WNOFF
        .WORD   WNLEN
        .WORD   WNSTS
.ENDM

.MACRO  .WDBDF
W.NID=0
W.NAPR=1
W.NBAS=2.
W.NSIZ=4.
W.NRID=6.
W.NOFF=^O10
W.NLEN=^O12
W.NSTS=^O14
W.NLGH=^O16
WS.CRW=^O100000
WS.UNM=^O40000
WS.ELW=^O20000
WS.MAP=^O400
.ENDM
```

```
.MACRO   .WRITC  AREA,CHAN,BUF,WCNT,CRTN,BLK,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9,<CODE>,220
.ENDM

.MACRO   .WRITE  AREA,CHAN,BUF,WCNT,BLK,CRTN=#1,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9,<CODE>,220
.ENDM

.MACRO   .WRITW  AREA,CHAN,BUF,WCNT,BLK,CRTN=#0,CODE
.IF NDF ...V1
.MCALL .MACS
.MACS
.ENDC
...CM7  <AREA>,<CHAN>,<BUF>,<WCNT>,<BLK>,<CRTN>,9,<CODE>,220
.ENDM
```

INDEX

ABTIO$
  defined by .DRDEF, 2–33
.ABTIO programmed request, 2–2
  summary, 1–32
Addressing modes
  description, 1–10
AJFLT system subroutine, 3–1
  summary, 1–65
  using, 1–57

Blank arguments
  description, 1–9
.BLANK graphics macro, A–3

C.CSW
  returned by .SAVESTATUS, 2–111
C.DEVQ
  returned by .SAVESTATUS, 2–111
C.LENG
  returned by .SAVESTATUS, 2–111
C.SBLK
  returned by .SAVESTATUS, 2–111
C.UNIT
  returned by .SAVESTATUS, 2–111
C.USED
  returned by .SAVESTATUS, 2–111
.CDFN programmed request, 2–3
  effect of .EXIT, 2–44
  effect on .GTJB, 2–54
  relationship to .CHAIN, 2–5
  relationship to .SRESET, 2–132
  restrictions, 1–26
  summary, 1–32
  using, 1–16
.CHAIN programmed request, 2–4
  summary, 1–32
  using, 1–24
CHAIN system subroutine, 3–1
  summary, 1–64
Channel allocation
  using .CDFN, 1–16
Channel numbers
  description, 1–11
  system subroutine library, 1–39
Channel status word
  See CSW
Character strings
  allocating in FORTRAN, 1–59
  passing to subprograms, 1–60
  quoted literals, 1–61
  support in SYSLIB, 1–57
.CHCOPY programmed request, 2–6
  summary, 1–36
  using, 1–23
  Version 4, 1–29
.CLEAR graphics macro, A–4
CLOSEC system subroutine, 3–2
  relationship to ICSI, 3–16
  relationshp to IENTER, 3–21
  summary, 1–61
  USR requirements, 1–42
CLOSE keyboard command
  after .EXIT, 1–24
  relationship to .EXIT, 2–43

.CLOSE programmed request, 2–8
not done by .CSISPC, 2–21
on a protected file, 2–49
relationship to .CHCOPY, 2–6
relationship to .ENTER, 2–42
relationship to .LOOKUP, 2–65
relationship to .PURGE, 2–90
relationship to .SERR, 2–59
requires device handler, 2–45
summary, 1–32
using, 1–19, 1–24
.CMKT programmed request, 2–9
relationship to .MRKT, 2–71
summary, 1–32
using, 1–24
.CNTXSW programmed request, 2–10
restrictions, 1–26
summary, 1–36
using, 1–16
Command String Interpreter
See CSI
Completion routines
introduction, 1–21
restrictions, 1–22, 1–40, 2–2
system subroutine library, 1–39
CONCAT system subroutine, 3–4
summary, 1–58, 1–66
.CRAW programmed request, 2–12
relationship to .GMCX, 2–50
summary, 1–36
using, 1–26
.CRRG programmed request, 2–15
summary, 1–36
using, 1–26
CSI
implicit .UNLOCK, 2–63
introduction, 1–18
options, 2–19
using, 2–16
.CSIGEN programmed request, 2–16
compared to .GTLIN, 2–55
implicit .UNLOCK, 2–63
relationship to .LOOKUP, 2–65
summary, 1–32
using, 1–18
.CSISPC programmed request, 2–21
compared to .GTLIN, 2–55
implicit .UNLOCK, 2–63
relationship to .SETTOP and USR, 2–119
summary, 1–32
using, 1–19
CSI special mode
See .CSISPC
.CSTAT programmed request, 2–24
summary, 1–32
Version 5, 1–29
.CSTATUS programmed request
using, 1–18
CSW
bits defined by .DRDEF, 2–34
.CTIMIO macro, 2–25
expansion, 2–26
relationship to .DRDEF, 2–33
summary, 1–32
CTRL/C
disabling, 2–112
CTRL/O
reset by .RCTRLO, 2–95
CVTTIM system subroutine, 3–5
instead of .GTIM, 2–52
summary, 1–64
using, 1–56

Date
.GTIM required for date rollover, 2–52
internal format, 2–26
month and year rollover, 2–26
set by .SDTTM, 2–117
.DATE programmed request, 2–26
summary, 1–32
using, 1–18
DATE subroutine (in FORLIB)
using, 1–56
.DELETE programmed request, 2–27
on a protected file, 2–49
requires device handler, 2–45
summary, 1–32
using, 1–19
Device blocks
description, 1–12
with system subroutine library, 1–41
Device handler macros
.CTIMIO, 2–25
.DRAST, 2–30
.DRBEG, 2–32
.DRBOT, 2–32
.DREND, 2–34
.DRFIN, 2–35
.DRSET, 2–35
.DRVTB, 2–36
.FORK, 2–47
.INTEN, 2–61
.SYNCH, 2–132
.TIMIO, 2–135
Device handlers
writing, 1–27

Device identification codes
list of values, 2–37
.DEVICE programmed request, 2–28
summary, 1–36
using, 1–17
Device status word
contents, 2–37
defined by .DRDEF, 2–33
DEVICE system subroutine, 3–5
relationship to INTSET, 3–29
summary, 1–64
DHALT display halt instruction, A–14
Display file handler
assembling graphics programs, A–16
assembly instructions, A–24
description, A–1
example, A–27
linking, A–16
linking graphics programs, A–16
subroutine summary, A–21
using, A–15
Display file structure, A–17
BASIC–11 graphics software, A–20
subroutine calls, A–18
Display processor mnemonics, A–23
DJFLT system subroutine, 3–6
summary, 1–65
using, 1–57
DJSR subroutine call instruction, A–13
DNAME load name register instruction, A–14
.DRAST macro, 2–30
relationship to .DRDEF, 2–33
relationship to .FORK, 2–48
summary, 1–32
using, 1–27
.DRBEG macro, 2–32
relationship to .DRDEF, 2–33
relationship to .DRVTB, 2–36
relationship to .FORK, 2–48
summary, 1–32
using, 1–27
.DRBOT macro, 2–32
relationship to .DRDEF, 2–33
summary, 1–32
using, 1–27
.DRDEF macro, 2–33
summary, 1–33
use before .DRBEG, 2–32
using, 1–27
.DREND macro, 2–34
called by .DRBOT, 2–32
relationship to .DRDEF, 2–33
relationship to .DRVTB, 2–36
relationship to .FORK, 2–48
summary, 1–33
using, 1–27
DRET subroutine return instruction, A–13
.DRFIN macro, 2–35
relationship to .DRDEF, 2–33
relationship to .FORK, 2–48
summary, 1–33
using, 1–27
.DRSET macro, 2–35
relationship to .DRDEF, 2–33
summary, 1–33
using, 1–27
.DRVTB macro, 2–36
relationship to .DRDEF, 2–33
summary, 1–33
using, 1–27
DSTAT display status instruction, A–14
.DSTATUS programmed request, 2–36
relationship to .SETTOP and USR, 2–119
summary, 1–33
using, 1–18

.ELAW programmed request, 2–39
relationship to .CRAW, 2–13
summary, 1–36
using, 1–26
$ELPTR
defined by .DREND, 2–34
.ELRG, 2–15
.ELRG programmed request, 2–40
summary, 1–36
using, 1–26
EMT codes
See also Programmed requests
EMT 374, 1–7
EMT 375, 1–8
meaning of different values, 1–3
EMT instructions
See Programmed requests
.ENTER programmed request, 2–40
done by .CSIGEN, 2–16
not done by .CSISPC, 2–21
on a protected file, 2–49
relationship to .CHCOPY, 2–7
relationship to .CLOSE, 2–8
relationship to .CSTAT, 2–24
relationship to .READx, 2–100
relationship to .SAVESTATUS, 2–110
relationship to .SERR, 2–59
relationship to .SRESET, 2–131
relationship to .WRITx, 2–149

requires device handler, 2–45
summary, 1–33
using, 1–19
EOF$
defined by .DRDEF, 2–34
ERL$G
defined by .DRDEF, 2–33
$ERLOG pointer
in handler termination table, 2–34
Error processing
monitor errors, 1–17
Errors
intercepting monitor errors, 2–137
programmed requests, 1–12
.EXIT programmed request, 2–43
relationship to .DEVICE, 2–28
summary, 1–33
using, 1–24
Extended memory monitor
See XM monitor

FB monitor
foreground job and .FETCH, 2–45
introduction, 1–2
.FETCH programmed request, 2–45
done by .CSIGEN, 2–16
fills in $FKPTR, 2–48
not done by .CSISPC, 2–21
relationship to $INPTR, 2–30
relationship to .ENTER, 2–42
relationship to .SETTOP and USR, 2–119
relationship to .SRESET, 2–131
relationship to handler termination table, 2–35
summary, 1–33
Version 5, 1–29
File operations
introduction, 1–19
FILST$
defined by .DRDEF, 2–33
$FKPTR
defined by .DREND, 2–34
setup by user program, 2–49
Foreground/background
communications, 1–23
context switch, 1–23
with FORTRAN programs, 1–52
Foreground/background monitor
See FB monitor
.FORK macro, 2–47
relationship to .DRDEF, 2–33
summary, 1–33
$FORK pointer
in handler termination table, 2–34
FORLIB.OBJ
linking, 1–55
FORTRAN logical units
relationship to .CHAIN, 2–5
FORTRAN programs
calculating workspace, 1–53
.FPROT programmed request, 2–49
relationship to .RENAME, 2–108
requires device handler, 2–45
summary, 1–33
using, 1–20

General mode
See .CSIGEN
$GETBYT pointer
in handler termination table, 2–34
GETSTR system subroutine, 3–7
summary, 1–58, 1–66
USR requirements, 1–42
.GMCX programmed request, 2–50
summary, 1–37
using, 1–26
Graphics macro calls
summary, A–21
$GTBYT
defined by .DREND, 2–34
.GTIM programmed request, 2–51
summary, 1–33
using, 1–18
GTIM system subroutine, 3–7
summary, 1–64
.GTJB programmed request, 2–53
summary, 1–33
using, 1–18
Version 4, 1–29
GTJB system subroutine, 3–8
summary, 1–64
.GTLIN programmed request, 2–55
implicit .UNLOCK, 2–63
relationship to .SETTOP and USR, 2–119
summary, 1–33
using, 1–19
Version 5, 1–29
GTLIN system subroutine, 3–9
summary, 1–62
USR requirements, 1–42
GT OFF keyboard command, A–2
GT ON keyboard command, A–2
.GVAL programmed request, 2–57
compared with .PEEK, 2–86
summary, 1–33

HDERR$
  defined by .DRDEF, 2–34
.HERR programmed request, 2–58
  summary, 1–33
  using, 1–17
HNDLR$
  defined by .DRDEF, 2–33
.HRESET programmed request, 2–61
  relationship to .CDFN, 2–3
  relationship to .LOOKUP, 2–65
  relationship to .PURGE, 2–90
  relationship to .QSET, 2–93
  summary, 1–33
  using, 1–24

I/O operations
  introduction, 1–20
IABTIO system subroutine, 3–10
  summary, 1–62
IADDR system subroutine, 3–10
  summary, 1–67
IAJFLT system subroutine, 3–11
  summary, 1–65
  using, 1–57
IASIGN system subroutine, 3–11
  summary, 1–63
ICDFN system subroutine, 3–13
  summary, 1–63
  USR requirements, 1–42
ICHCPY system subroutine, 3–14
  summary, 1–63
ICLOSE system subroutine, 3–2
  relationship to IENTER, 3–21
  summary, 1–61
  USR requirements, 1–42
ICMKT system subroutine, 3–15
  cancelling an ITIMER request, 3–56
  cancelling ISCHED requests, 3–44
  summary, 1–64
ICSI system subroutine, 3–16
  summary, 1–63
  using argument from IFETCH, 3–23
  using with IASIGN, 3–11
  USR requirements, 1–42
ICSTAT system subroutine, 3–18
  summary, 1–63
IDATE subroutine (in FORLIB)
  using, 1–56
IDELET system subroutine, 3–18
  summary, 1–61
  USR requirements, 1–42
IDJFLT system subroutine, 3–20
  summary, 1–65
  using, 1–57
IDSTAT system subroutine, 3–20
  summary, 1–64
  USR requirements, 1–42
IENTER system subroutine, 3–21
  relationship to CLOSE, 3–3
  relationship to ICSI, 3–16
  summary, 1–61
  USR requirements, 1–42
IFETCH system subroutine, 3–23
  relationship to ICSI, 3–16
  relationship to IDELET, 3–19
  summary, 1–65
  USR requirements, 1–42
IFPROT system subroutine, 3–23
  summary, 1–61
IFREEC system subroutine, 3–24
  summary, 1–63
IGETC system subroutine, 3–24
  summary, 1–63
IGETSP system subroutine, 3–25
  summary, 1–67
IGTJB system subroutine, 3–8
  summary, 1–64
IJCVT system subroutine, 3–26
  summary, 1–65
  using, 1–57
ILUN system subroutine, 3–26
  summary, 1–63
INDEX system subroutine, 3–27
  summary, 1–58, 1–66
Indirect command files
  relationship to .CSIGEN, 2–17
$INPTR
  defined by .DREND, 2–34
  referenced by .DRAST, 2–30
Input/output operations
  See I/O operations
INSERT system subroutine, 3–27
  summary, 1–58, 1–66
.INSRT graphics macro, A–5
INTEGER*4 support
  in SYSLIB, 1–56
.INTEN macro, 2–61
  must precede .FORK, 2–48
  relationship to .SPND/.RSUM, 2–130
  summary, 1–33
  using, 1–26
$INTEN monitor routine
  referenced by .DRAST, 2–30
$INTEN pointer
  in handler termination table, 2–34
Interrupt service routines, 1–26
  use of .SYNCH, 2–132
INTSET system subroutine, 3–28
  summary, 1–67

IPEEKB system subroutine, 3–30
  restrictions, 1–45
  summary, 1–67
IPEEK system subroutine, 3–30
  restrictions, 1–45
  summary, 1–67
  using with IGETSP, 3–25
IPOKEB system subroutine, 3–31
  restrictions, 1–45
  summary, 1–67
IPOKE system subroutine, 3–31
  restrictions, 1–45
  summary, 1–67
  using with IGETSP, 3–25
IPUT system subroutine, 3–32
  summary, 1–67
IQSET system subroutine, 3–32
  summary, 1–65
  using, 1–45
  USR requirements, 1–42
IRAD50 system subroutine, 3–33
  summary, 1–67
  using, 1–57
IRCVDC system subroutine, 3–33
  requires queue element, 1–45
  summary, 1–62
IRCVDF system subroutine, 3–33
  requires queue element, 1–45
  summary, 1–62
IRCVD system subroutine, 3–33
  requires queue element, 1–45
  summary, 1–62
IRCVDW system subroutine, 3–33
  requires queue element, 1–45
  summary, 1–62
IREADC system subroutine, 3–35
  requires queue element, 1–45
  summary, 1–62
IREADF system subroutine, 3–35
  requires queue element, 1–45
  summary, 1–62
IREAD system subroutine, 3–35
  requires queue element, 1–45
  summary, 1–62
IREADW system subroutine, 3–35
  requires queue element, 1–45
  summary, 1–62
IRENAM system subroutine, 3–40
  summary, 1–61
  USR requirements, 1–42
IREOPN system subroutine, 3–41
  summary, 1–63
ISAVES system subroutine, 3–42
  relationship to IREOPN, 3–41
  summary, 1–63
ISCHED system subroutine, 3–43
  cancelled by ICMKT, 3–15
  requires queue element, 1–45
  summary, 1–64
ISCOMP system subroutine, 3–89
  summary, 1–66
ISDATC system subroutine, 3–44
  requires queue element, 1–45
  summary, 1–62
ISDATF system subroutine, 3–44
  requires queue element, 1–45
  summary, 1–62
ISDAT system subroutine, 3–44
  requires queue element, 1–45
  summary, 1–62
ISDATW system subroutine, 3–44
  requires queue element, 1–45
  summary, 1–62
ISDTTM system subroutine, 3–47
  summary, 1–64
ISFDAT system subroutine, 3–47
  summary, 1–61
ISLEEP system subroutine, 3–48
  requires queue element, 1–45
  summary, 1–64
  using, 1–56
ISPFNC system subroutine, 3–49
  requires queue element, 1–45
  summary, 1–65
ISPFNF system subroutine, 3–49
  requires queue element, 1–45
  summary, 1–65
ISPFN system subroutine, 3–49
  requires queue element, 1–45
  summary, 1–65
ISPFNW system subroutine, 3–49
  requires queue element, 1–45
  summary, 1–65
ISPY system subroutine, 3–55
  restrictions, 1–45
  summary, 1–67
ISR
  See Interrupt service routines
ITIMER system subroutine, 3–55
  cancelled by ICMKT, 3–15
  requires queue element, 1–45
  rescheduling FORTRAN subroutines, 3–44
  summary, 1–64
ITLOCK system subroutine, 3–57
  summary, 1–65
  using, 1–44
  USR requirements, 1–42

ITTINR system subroutine, 3–57
multiterminal equivalent, 3–79
summary, 1–62
ITTOUR system subroutine, 3–59
multiterminal equivalent, 3–80
summary, 1–62
ITWAIT system subroutine, 3–59
relationship to SUSPND/RESUME, 3–93
requires queue element, 1–45
summary, 1–64
using, 1–56
IUNTIL system subroutine, 3–60
requires queue element, 1–45
summary, 1–64
using, 1–56
IVERIF system subroutine
See VERIFY system subroutine
summary, 1–66
IWAIT system subroutine, 3–61
summary, 1–62
use with ISPFN, 3–49
IWRITC system subroutine, 3–61
requires queue element, 1–45
summary, 1–62
IWRITE system subroutine, 3–61
requires queue element, 1–45
summary, 1–62
IWRITF system subroutine, 3–61
requires queue element, 1–45
summary, 1–62
IWRITW system subroutine, 3–61
requires queue element, 1–45
summary, 1–62

JADD system subroutine, 3–64
summary, 1–65
JAFIX system subroutine, 3–65
summary, 1–65
using, 1–57
JCMP system subroutine, 3–65
summary, 1–65
JDFIX system subroutine, 3–66
summary, 1–65
using, 1–57
JDIV system subroutine, 3–66
summary, 1–66
JICVT system subroutine, 3–67
summary, 1–66
using, 1–57
JJCVT system subroutine, 3–68
summary, 1–66
JMOV system subroutine, 3–68
summary, 1–66
JMUL system subroutine, 3–69
summary, 1–66
Job status word
See JSW
JSUB system subroutine, 3–69
summary, 1–66
JSW
bit 11, 2–44
bit 12, 1–23
effect on terminal input, 2–140
relationship to ITTINR, 3–57
bit 14, 2–55
effect on terminal input, 2–140
relationship to ITTINR, 3–58
bit 3, 2–55
bit 4, 2–140
bit 5, 2–44
bit 6
compared with M.TSTS bit 6, 2–79
relationship to .TTINR, 2–139
relationship to .TTOUTR, 2–141
relationship to ITTINR, 3–57
relationship to ITTOUR, 3–59
bit 8, 2–5
issue .MTRCTO or .RCTRLO after changing, 2–95
JTIME system subroutine, 3–70
summary, 1–64
using, 1–56

Keyword macro arguments
description, 1–11

LEN system subroutine, 3–70
summary, 1–58, 1–66
.LNKRT graphics macro, A–5
LOAD keyboard command
before running foreground job, 2–45
fills in $FKPTR, 2–48
relationship to .SRESET, 2–131
relationship to handler termination table, 2–35
relationship to IDELET, 3–19
.LOCK programmed request, 2–62
compared to .TLOCK, 2–136
effect of .EXIT, 2–44
relationship to .CSIGEN, 2–21
summary, 1–33
using, 1–16
LOCK system subroutine, 3–71
summary, 1–65
USR requirements, 1–42
Logical job names, 2–67
assigning, 1–25

.LOOKUP programmed request, 2–65
done by .CSIGEN, 2–16
not done by .CSISPC, 2–21
on a protected file, 2–49
relationship to .CLOSE, 2–8
relationship to .CSTAT, 2–24
relationship to .ENTER, 2–42
relationship to .READx, 2–100
relationship to .REOPEN, 2–109
relationship to .SAVESTATUS, 2–110
relationship to .SERR, 2–59
relationship to .WRITx, 2–149
requires device handler, 2–45
summary, 1–33
system job, 2–67
using, 1–19
Version 4, 1–29
LOOKUP system subroutine, 3–72
relationship to CLOSE, 3–3
relationship to ICSI, 3–16
summary, 1–61
USR requirements, 1–42
.LPEN graphics macro, A–7

M.FCNT
contents, 2–77
M.TFIL
contents, 2–77
M.TST2
contents, 2–78
in multiterminal status block, 2–77
M.TSTS
bit 12
relationship to .MTIN, 2–79
bit 6
relationship to .MTIN, 2–79
relationship to .MTOUT, 2–80
contents, 2–77
M.TSTW
contents, 2–78
in multiterminal status block, 2–77
M.TWID
contents, 2–77
.MAP programmed request, 2–68
summary, 1–37
using, 1–26
MAXJOB
in timer block, 2–25
.MCALL directive
use, 1–6
Memory allocation
swapping USR, 1–15
with .SETTOP, 1–15
Message handler
See MQ handler
.MFPS programmed request, 2–69
summary, 1–34
using, 1–18
MMG$T
defined by .DRDEF, 2–33
Monitor fixed offset area
introduction, 1–3
Monitor services
introduction, 1–1
$MPPHY pointer
in handler termination table, 2–34
$MPPTR
defined by .DREND, 2–34
MQ handler
relationship to system job .LOOKUP, 2–67
using, 1–25
.MRKT programmed request, 2–71
relationship to .CMKT, 2–9
requires queue element, 2–93
summary, 1–34
using, 1–24
MRKT system subroutine, 3–75
cancelled by ICMKT, 3–15
requires queue element, 1–45
summary, 1–64
.MTATCH programmed request, 2–73
relationship to .MTGET, 2–77
summary, 1–34
using, 1–23
MTATCH system subroutine, 3–76
summary, 1–63
.MTDTCH programmed request, 2–75
summary, 1–34
using, 1–23
MTDTCH system subroutine, 3–78
summary, 1–63
.MTGET programmed request, 2–76
relationship to .MTATCH, 2–73
summary, 1–34
using, 1–18, 1–23
MTGET system subroutine, 3–79
summary, 1–63
.MTIN programmed request, 2–79
summary, 1–34
using, 1–23
MTIN system subroutine, 3–79
summary, 1–63
.MTOUT programmed request, 2–80
summary, 1–34
using, 1–23
MTOUT system subroutine, 3–80
summary, 1–63

.MTPRNT programmed request, 2–81
summary, 1–34
using, 1–23
MTPRNT system subroutine, 3–81
summary, 1–63
.MTPS programmed request, 2–69
summary, 1–34
using, 1–18
.MTRCTO programmed request, 2–82
summary, 1–34
MTRCTO system subroutine, 3–81
summary, 1–63
.MTSET programmed request, 2–83
summary, 1–34
using, 1–23
MTSET system subroutine, 3–81
summary, 1–63
.MTSTAT programmed request, 2–84
summary, 1–34
using, 1–18, 1–23
MTSTAT system subroutine, 3–82
summary, 1–63
Multiterminal feature
introduction, 1–2
Multiterminal requests
introduction, 1–23
Multiterminal status block
contents, 2–77
contents after .MTSET, 2–83
.MWAIT programmed request, 2–85
relationship to .RCVDx, 2–96
relationship to system job .LOOKUP, 2–67
summary, 1–37
using, 1–23
MWAIT system subroutine, 3–83
requires queue element, 1–45
summary, 1–63

.NAME graphics macro, A–8
.NOSYN graphics macro, A–11

.PEEK programmed request, 2–86
summary, 1–34
.POKE programmed request, 2–86
summary, 1–34
.PRINT programmed request, 2–87
multiterminal equivalent, 2–81
summary, 1–34
using, 1–22
PRINT system subroutine, 3–83
summary, 1–63
Processor status word
See PSW
Programmed requests
See also EMT codes
addressing modes, 1–10
blank arguments, 1–9
channel numbers, 1–11
conversion to Version 5, 1–29
device blocks, 1–12
errors, 1–12
execution, 1–4
extended memory, 1–25
format, 1–6
introduction, 1–1, 1–3
keyword macro arguments, 1–11
registers available, 1–11
summary, 1–32
using, 1–15
USR requirements, 1–13
Version 1, 1–27
Version 2, 1–28
Version 3, 1–28
Version 4, 1–29
Version 5, 1–29
.PROTECT programmed request, 2–88
summary, 1–37
using, 1–17
PSW
referenced by .MFPS/.MTPS, 2–69
$PTBYT
defined by .DREND, 2–34
$PTWRD
defined by .DREND, 2–34
.PURGE programmed request, 2–90
relationship to .CHCOPY, 2–6
relationship to .LOOKUP, 2–65
relationship to .SERR, 2–59
summary, 1–34
using, 1–19
PURGE system subroutine, 3–84
in place of CLOSE, 3–3
$PUTBYT pointer
in handler termination table, 2–34
PUTSTR system subroutine, 3–84
summary, 1–58, 1–66
USR requirements, 1–42
$PUTWRD pointer
in handler termination table, 2–34
.PVAL programmed request, 2–57
compared with .POKE, 2–86
summary, 1–34
to change default .ENTER size, 2–41

Q$BLKN
defined by .QELDF, 2–92

Q$BUFF
  defined by .QELDF, 2–92
Q$COMP
  defined by .QELDF, 2–92
Q$CSW
  defined by .QELDF, 2–92
Q$FUNC
  defined by .QELDF, 2–92
Q$JNUM
  defined by .QELDF, 2–92
Q$LINK
  defined by .QELDF, 2–92
Q$PAR
  defined by .QELDF, 2–92
Q$UNIT
  defined by .QELDF, 2–92
Q$WCNT
  defined by .QELDF, 2–92
Q.BLKN
  defined by .QELDF, 2–92
Q.BUFF
  defined by .QELDF, 2–92
Q.COMP
  defined by .QELDF, 2–92
  relationship to .SYNCH, 2–134
Q.CSW
  defined by .QELDF, 2–92
Q.ELGH
  defined by .QELDF, 2–92
Q.FUNC
  defined by .QELDF, 2–92
Q.JNUM
  defined by .QELDF, 2–92
Q.LINK
  defined by .QELDF, 2–92
Q.PAR
  defined by .QELDF, 2–92
Q.UNIT
  defined by .QELDF, 2–92
Q.WCNT
  defined by .QELDF, 2–92
.QELDF macro, 2–92
  relationship to .DRDEF, 2–33
  relationship to .FORK, 2–48
  summary, 1–34
.QSET programmed request, 2–92
  effect of .EXIT, 2–44
  relationship to .RCVDx, 2–96
  relationship to .READx, 2–100
  relationship to .SRESET, 2–132
  relationship to .TWAIT, 2–143
  relationship to .WRITx, 2–148
  restrictions, 1–26
  summary, 1–34
  using, 1–16
R.GID
  defined by .RDBDF, 2–100
R.GLGH
  defined by .RDBDF, 2–100
R.GSIZ
  defined by .RDBDF, 2–100
R.GSTS
  defined by .RDBDF, 2–100
R50ASC system subroutine, 3–85
  summary, 1–67
  using, 1–57
RAD50 system subroutine, 3–85
  summary, 1–67
  using, 1–57
Radix–50 support
  in SYSLIB, 1–57
RCHAIN system subroutine, 3–86
  relationship to CHAIN, 3–2
  summary, 1–65
.RCTRLO programmed request, 2–95
  multiterminal equivalent, 2–82
  summary, 1–34
  using, 1–22
RCTRLO system subroutine, 3–86
  multiterminal equivalent, 3–81
  summary, 1–65
.RCVDC programmed request, 2–95
  relationship to .SDATx, 2–113
  requires queue element, 2–93
  summary, 1–37
  using, 1–23
.RCVD programmed request, 2–95
  relationship to .SDATx, 2–113
  relationship to system job .LOOKUP, 2–67
  requires queue element, 2–93
  summary, 1–37
  use with .MWAIT, 2–85
  using, 1–23, 1–25
.RCVDW programmed request, 2–95
  relationship to .SDATx, 2–113
  requires queue element, 2–93
  summary, 1–37
.RDBBK macro, 2–99
  summary, 1–37
  using, 1–26
.RDBDF macro, 2–99
  summary, 1–37
  using, 1–26
.READC programmed request, 2–100
  for messages between jobs, 2–67
  requires device handler, 2–45
  requires queue element, 2–93
  summary, 1–35
  using, 1–21

.READ programmed request, 2–100
for messages between jobs, 2–67
relationship to .SAVESTATUS, 2–110
relationsip to .CHCOPY, 2–6
requires device handler, 2–45
requires queue element, 2–93
summary, 1–34
use with .WAIT, 2–145
using, 1–20
.READW programmed request, 2–100
for messages between jobs, 2–67
requires device handler, 2–45
requires queue element, 2–93
summary, 1–35
using, 1–20
REENTER keyboard command
after .EXIT, 1–24
relationship to .EXIT, 2–43
.RELEASE programmed request, 2–45
after .FETCH, 2–46
sometimes ignored, 2–46
summary, 1–35
$RELOC pointer
in handler termination table, 2–34
.REMOV graphics macro, A–9
.RENAME programmed request, 2–107
on a protected file, 2–49
requires device handler, 2–45
summary, 1–35
using, 1–19
.REOPEN programmed request, 2–109
summary, 1–35
using, 1–19
REPEAT system subroutine, 3–87
summary, 1–58, 1–66
.RESTR graphics macro, A–9
RESUME system subroutine, 3–87
relationship to SUSPND, 3–93
summary, 1–65
$RLPTR
defined by .DREND, 2–34
RONLY$
defined by .DRDEF, 2–33
RS.CRR
defined by .RDBDF, 2–100
RS.NAL
defined by .RDBDF, 2–100
RS.UNM
defined by .RDBDF, 2–100
.RSUM programmed request, 2–130
effect of .TWAIT, 2–144
relationship to .SRESET, 2–132
summary, 1–37
.SAVESTATUS programmed request, 2–110
relationship to .ENTER, 2–42
relationship to .LOOKUP, 2–65
relationship to .PURGE, 2–90
relationship to .REOPEN, 2–109
summary, 1–35
using, 1–19
.SCCA programmed request, 2–112
summary, 1–35
SCCA system subroutine, 3–88
summary, 1–65
SCOMP system subroutine, 3–89
summary, 1–58, 1–66
SCOPY system subroutine, 3–89
summary, 1–58, 1–66
.SCROL graphics macro, A–9
.SDATC programmed request, 2–113
relationship to .RCVDx, 2–96
requires queue element, 2–93
summary, 1–37
using, 1–23
.SDAT programmed request, 2–113
relationship to .RCVDx, 2–95
relationship to system job .LOOKUP, 2–67
requires queue element, 2–93
summary, 1–37
use with .MWAIT, 2–85
using, 1–23, 1–25
.SDATW programmed request, 2–113
relationship to .RCVDx, 2–96
requires queue element, 2–93
summary, 1–37
.SDTTM programmed request, 2–117
summary, 1–35
using, 1–18
SECNDS system subroutine, 3–90
instead of .GTIM, 2–52
summary, 1–64
.SERR programmed request, 2–58
relationship to .DELETE, 2–27
relationship to .ENTER, 2–42
summary, 1–35
using, 1–17
SETCMD system subroutine, 3–91
summary, 1–65
SET keyboard commands
relationship to .DRSET, 2–35
SET EXIT NOSWAP
relationship to .EXIT, 2–43
relationship to .SETTOP and USR, 2–120

SET EXIT SWAP
relationship to .EXIT, 2–43
SET TT QUIET
relationship to .GTLIN, 2–55
SET USR NOSWAP
relationship to .SETTOP, 2–119
relationship to LOCK/UNLOCK, 3–72
SET option table
defined by .DRSET, 2–35
.SETTOP programmed request, 2–119
in XM monitor, 2–121
restrictions, 1–26
summary, 1–35
using, 1–15
.SFDAT programmed request, 2–122
relationship to .RENAME, 2–108
requires device handler, 2–45
summary, 1–35
using, 1–20
.SFPA programmed request, 2–123
relationship to .CNTXSW, 2–10
summary, 1–35
using, 1–17
Single-job monitor
See SJ monitor
Single-line editor
relationship to .TTYIN, 2–140
SJ monitor
introduction, 1–2
.SPCPS programmed request, 2–124
summary, 1–37
using, 1–22
Special (single-character) mode for terminal, 1–23
SPECL$
defined by .DRDEF, 2–33
SPFUN$
defined by .DRDEF, 2–33
.SPFUN programmed request, 2–126
bit in device status word, 2–38
function codes, 2–127
requires device handler, 2–45
summary, 1–35
using, 1–17
.SPND programmed request, 2–130
effect of .TWAIT, 2–144
relationship to .SRESET, 2–132
summary, 1–37
using, 1–24
.SRESET programmed request, 2–131
performed by .HRESET, 2–61
relationship to .CDFN, 2–3
relationship to .LOOKUP, 2–65
relationship to .PURGE, 2–90
relationship to .QSET, 2–93
summary, 1–35
using, 1–24
Stack pointer
caution with .CHAIN, 2–5
during .EXIT, 2–44
.START graphics macro, A–10
START keyboard command
after .EXIT, 1–24
relationship to .EXIT, 2–43
.STAT graphics macro, A–10
.STOP graphics macro, A–11
STRPAD system subroutine, 3–91
summary, 1–58, 1–66
SUBSTR system subroutine, 3–92
summary, 1–58, 1–66
Suspension of a program, 1–24
SUSPND system subroutine, 3–93
relationship to RESUME, 3–87
summary, 1–65
.SYNC graphics macro, A–11
.SYNCH macro, 2–132
relationship to .SPND/.RSUM, 2–130
summary, 1–35
using, 1–21, 1–26
SYSCOM area
introduction, 1–3
SYSLIB.OBJ
additional services, 1–56
introduction, 1–1
SYSLIB functions, 3–1
channel, 1–63
character string, 1–57, 1–66
data transfer, 1–62
date and time, 1–56
device and file, 1–63
file-oriented, 1–61
INTEGER*4, 1–41, 1–56, 1–65
miscellaneous, 1–67
program suspension, 1–56
Radix–50, 1–57, 1–67
RT–11 services, 1–64
summary, 1–61
timer, 1–64
SYSLOW, 2–121
SYSMAC.MAC
listing, B–1
SYSMAC.SML
description, 1–6
introduction, 1–1
macros for device handlers, 1–27
macros for interrupt service routines, 1–26

System communication area
See SYSCOM area
System jobs
communicating with, 1–25
.LOOKUP, 2–67
System job support
introduction, 1–2
System macro library
introduction, 1–1
listing, B–1
System status
how to get, 1–18
System subroutine library
calling conventions, 1–46
capabilities, 1–38
channel numbers, 1–39
completion routines, 1–39
restrictions, 1–40
conventions, 1–38
device blocks, 1–41
FORTRAN/MACRO interface, 1–47
introduction, 1–1
subroutine argument block, 1–47
subroutine register usage, 1–48
system restrictions, 1–45
using, 1–37

Terminal I/O
introduction, 1–22
special mode, 1–23
Termination of a program, 1–24
TIM$IT
defined by .DRDEF, 2–33
TIMASC system subroutine, 3–94
instead of .GTIM, 2–52
summary, 1–64
using, 1–56
Time
internal format, 2–118
set by .SDTTM, 2–118
TIME keyboard command
relationship to GTIM system subroutine, 3–7
Timer block format, 2–25
Timer support
introduction, 1–24
TIME system subroutine, 3–95
instead of .GTIM, 2–52
summary, 1–64
.TIMIO macro, 2–135
relationship to .CTIMIO, 2–25
relationship to .DRDEF, 2–33
summary, 1–35
timer block format, 2–25
$TIMIO pointer
in handler termination table, 2–34
$TIMIT
defined by .DREND, 2–34
.TLOCK programmed request, 2–136
summary, 1–35
using, 1–16
.TRACK graphics macro, A–11
TRANSL system subroutine, 3–95
summary, 1–58, 1–66
TRIM system subroutine, 3–97
summary, 1–58, 1–66
.TRPSET programmed request, 2–137
summary, 1–35
using, 1–17
.TTINR programmed request, 2–139
summary, 1–36
using, 1–22
with indirect command file, 2–56
.TTOUTR programmed request, 2–141
summary, 1–36
using, 1–22
when to use .PRINT, 2–88
.TTYIN programmed request, 2–139
multiterminal equivalent, 2–79
summary, 1–36
using, 1–22
with indirect command file, 2–56
.TTYOUT programmed request, 2–141
multiterminal equivalent, 2–80
summary, 1–36
using, 1–22
when to use .PRINT, 2–88
.TWAIT programmed request, 2–143
relationship to .CMKT, 2–10
relationship to .SPND/.RSUM, 2–130
requires queue element, 2–93
summary, 1–36
using, 1–24
Version 5, 1–29

.UNLNK graphics macro, A–12
.UNLOCK programmed request, 2–62
implicit by .CSIGEN, 2–16, 2–21
implicit by .CSISPC, 2–22
performed by .EXIT, 2–44
summary, 1–36
using, 1–16
UNLOCK system subroutine, 3–97
relationship to LOCK, 3–71
summary, 1–65
.UNMAP programmed request, 2–145
relationship to .MAP, 2–68
summary, 1–37
using, 1–26

.UNPROTECT programmed request, 2–88
summary, 1–37
using, 1–17
User service routine
See USR
USR
ownership by different jobs, 2–136
USR locking
effect of .LOCK, 2–62
effect of UNLOCK system subroutine, 3–97
how to minimize, 1–44
relationship to .CSIGEN, 2–16
using LOCK system subroutine, 3–71
USR requirements
.CLOSE, 2–8
FORTRAN interface, 1–42
programmed requests, 1–13
swapping, 1–14
USR swapping
controlling, 1–15, 1–43, 2–110
effect of .LOCK, 2–62
strategies, 1–43

..V1.. macro
summary, 1–36
..V2.. macro
summary, 1–36
VARSZ$
defined by .DRDEF, 2–33
VERIFY system subroutine, 3–98
summary, 1–58, 1–66
Version 1
differences, 1–27
Version 2
differences, 1–28
Version 3
differences, 1–28
Version 4
differences, 1–29
Version 5
differences, 1–29
VTBASE.OBJ display file handler module, A–2, A–15
VTCAL1.OBJ display file handler module, A–2, A–15
VTCAL2.OBJ display file handler module, A–2, A–15
VTCAL3.OBJ display file handler module, A–2, A–15
VTCAL4.OBJ display file handler module, A–2, A–15
VTHDLR.OBJ concatenated display modules, A–2, A–16
VTLIB.OBJ display file handler library, A–2, A–15
components, A–16
linking, A–16
VTMAC.MAC
listing, A–25
VTMAC.MAC display file handler macros, A–2, A–15

W.NAPR
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
use with .CRAW, 2–12
W.NBAS
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
W.NID
defined by .WDBDF, 2–148
W.NLEN
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
W.NLGH
defined by .WDBDF, 2–148
W.NOFF
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
W.NRID
defined by .WDBDF, 2–148
W.NSIZ
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
W.NSTS
defined by .WDBDF, 2–148
modified by .GMCX, 2–51
use with .CRAW, 2–12
W.RID
modified by .GMCX, 2–51
.WAIT programmed request, 2–145
compared with .MWAIT, 2–85
summary, 1–36
use with .CSIGEN, 2–16
use with .READx, 2–101
using, 1–20
.WDBBK macro, 2–147
summary, 1–37
using, 1–26
.WDBDF macro, 2–148
automatically called by .WDBBK, 2–147
summary, 1–37
using, 1–26

WONLY$
  defined by .DRDEF, 2–33
.WRITC programmed request, 2–148
  for messages between jobs, 2–67
  relationship to .CSTAT, 2–24
  requires device handler, 2–45
  requires queue element, 2–93
  summary, 1–36
  using, 1–21
.WRITE programmed request, 2–148
  for messages between jobs, 2–67
  relationship to .CHCOPY, 2–6
  relationship to .CSTAT, 2–24
  relationship to .SAVESTATUS, 2–110
  requires device handler, 2–45
  requires queue element, 2–93
  summary, 1–36
  to a protected file, 2–49
  use with .WAIT, 2–145
  using, 1–20
.WRITW programmed request, 2–148
  for messages between jobs, 2–67
  relationship to .CSTAT, 2–24
  requires device handler, 2–45
  requires queue element, 2–93
  summary, 1–36
  using, 1–20
WS.CRW
  defined by .WDBDF, 2–148
  use with .CRAW, 2–12
WS.ELW
  defined by .WDBDF, 2–148
  use with .CRAW, 2–12
WS.MAP
  defined by .WDBDF, 2–148
  effect on .CRAW, 2–12, 2–13
  optional argument to .WDBBK, 2–147
WS.UNM
  defined by .WDBDF, 2–148
WS.VNM
  use with .CRAW, 2–12

XM monitor
  introduction, 1–2
  using, 1–25

RT–11 Programmer's
Reference Manual
AA–H378B–TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ___ Assembly language programmer
- ___ Higher-level language programmer
- ___ Occasional programmer (experienced)
- ___ User with little programming experience
- ___ Student programmer
- ___ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country

— — Do Not Tear — Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG/ML PUBLICATIONS, MLO5–5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754

— — Do Not Tear — Fold Here —

Cut Along Dotted Line