Programming with RT-11

VOLUME 2

Callable System Facilities

Stephen Peters
Kevin Small
Anne Summerfield
Julie Wright

The RT-11 Technical User's Series

Programming with RT–11

VOLUME 2

Callable System Facilities

Programming with RT–11

VOLUME 2

Callable System Facilities

Stephen Peters
Kevin Small
Anne Summerfield
Julie Wright

digital
DECbooks

Designed by Virginia J. Mason
Printed in the United States of America

10 9 8 7 6 5 4 3 2

Documentation number EY-00023-DP
ISBN 0-932376-33-9

Library of Congress Cataloging in Publication Data

Clinch, Simon, 1959–
Programming with RT–11.

(RT–11 technical user's series)
Includes indexes.
Contents: v. 1. Program development facilities—v. 2. Callable system facilities.
1. RT–11 (Computer operating system) 2. MACRO–11 (Computer program language) 3. FORTRAN (Computer program language) 4. Basic (Computer program language) I. Peters, Stephen, 1951– . II. Title.
QA76.6.C55 1984 001.64'2 84-5031
ISBN 0-932376-32-0 (v. 1)
ISBN 0-932376-33-9 (v. 2)

Contents

Acknowledgment

We would like to thank all those who contributed to this publication. In particular, we are grateful to Martin Gentry, who reviewed and updated the material in this book, and Bernard Volz, who reviewed the text and revised practice exercises. The staff at Digital Press deserve special commendation for their invaluable assistance.

Introduction

Programming with RT–11 examines the RT–11 facilities that enable you to develop executable programs in MACRO–11, FORTRAN IV, or BASIC–11. *Programming with RT–11* comprises two volumes. Volume 1 covers the program development process, RT–11 debugging aids, libraries, overlays, and the FORTRAN IV and BASIC–11 subroutine conventions for MACRO–11 interfacing. Volume 2 discusses the use of programmed requests to perform file and terminal input/output, foreground/background communication, and synchronous and nonsynchronous input/output operations.

Volume 2 contains chapters 8 through 20. Chapter 8, "Using System Services," examines the system services: when to use them, how they work, and how to call them. Chapter 9, "Gaining Access to System Information," describes the structure of the RMON fixed offset area and the function of each of its parts. Chapter 10, "Controlling Program Execution," discusses chaining and passing command lines to KMON on exit. Chapter 11, "Using Input/Output Systems," gives an overview of device I/O and describes the system components involved in an I/O transfer. Chapter 12, "Using Terminal Input/Output," describes single-character I/O and commands to input or output text a line at a time. Chapter 13, "Using Multiterminal Input/Output," discusses the input and output of characters for multiterminal support. Chapter 14, "Using Queued Input/Output," examines synchronous queued I/O requests for writing to or reading from file-structured and non-file-structured devices. Chapter 15, "Using Nonsynchronous Queued Input/Output," discusses asynchronous queued I/O and event-driven I/O. Chapter 16, "Scheduling and Blocking," examines event-driven blocking and foreground/background scheduling. Chapter 17, "Transferring Data Between Jobs," describes the sharing of buffers and the transfer of data between foreground and background jobs. Chapter 18, "Using Memory," looks at the dynamic allocation of buffers and ways to manipulate the User Service Routine (USR). Chapter 19, "Using the Command String Interpreter," examines the interface between the Command String Interpreter (CSI) and the user and programmer and describes related data structures and programmed requests. Chapter 20, "Writing Time-dependent Programs," discusses programs that make use of timed services, in particular, mark-time requests, scheduling, and setting the system date and time.

Equipment

In order to do the practice exercises, you will need access to a working RT–11 system with at least 500 blocks of disk space for your files. By a *working* system, we mean that:

- The RT–11 monitor program has been transferred from its storage disk to main memory (in other words, the system has been bootstrapped)
- The FORTRAN IV compiler or BASIC–11 interpreter has been installed and is available for use

Resources

Although every effort has been made to make *Programming with RT–11* self-contained volumes, you may need to refer to the following manuals from the RT–11 documentation set for additional information:

- *RT–11 Installation Guide*
- *RT–11 Programmer's Reference Manual*
- *RT–11 Software Support Manual*
- *RT–11 System Generation Guide*
- *RT–11 System Message Manual*
- *RT–11 System User's Guide*
- *RT–11 System Utilities Guide*

The documentation to which we refer throughout the text is written for RT–11 version 5.0. We also used a computer system equipped with RT–11 version 5.0 to generate the programs in our examples and practices. If you own a newer version of RT–11, you may also need a copy of the latest *System Release Notes* to determine the difference between your system and the one described here.

Programming with RT–11 is written under the assumption that you know how to program in MACRO–11, FORTRAN IV, or BASIC–11. The authors assume that you can manipulate files and get directory listings on an RT–11 system and are familiar with RT–11 conventions for device and file specifications, the operation of the Foreground/ Background monitor, and monitor components and their functions. If you need additional information on RT–11 conventions and programming procedures you may refer to the publications listed below:

- *Working with RT–11* (Digital Press, 1983)
- *Tailoring RT–11: System Management and Programming Facilities* (Digital Press, 1984)
- *RT–11/RSTS/E FORTRAN IV User's Guide*
- *BASIC–11 Language Reference Manual*
- *BASIC–11/RT–11 Installation Guide*
- *BASIC–11/RT–11 User's Guide*

For a directory of documentation products, write:
Digital Equipment Corporation
Circulation Department, MK01/W83
Continental Boulevard
Merrimack, NH 03054

Notations

The following symbols are used in this volume to represent specific elements:

⟨KEY⟩ indicates keyboard and keypad keys, their functions, or key combinations

COMMANDS (uppercase) indicates input

Prompts (upper and lowercase) indicates computer output

[ ] indicates parts of a command that are optional (the brackets are not part of the command string)

An example box acts as a window that shows either the interaction between the user and the computer or a portion of the codes in a program. If the code in an example does not include a label, blank spaces have not been included to account for the label field.

8

8

Using System Services

In addition to the utility programs, RT–11 provides a large number of programmed requests and system subroutines (referred to as system services) which gain access to capabilities such as file creation, file maintenance, and event timing.

If you are programming in FORTRAN IV, you reach the RT–11 system services through calls to the system subroutine library SYSLIB.OBJ. This library has a package which handles character strings and supports routines for two-word integers. It also provides access to many system services obtained by MACRO–11 programmed requests. Using these services, you can write most applications programs entirely in FORTRAN IV.

In this chapter you will learn to call system service routines in FORTRAN IV, using SYSLIB functions or subroutines and call programmed requests in MACRO–11 (using the system macro library) and dynamically change the arguments for requests by using an argument block. You will also learn to write code to detect the success or the specific cause of failure of a given programmed request and trap error conditions that would otherwise cause the program to abort.

Programmed Requests

You gain access to the RT–11 system services by using macro calls or FORTRAN IV function or subroutine calls in your program. When your program is assembled or compiled, and linked, these programmed requests provide access to the RT–11 monitor routines which perform functions such as I/O operations, system interrogation (for example, memory contents), and communication between multiple tasks.

Figures 19 and 20 show the use of the PRINT programmed request in a MACRO–11 and a FORTRAN IV program. The PRINT request prints characters from a specified string at the console terminal. The two programs call on the same monitor routine to perform the print operation. This routine performs the I/O operation more efficiently than the FORTRAN formatted I/O system. The services provided as programmed requests are listed and discussed in the *RT–11 Programmer's Reference Manual.*

Executing Programmed Requests

When your program is executing, calls to system services cause a transfer of control to the appropriate monitor code. This code then processes the given arguments and performs the function requested.

Figure 19.
MACRO–11 Program Using the .PRINT Request

```
        .TITLE PRINEX
        ;THIS MACRO PROGRAM PRINTS A STRING USING
        ;THE PRINT REQUEST
        .MCALL .PRINT, .EXIT
STRING: .ASCIZ   /THIS IS A STRING OF CHARACTERS/
        .EVEN
ST:     .PRINT   #STRING
        .EXIT
        .END ST
```

Figure 20.
FORTRAN IV Program Using the .PRINT Request

```
      PROGRAM PRINEX
C     THIS FORTRAN PROGRAM PRINTS A STRING
C     USING THE PRINT REQUEST
      CALL PRINT ('THIS IS A STRING OF CHARACTERS')
      CALL EXIT
      END
```

Load Image Code

Most programmed requests generate code to move the request arguments to some location in memory. In figures 19 and 20 only one argument, the address of the string, must be copied into a general purpose register. When it has been copied, the request generates an emulator trap (EMT) instruction whose execution causes control to be passed to an address listed in a monitor address (or dispatch) table.

The expansion of a programmed request may have as few as one or two machine instructions. The number of instructions depends on the operations performed by that programmed request or the number of arguments specified in a macro call.

The EMT Instruction

When the EMT instruction is executed, control is passed to the EMT processor routine in the monitor. The EMT instruction execution uses a sequence of events very similar to an interrupt which transfers control to an interrupt service routine. This event sequence follows:

1. The current PC (program counter) and PSW (processor status word) registers are saved on the stack.
2. The PC and PSW registers are loaded with the contents of the EMT vector, locations 30 and 32. Loca-

tion 30 contains the address of the EMT processor routine in the RT–11 monitor; location 32 contains the processor status the monitor uses to execute the request.

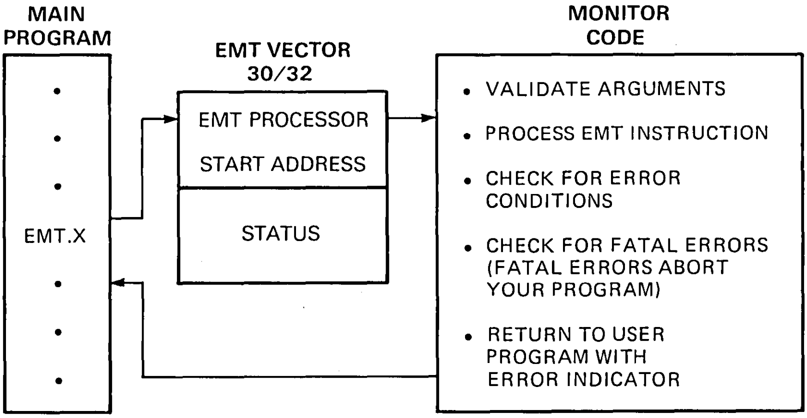
3. The monitor processes the EMT. The type of request is determined and argument checking is done. Invalid or incorrect arguments cause control to be returned immediately to the requesting program.

4. The request is processed; for example, a string of characters is printed on the console.

5. The monitor executes an RTI (return from interrupt) instruction, which restores the PC and PSW register values saved on the stack, and control returns to the requesting program.

The low-order byte of the EMT instruction contains an EMT code, which is interpreted by the monitor according to its value. Table 1–1 in the *RT–11 Programmer's Reference Manual* lists these codes and their meanings. The important EMT codes are 375, 374, and 340 to 357. The forms of programmed requests that generate each of these EMT codes are discussed later in this chapter. Figure 21 shows

Figure 21.
Executing an EMT Instruction


MAIN PROGRAM
•
•
•
EMT.X
•
•
•
EMT VECTOR 30/32
EMT PROCESSOR
START ADDRESS
STATUS
MONITOR CODE
• VALIDATE ARGUMENTS
• PROCESS EMT INSTRUCTION
• CHECK FOR ERROR CONDITIONS
• CHECK FOR FATAL ERRORS (FATAL ERRORS ABORT YOUR PROGRAM)
• RETURN TO USER PROGRAM WITH ERROR INDICATOR

the flow of control during the execution of an EMT instruction generated by a programmed request.

Types of Services

Several types of system services can be accessed by programmed requests and FORTRAN IV subroutines in SYSLIB. The types of services are described in table 7.

MACRO–11 Specific Implementation

MACRO–11 programmed requests are implemented as system macros. The macros are collected in the system macro library SYSMAC.SML and are automatically called when needed during program assembly.

All programmed requests start with a period (.) to distinguish them from programmer-defined symbols and macros. Most programmed requests need arguments, which must be valid assembler expressions. All programmed requests must be explicitly declared using the .MCALL directive to make the macro definition available from the system macro library.

This section focuses on services which have macro calls. Additional capabilities are available using routines found in the FORTRAN IV system subroutine library SYSLIB.OBJ discussed later in this chapter. MACRO–11 programs can gain access to some of these subroutines using the standard MACRO–11 calling conventions.

Form of the Macro Call

You specify MACRO–11 programmed requests in two ways. With requests such as .PRINT or .GTLIN, you simply supply the name of the request followed by the arguments. The format is:

Table 7.
System Services

Type	Functions
Program initialization and control	Allocates memory Allocates input/output resources Turns devices on and off Processes errors
System or job resource and status interrogation	Supplies date and job information
Command translation	Includes the Command String Interpreter (CSI)
File operations	Open, close, create, rename, delete, and change protection status or creation date of files
File I/O operations	Perform synchronous and nonsynchronous I/O
Console terminal I/O	Controls I/O operations and sends and receives data on the console terminal
Multiterminal I/O	Allows your program to control and perform I/O on 1 to 16 terminals
Foreground/background communications	Enable two-way transfer of data in memory buffers
Timer support	Starts or ends jobs on the basis of elapsed system-clock time
System job communication	Allows jobs to communicate
Interrupt service routines	Allow interrupt service routines to communicate with the monitor by means of macros available in the SYSMAC library
Extended memory functions	Include four types of requests for creating and using extended memory (extended memory requests are not available for FORTRAN programs)
INTEGER*4	Performs arithmetic operations on this data type (for FORTRAN programs)
FORTRAN IV character string functions	Compare, copy, find character strings and concatenate ASCII strings
RADIX-50 conversion	Converts FORTRAN IV to RADIX-50 format and RADIX-50 to FORTRAN IV
Miscellaneous FORTRAN IV routines	Allow you to examine and modify absolute memory locations and specify a FORTRAN IV subroutine as an interrupt service that run at a specific priority

.PRGREQ arg1,arg2,...,argn

Here ".PRGREQ" is the name of a programmed request and "arg1,arg2,...,argn" is the list of arguments you give.

EXAMPLE

```
.PRINT   #STRING
```

With requests such as .PEEK or .POKE, you supply the name of the request, the symbolic address of a memory area where the arguments will be stored, and the arguments. The format is:

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

Here ".PRGREQ" is the name of a programmed request; "area" is the symbolic label pointing to the EMT argument block, which is a set of words used to pass the arguments to the monitor; and "arg1,arg2,...,argn" is the list of arguments you give.

EXAMPLE

```
         .GTIM    #AREA,#TIMBUF
           .
           .
AREA:    .BLKW    2  ;Argument Block for GTIM
TIMBUF:  .BLKW    2  ;Buffer to receive System Time
```

Passing of Arguments

Macros of the simple format (type 1) generate either an EMT 374 or one of the EMTs in the range 340 to 357. Requests that generate an EMT 374 have only one argument. R0 contains a function code in the high-order byte, to indicate which request is used. Requests that generate EMTs 340 to 357 are each unique to one programmed request. The corresponding programmed requests have their arguments either in R0, or on the stack, or both.

The macros in which you specify a memory area (type 2) always generate an EMT 375. R0 contains the address of "area" within your program. The first word of the "area" block is set by the programmed request to identify the call. The remaining words in the block may contain pointers to each of the arguments "arg1" to "argn."

The macro expansion moves the specified arguments into the argument block. If your program has previously moved values into the argument block, you can omit those arguments in the macro call. In this way you can change the macro arguments dynamically.

R0 is often used to return important information to your program after the request has completed, and so it is never preserved across a call.

Macro Expansion

You seldom need to examine the actual code generated on expansion of a programmed request; however, you may find it useful, or even necessary, to trace through the expansion in order to detect programming errors. An example of a MACRO–11 programmed request expansion (shown in figure 22) is the assembly listing of the code in figure 19. The .LIST MEB directive is used to direct the assembler to list the expansion. Note that the expansion of .PRINT passes the address of the string to the EMT processor via %0, which is the register R0.

Error Return

In addition to processing programmed requests used in your program, the monitor can return error information based on the results of executing these requests. If an error occurs during execution of the request, the monitor returns to your program with the C-bit set.

Figure 22.
Expanding the .PRINT Programmed Request

```
 1                                                  .TITLE PRINEX
 2                                                  .LIST MEB
 3                                                  ;THIS PROGRAM PRINTS
 4                                                  ;A STRING USING THE
 5                                                  ;PRINT REQUEST
 6                                                  .MCALL .PRINT, .EXIT
 7 000000    101      040      123      STRING:    .ASCIZ /A STRING/
   000003    124      122      111
   000006    116      107      000
 8                                                  .EVEN
 9 000012                               ST:        .PRINT   #STRING
   000012   012700   000000'                        MOV      #STRING,%0
   000016   104351                                  EMT      ^O351
10 000020                                           .EXIT
   000020   104350                                  EMT      ^O350
11          000012'                                 .END     ST
```

Some programmed requests may return one of many possible error conditions. Each condition is identified by a code returned by the monitor in byte 52. Your program should refer to byte 52 with absolute addressing, gaining access to it as a byte. Never refer to location 52 as a word because byte 53 has a separate function. Chapter 9, "Gaining Access to System Information," gives more details about this byte and other system data. The code in figure 23 tests the result of a programmed request execution.

Figure 23.
Error Checking Code

```
AREA:   .BLKW    M                  ; Argument block
        ERRBYT=52                   ; Error byte (absolute address)
        .
        .
        .READ    #AREA, ......
        BCS      ERROR              ; Check if programmed request
                                    ;  was executed without error

        .

ERROR:  TSTB     #ERRBYT          ; Now check type of error
        .
```

The codes that may be returned for any programmed request are individually listed and defined in chapter 2 of the *RT–11 Programmer's Reference Manual.* You will find that many requests return no error information.

Serious Error Conditions

Serious errors cause a message to be generated by the monitor and printed on the console terminal. Fatal errors cause termination of your program, instead of an error return. The monitor prints a message indicating the type of error after the code:

```
?MON-F-
```

Some fatal errors can be trapped and have their values returned in byte 52. These are discussed below. You should always supply appropriate error checking after a program request.

Return of Auxiliary Information

In general, the content of R0 is not saved across a programmed request. It may be used to return important information to your program. All other registers are preserved. For example, the programmed request .LOOKUP opens a file on a device. On completion, R0 contains the number of blocks in the file that has been opened.

Chapter 2 of the *RT–11 Programmer's Reference Manual* describes each of the MACRO–11 programmed requests. Each description indicates what the request does, how it is called, what information is returned in R0, and what possible error conditions may be returned.

FORTRAN IV Specific Implementation

The FORTRAN IV implementation of programmed requests is in the form of system subroutines, which are stored in the system library SYSLIB.OBJ. These subroutines implement a number of the programmed requests, as well as other functions listed in table 7.

Form of the FORTRAN Call

SYSLIB subroutines are called in the same way as user-written subroutines. SYSLIB contains both function and subroutine subprograms. Function subprograms receive control by means of a function reference, in the form:

Variable=function-name (arguments)

Subroutine subprograms are called using the CALL statement:

CALL subroutine-name (arguments)

All routines in SYSLIB can be called as function subprograms to return the value of the routine, or as subroutine subprograms if no return value is needed. If you use a function call for a SYSLIB subprogram that does not return results, the value returned has no meaning.

Figure 24 uses the AJFLT function subprogram, which converts an INTEGER*4 value to a REAL*4 value, returning it as the function value. The code in figure 24 converts the INTEGER*4 value in JVAL to REAL*4, multiplies it by 3.5, and stores the result in VALUE.

Figure 24.
A SYSLIB Call

```
C DECLARE VARIABLES
      REAL*4 VALUE
      INTEGER*4 JVAL
C PROGRAM CODE
        .
        .
      VALUE=AJFLT(JVAL)*3.5
```

Error Return

Some SYSLIB routines that return a condition code value allow you to determine if the operation of the routine was successful. Your program must check this value to see if an error occurred. For example, the function JADD computes the sum of two INTEGER*4 values. The result assigned to the variable on the left side of the assignment operator (=) is set to one of the four values, depending on the result of the computation. So, the code:

```
K = JADD(I,L,M)
```

computes the sum of I and L, stores the result in M, and returns a condition code in K. The possible codes for the JADD function are:

−2	For an overflow
0	For a normal return of zero
1	For a normal return with a positive result
−1	For a normal return with a negative result

The use of JADD to check for an error on return from a SYSLIB call is shown below.

EXAMPLE

```
IF (JADD(I,L,M).EQ.-2) STOP 'ERROR: JADD OVERFLOW'
```

You should always check for an error return after any SYSLIB call.

Other Error Conditions

Other system services that allow your program to control the monitor's action when serious error conditions occur are mentioned here primarily to let you know that they exist. The *RT–11 Programmer's Reference Manual* provides details and examples of use.

Control of Serious Error Conditions

Normally, when a serious error occurs in a programmed request, the system aborts the job and displays an error message. In some applications you may want a program to recover, or continue, after a serious error. If the .SERR request is used, then any serious errors are reported to the program. The carry bit is set, and byte 52 contains a negative code to indicate that the error is serious. Table 2–2 of the *RT–11 Programmer's Reference Manual* lists the error codes returned by .SERR.

The .HERR request (the system default) allows the system to abort your program on serious errors and generate a system error message. This request overrides the .SERR request.

Trap Handlers

Some programs execute instructions or reference memory that may not be present. These programs may check whether particular instructions (for example, floating point instructions) are supported by the CPU, whether particular hardware devices (such as the RT–11 memory management unit, I/O peripherals) are present, or the amount of memory available. If a tested address does not exist, a trap to 4 occurs. If an instruction that is checked for is not supported, a trap to 10 occurs. RT–11 normally aborts the program when a trap to 4 or 10 occurs. Before aborting the program, the monitor displays a message:

```
?MON-F-TRAP to 4
```

or

```
?MON-F-TRAP to 10
```

To prevent the monitor from aborting the program, you can use the .TRPSET programmed request.

You write a trap handler to recover from the error. The .TRPSET request declares the address of the trap handler. The trap handler routine must also issue another .TRPSET request before it exits because RT–11 cancels .TRPSET each time a trap occurs. See chapter 2 of the *Programmer's Reference Manual* for details.

Floating Point Exception Handlers

If you have the floating point hardware option, the .SFPA request allows you to set up a routine to handle floating point exceptions. If you do not set up a trap address, then your job is aborted by the monitor when a floating point exception condition occurs. .SFPA is also used under the FB and XM monitors to allow the floating point registers to be used by more than one job.

References

RT–11 Programmer's Reference Manual. Chapter 1 provides MACRO–11 programmers with detailed information on EMT instruction codes, programmed request format, programmed request errors, and other error conditions. It also introduces the FORTRAN IV system subroutine library.

Working With RT–11. Chapter 6 discusses utility programs.

PDP–11 Processor Handbook. Refer to the handbook written for the PDP–11 processor on your system for additional information on EMTs, traps, interrupts.

9

9

Gaining Access to System Information

The RT–11 monitor maintains in memory a large amount of information about its current state and the state of jobs currently running. Some of this information is useful to applications programs and can be obtained from MACRO–11 and FORTRAN IV programs using system services provided for that purpose. In this chapter you will learn how to read and modify this data using the programmed requests: .PEEK, .POKE, .GVAL, and .GTJB (MACRO–11) and IPEEK/B, IPOKE/B, ISPY, and GTJB (FORTRAN IV).

You will also learn to use two MACRO-11 programmed requests, .MFPS and .MTPS, to gain access to the processor status word (PSW) independent of the processor type.

When you have completed this chapter, you will be able to obtain data from any location in the system communication area; get the data from any location in the RMON fixed offset area or in the job's impure area; get job information from RMON using the GTJB request; and identify the restrictions on the use of the RMON fixed offset area and the impure area.

System Information

The Resident Monitor (RMON) maintains a large amount of information about its own status and that of jobs currently running. This is held in three main areas of memory:

- The system communication area (SYSCOM)
- The RMON fixed offset area
- The impure area (for each job)

Figure 25 shows the location of these areas in the memory of a typical FB monitor system.

The system communication area (SYSCOM) occupies absolute locations 40 to 57 in memory. This area contains information about the job currently running. Some of this information is provided by the linker and copied into memory when the program is loaded. The remaining locations are used at run time to enable RMON and the job to communicate.

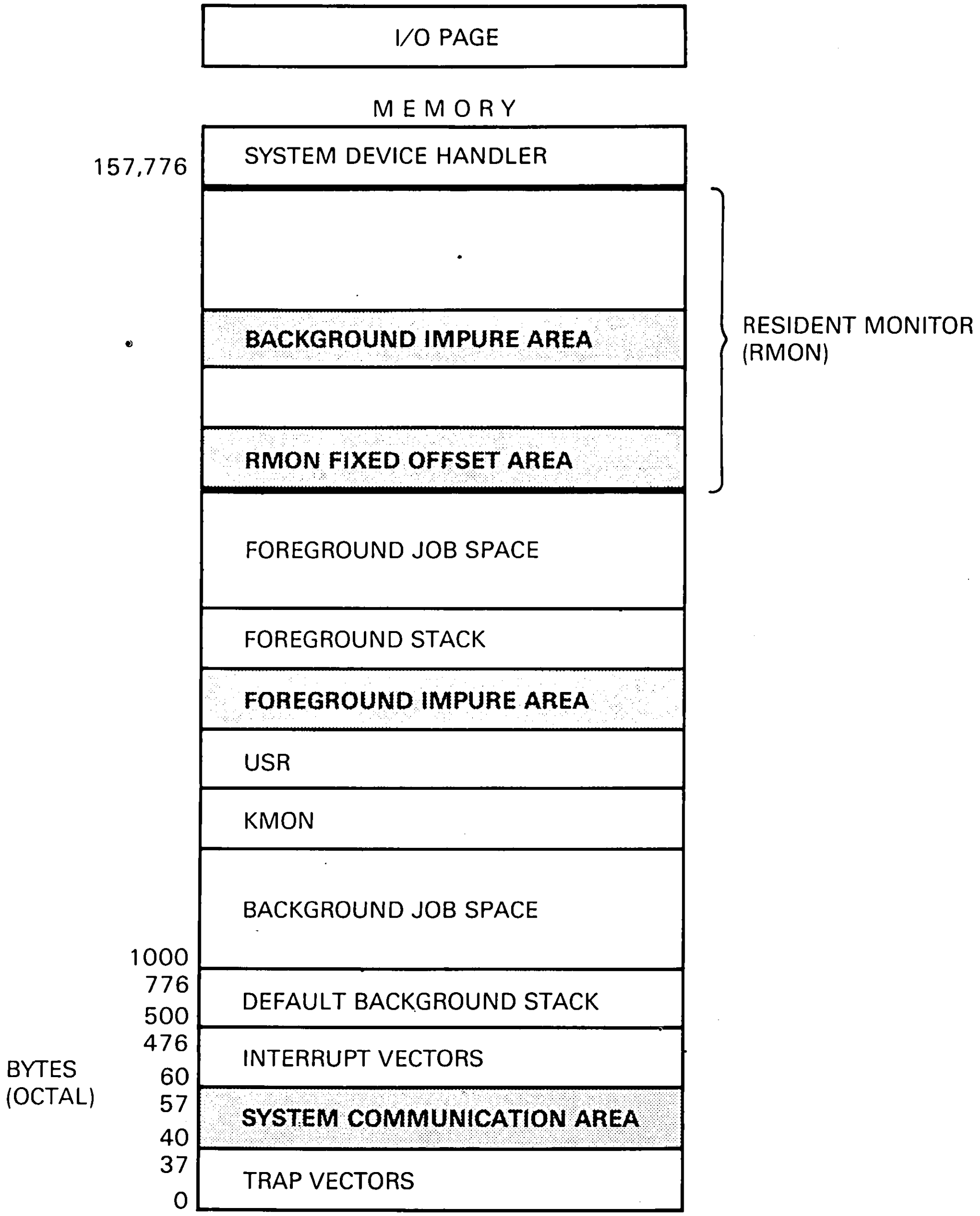
The structure of RMON differs from monitor to monitor, depending on the SYSGEN options selected. Some data, however, are always located at a fixed position relative to the start of RMON. This area, called the RMON fixed offset area, contains information about the monitor itself and about the current hardware and software configuration.

Each job also has its own impure area, which contains information specific to the job, including the terminal input/output buffers. This area is maintained and used by RMON, but user programs can also retrieve data from it (except in SJ systems).

System Communication Area

The system communication area (SYSCOM) occupies locations 40 (octal) to 57 (octal) in memory, and contains information about the job currently executing. The contents of each of these locations are described below.

Figure 25.
Information Areas in a Foreground/Background System



Information set by the linker and copied into memory from the load image:

1. Word 40 (locations 40 to 41) contains the normal start address of the first executable instruction in the program. This is used by the R, RUN, and START commands.
2. Word 42 (locations 42 to 43) contains the initial value of the stack pointer for the job currently executing. By default, the top of the stack is immediately below the lowest program address (at location 776 for a background job). The default can be changed using the .ASECT directive or, for a background job, the linker /STACK option.
3. Word 50 (locations 50 to 51) contains the program high limit—the highest address your program can use. In XM systems, it is the address of the top of the root section plus the low memory overlay regions. Programs must never change this word directly. If you want to change it, use the .SETTOP directive.

Information given by RMON to the job:

1. Byte 52 is the monitor error byte. If RMON detects errors in a programmed request, it places the error code in this byte and sets the carry bit.
2. Word 54 (locations 54 to 55) is the address of the start of RMON. This can be used as a pointer to the RMON fixed offset area, as discussed later. Your programs must never modify the contents of this word.

Information given by the job to RMON:

1. Word 44 (locations 44 to 45) contains the job status word (JSW) used to control the operation of some of the programmed requests and to enable certain programmed request features. This word is discussed later.

2. Word 46 (locations 46 to 47) contains the alternate User Service Routine (USR) load address. Its value is normally zero, which indicates that the USR swaps into its default location below RMON, the foreground and system jobs, and the device handlers. If you set this word to a value other than zero, either in the load image file or at run time, this value is used as the address at which the USR swaps. The swapping position should be selected with care. For more information, see chapter 18, "Using Memory."

3. Byte 53 is the user error byte. The user job sets a value in this byte to indicate whether any errors have occurred during program execution. For example, code 0 means that the job has terminated successfully. In indirect file processing, KMON examines this byte upon program termination. If a significant error has been reported, KMON can abort any indirect files in use when commands that follow depend on successful completion of this program.

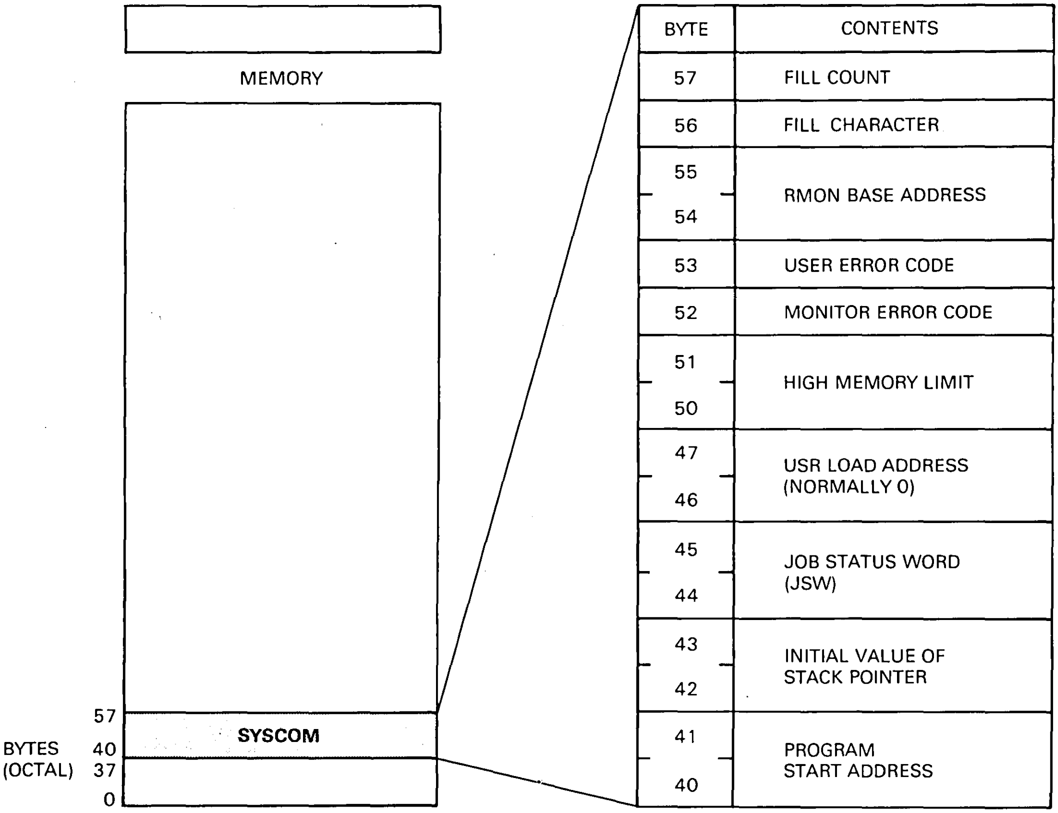
4. Byte 56 (the fill character) and byte 57 (the fill count) are used to specify the type and number of fill characters needed by some types of low-speed terminals. Because of the transmission rate, these terminals must have fill characters (nulls) inserted after certain characters. More information is given in chapter 2 of the *RT–11 Installation Guide.*

The contents of locations in the system communication area are shown in figure 26. Detailed information is given in chapter 2 of the *RT–11 Software Support Manual.*

The Job Status Word (JSW)

The settings of the individual bits of the JSW (word 44 in the system communication area) are used to control the operation of certain programmed requests and indicate whether certain special features are enabled. Some of the bits may be modified by user programs either at load time or during

Figure 26.
System Communication Area (SYSCOM)



BYTE	CONTENTS
57	FILL COUNT
56	FILL CHARACTER
55 54	RMON BASE ADDRESS
53	USER ERROR CODE
52	MONITOR ERROR CODE
51 50	HIGH MEMORY LIMIT
47 46	USR LOAD ADDRESS (NORMALLY 0)
45 44	JOB STATUS WORD (JSW)
43 42	INITIAL VALUE OF STACK POINTER
41 40	PROGRAM START ADDRESS

program execution. The word may also be set from the terminal or from an indirect file using the D (deposit) command.

Application Examples

Later you will need to access locations in the system communication area for specific applications, for instance:

1. You may want to write programs that do not echo input received from the terminal, for example, passwords. You can do this by setting bit 12 of the JSW (the special mode terminal bit).

2. When you use programmed requests to perform queued I/O, you may have to load a device handler from your program into memory. It is convenient to place the handler immediately above your program. To do this, you need to know your program's high memory limit, which is held in word 50 of SYSCOM.

Gaining Access to SYSCOM and Other Absolute Locations

RT–11 provides certain programmed requests that enable you to call absolute locations from MACRO–11 and FORTRAN IV programs. These requests are .PEEK and .POKE for MACRO–11, and IPEEK/B and IPOKE/B for FORTRAN IV.

Reaching Absolute Locations from MACRO–11 Programs

The .PEEK programmed request returns in R0 the contents of a memory location. .POKE changes the contents of a location. The macro calls are:

```
.PEEK   area,addr
.POKE   area,addr,value
```

In these macro calls:

area	is the address of a two- or three-word EMT argument block
addr	is the address of the location to examine
value	is the new contents to place in the location

The following example shows how to load R0 with the base address of RMON from word 54 of SYSCOM.

EXAMPLE

```
.PEEK    #EMTBLK,#54 ;RMON address to R0
```

When you use .POKE, be sure that your addressing modes are correct. In the following example, the two statements are equivalent because R2 contains the address of the location you want to change (such as 44), and R1 points to the data you want to move.

EXAMPLE

```
.POKE   #AREA,R2,(R1) ;Move data pointed to by R1
                      ;to address in R2
MOV     (R1),(R2)     ;Move data pointed to by R1
                      ;to location pointed
                      ;to by R2
```

Unless you intend to use the XM monitor, you can also access absolute addresses directly. In the following example the instruction sets bit 6 of word 44 (the JSW).

EXAMPLE

```
BIS     #100,@#44      ;Set JSW bit 6
```

Always use the .PEEK and .POKE programmed requests to access absolute memory locations that are in low memory, so that your programs can be run under any monitor.

Notice that .PEEK and .POKE only access word locations. If you want to access a byte, use .PEEK to retrieve the complete word and change only the bits in the high- or low-order byte, as appropriate. You can also use this method to set or clear specific bits in a word or byte, by combining .PEEK; a BIS or BISB instruction to set bits, or a BIC or BICB instruction to clear bits; and .POKE.

When you use .PEEK followed by .POKE, you must move the value returned in R0 to another location before you perform any operation on that value. Since these two requests use an EMT argument block, R0 is corrupted when you issue .POKE. The following example shows the correct use of .PEEK and .POKE to clear bits 5 and 6 of the JSW.

EXAMPLE

```
.PEEK   #EMTBLK,#44      ;Get JSW in R0
MOV     R0,R1            ;Move value to R1
BIC     #140,R1          ;Clear bits 5 and 6, then
.POKE   #EMTBLK,#44,R1   ;Write JSW back from R1
```

You should use .PEEK and .POKE with all RT–11 monitors for compatibility.

Reaching Absolute Locations from FORTRAN IV Programs

The FORTRAN IV language standard does not provide statements that allow your program to access memory locations by their absolute address. You must use the IPEEK and IPOKE functions to examine and modify these locations.

The IPEEK function returns the contents of the address specified as an argument. By default, the address is interpreted as a decimal value, so if you want to reference an address in octal, you must precede the first digit with a quotation mark (“). For instance, in the following example, the statement loads the variable IRMON with the contents of word 54 (octal) of SYSCOM—the base address of RMON.

EXAMPLE

```
IRMON=IPEEK("54)
```

Notice that the address given must be an even number because it is a word address. The corresponding function to retrieve the contents of a byte address is IPEEKB. The following example shows a statement that stores the user error code from byte 53 (octal) in the variable IUSERR.

EXAMPLE

```
IUSERR=IPEEKB("53)
```

The subroutines IPOKE and IPOKEB load a specified value into a specified word (IPOKE) or byte (IPOKEB) address. The statement in the following example loads the contents of variable IUSR into word 46 (octal)—the alternate USR swap address.

EXAMPLE

```
CALL IPOKE("46,IUSR)
```

For words such as the JSW, you will usually want to set or clear specific bits in the word instead of modifying the complete word. You can do this by combining IPEEK, a logical operation (such as a logical OR), and IPOKE. The logical operation uses a bit pattern called a mask. You can use this combination to set all the bits of a mask in the word, or clear all the bits of a mask from the word.

In the following example, a mask of 5 (binary 101 where bit 0 and bit 2 are set) is used to set and then clear bits 0 and 2 in a word that has value 3 (binary 011).

EXAMPLE

To set bits 0 and 2:

word	011
mask	101
result of .OR.	111

To clear bits 0 and 2:

word	011
.NOT. mask	010
result of .AND.	010

Figures 27 and 28 show FORTRAN IV subroutines that set and clear specific bits of a word.

The following code offers a shorter and more efficient way to perform the same operation. To set a bit you can use the statement:

```
CALL IPOKE(IADDR,IPEEK(IADDR).OR.IMASK)
```

Figure 27.
FORTRAN IV Bit-setting Routine

```
      CALL BITSET("44,"1400)   !SET BITS 8 AND 9 OF THE JSW
           .
           .
      SUBROUTINE BITSET(IADDR,IMASK)
C...     SETS ALL BITS IN "IADDR" WHICH ARE SET IN
C...     "IMASK".  THE OTHER BITS ARE NOT CHANGED.
      IOLD=IPEEK(IADDR)
      INEW=IOLD.OR.IMASK
      CALL IPOKE(IADDR,INEW)
      RETURN
      END
```

Figure 28.
FORTRAN IV Bit-clearing Routine

```
      CALL BITCLR("44,"1400)    !CLEAR BITS 8 AND 9 OF THE JSW
          .
          .
      SUBROUTINE BITCLR(IADDR,IMASK)
C...     CLEARS ALL THE BITS IN "IADDR" WHICH ARE SET
C...     IN "IMASK".  THE OTHER BITS ARE NOT CHANGED.
      IOLD=IPEEK(IADDR)
      INEW=IOLD.AND..NOT.IMASK
      CALL IPOKE(IADDR,INEW)
      RETURN
      END
```

To clear a bit:

CALL IPOKE(IADDR,IPEEK(IADDR).AND..NOT.IMASK)

EXAMPLE

```
CALL IPOKE("44,IPEEK("44).OR."100)
```

sets bit 6 of the JSW.

RMON Fixed Offset Area

The location of routines and data within the resident monitor depends on the SYSGEN options selected, so it differs from monitor to monitor. Some data, however, is always located at the same position relative to the start of RMON. This area is called the RMON fixed offset area. It contains information about the monitor itself, the current hardware configuration, and certain software conditions.

Contents of the RMON Fixed Offset Area

The full list of the contents of the RMON fixed offset area is given in table 3–8, chapter 3 of the *RT–11 Software Sup-*

port Manual. Some of the more commonly called offsets in this area are:

- Word 266 contains the default USR base address. This is the address where the USR resides when it is called into memory by the background job, and location 46 of SYSCOM is zero.
- Byte 276 is the monitor version number.
- Byte 277 is the monitor release number.
- Word 300 is the configuration word. The bit settings within this word give information about the hardware configuration and software conditions on your system.
- Word 370 is the extension configuration word. (See word 300.)
- Word 372 is the system generation features word. This location holds information indicating which major SYSGEN options are present.

Application Example

Bit 5 of the configuration word indicates whether your system has a 60- or 50-cycle clock. You might want to write a program that correctly processes the time and date information available through programmed requests (.GTIM and .DATE) under a 50–Hz or 60–Hz system. Such a program needs to refer to the RMON fixed offset area to find out at which frequency the system is running. Time-dependent programming is covered in chapter 20 of this book.

Retrieving Data from the RMON Fixed Offset Area

RT–11 has programmed requests specifically designed to retrieve data from the RMON fixed offset area. In MACRO–

11 programs, the .GVAL programmed request performs this function. The statement in the following example returns the contents of offset 266 (the normal USR swapping address) in R0. EMTBLK must be a two-word EMT argument block.

EXAMPLE

```
.GVAL    #EMTBLK,#266
```

When writing user programs, beware of modifying the contents of the RMON fixed offset area. This causes changes within the monitor. If you want to change the contents of the monitor, you use .PVAL (MACRO–11) or IPUT (FORTRAN IV). The *RT–11 Programmer's Reference Manual* covers this topic in detail.

In FORTRAN IV programs, you use the ISPY system service function to retrieve data from the RMON fixed offset area. ISPY calls the .GVAL programmed request to return the integer value of the word at a specified offset from the base address of RMON. The following example returns the configuration word (offset 300) in the variable ICONF.

EXAMPLE

```
ICONF=ISPY("300)
```

Impure Area

Each job under the FB and XM monitors has its own impure area which is maintained by the resident monitor. This area contains job-dependent data used primarily for:

- Input/Output
- Scheduling and blocking
- Memory access control (XM only)

The impure area for the background job is located within RMON, above the fixed offset area. The foreground job impure area is located below the foreground job and its stack (refer to figure 25).

The contents of the impure areas are listed in table 3–13 in chapter 3 of the *RT–11 Software Support Manual.* Notice that some locations cannot be addressed as fixed offsets, since locations change from one release of RT–11 to the next. SYSGEN options also affect these locations.

Retrieving Data from the Impure Area

To gain access to locations from the start of the impure area at fixed offsets, you need to know:

- The address of the start of the impure area
- The fixed offset

You can use the GTJB (get job information) programmed request to get information about the job, including a pointer to the impure area. The syntax of the request for MACRO–11 is:

```
.GTJB   area, addr
```

In this request, "area" is the address of a three-word EMT argument block, and "addr" is the address of an eight- or twelve-word block in which system information related to the job is returned.

The syntax of the GTJB request for FORTRAN IV is:

```
CALL GTJB(array)
```

In this request, "array" is an eight- or twelve-word integer array in which the data is returned.

Under FB or XM monitors that have the system jobs feature, twelve words of storage must be allocated in your program for the job information to be returned. Otherwise, eight words are sufficient. On return from the request, word 5 of this area contains the address of the job's impure area. This word has no meaning under the SJ monitor.

Figure 29.
MACRO–11 Code to Retrieve Data from Offset 32

```
AREA:    .BLKW    3               ;EMT ARGUMENT BLOCK
BLOCK:   .BLKW    8.              ;JOB INFORMATION  AREA
            .
            .
         .GTJB    #AREA,#BLOCK    ;GET JOB INFORMATION
         MOV      BLOCK+10,R1     ;ADDRESS OF IMPURE AREA
                                  ;FROM WORD  5  (BYTE   10 OCTAL)
         ADD      #32,R1          ;ADD  OFFSET  (32)
         .PEEK    #AREA,R1        ;RETRIEVE OUTSTANDING I/O
                                  ;COUNT FROM IMPURE AREA INTO R0
```

Figure 30.
FORTRAN IV Codes to Retrieve Data from Offset 32

```
      INTEGER*2  IBLOCK(8)
C...    JOB INFORMATION AREA
      CALL GTJB(IBLOCK)
      IWORD=IPEEK("32+IBLOCK(5))
C...    ADD OFFSET TO START ADDRESS OF IMPURE AREA
C...    AND RETRIEVE THE OUTSTANDING I/O COUNT.
```

When you know the address of the start of the impure area, add the offset of the location you want to access. Then access the contents of the location in the area using the .PEEK (for MACRO–11) or IPEEK (for FORTRAN IV) programmed requests. For instance, suppose you want to know how many I/O requests are outstanding for your job. This is held in offset 32 (octal) of the impure area. Figures 29 and 30 show MACRO–11 and FORTRAN IV code that retrieve data from this location.

Gaining Access to the Processor Status Word

Two programmed requests, which can only be used in MACRO–11 programs, allow processor-independent access to the processor status word (PSW). The two requests are .MFPS and .MTPS. The contents of R0 are not destroyed by either call. The .MFPS request is used to examine the priority bits only. Condition codes are destroyed during the call. The .MTPS request is used to load the priority bits.

These requests are useful since they make programs transportable. Some PDP–11 models support direct access to the PSW via address 177776, while others only support the MTPS and MFPS instructions. Some models support both methods.

References

RT–11 Programmer's Reference Manual. Chapter 1 describes EMT codes, programmed request format, and programmed request errors. Chapter 2 discusses .GTJB, .MFPS, .MTPS, .PEEK, and .POKE programmed requests in MACRO–11 programs and using .PVAL in modifying the contents of the RMON fixed offset area. Chapter 3 explains the IPEEK, IPEEKB, IPOKE, and IPOKEB routines and the ISPY and GTJB requests in FORTRAN IV programs. Chapter 3 also explains how to use IPUT in modifying the contents of the RMON fixed offset area.

RT–11 Software Support Manual. Chapter 2 discusses the user error byte and various error conditions. Table 2–4 shows the meaning of each bit in JSW. Chapter 3 explains bit settings for the configuration word (offset 300), the extension configuration word (offset 370), and the system generation features word (offset 372).

RT–11 Installation Guide.

10

10

Controlling Program Execution

This chapter discusses the different ways you can start and stop program execution. Normally, you use the RUN command to load a background program into memory and start execution at the first instruction. Sometimes, such as when you are debugging, you may want to separate the tasks of loading the job and starting execution, or start execution at a different address. This chapter describes the monitor commands used to do this. It also describes how to write MACRO–11 programs that have more than one entry point.

When a program completes, it returns control to KMON or to any indirect file that was active when the command was issued to run the program. Programs may complete successfully, or they may stop because of an error condition. On completion, your program can pass one or more command lines to KMON, which executes these commands before prompting for commands from the keyboard. You can also write background jobs that pass control and information to another job on completion.

This chapter describes the monitor commands: GET, FRUN, RUN, R, START, and REENTER. The programmed requests and SYSLIB subroutines discussed are: .EXIT, .CHAIN, EXIT, SETCMD, CHAIN, RCHAIN, RAD50, IRAD50, and R50ASC.

In this chapter you will learn to restart or reenter a job. You will be able to write a program that passes one or more command lines to KMON on exiting, and you will learn to write two programs; the first of which executes the second and passes information to it.

If you are programming in FORTRAN IV, you will write a program which, upon termination, exits to the monitor without printing any termination messages. If you are programming in MACRO–11, you will write a program that can be reentered.

Starting Execution

When you execute a program using the RUN or R command, execution starts at the transfer address. In MACRO–11, this is the address specified as the argument to the .END directive in your main module. In FORTRAN IV, it is the address of the first instruction in the main program.

When the program is linked, the transfer address is stored in word 40 of the load module file. When the program is loaded into memory, this address is in word 40 of the system communication area (SYSCOM). You can change the transfer address of a program written in either MACRO–11 or FORTRAN IV by specifying the /TRANSFER option when the program is linked.

Starting Foreground Jobs

RT–11 supports only one way of starting a foreground job—the FRUN command. This loads the program into memory and starts execution at its transfer address. Options to FRUN include /BUFFER:n (reserves extra space in memory), /NAME:name (assigns a logical name to the foreground job), and /PAUSE (allows you to debug a program).

Starting Background Jobs

The RUN command is normally used to load background jobs into memory and start execution at their transfer address. You can separate the tasks of loading and executing by issuing a GET command followed by a START. These commands can be useful if you are debugging the program or if you want to run the same job several times. When a job completes successfully, it remains in memory until you load another program with RUN or GET. You can execute the job again using the REENTER or START command. These commands have the following features:

- RUN loads a background program into memory, from the specified device (default DK:), and starts execution at the transfer address.
- R is similar to RUN, except that it can only load the program from the system device because it is not capable of loading any other device handlers.
- GET loads a program into memory but does not start execution.
- REENTER starts execution at the reentry address, which is the transfer address minus two (bytes). You can REENTER a program only if it sets bit 13 of the JSW and then exits normally (exiting is discussed later). You can use this command to place a second entry point in MACRO–11 programs. This command is less useful if you are programming in FORTRAN IV because you cannot control where instructions are placed when the program is linked.
- START starts execution at the specified address. If no address is given, it starts at the transfer address.

You use START instead of RUN when:

- You want to execute a program several times. (Saves the time needed to fetch the program from disk for each execution.)

- You want to use data created during the first execution.
- You want to debug your program using the methods discussed in chapter 4, "Debugging Programs," of *Programming With RT–11, Volume 1.*

In the following example, the normal transfer address is at the label START, and the reentry address is the branch instruction.

EXAMPLE

```
        BR      ENTRY2          ;Reentry point
START:                          ;Initial entry point
            .
        Initialization code
            .
            .
ENTRY2:
            .
        Common code
            .
            .
        .END    START
```

If you want to have more than two entry points, you must use START and specify the address for each entry point (from the load map). Certain restrictions apply when you use these commands with overlaid programs or programs that use extended memory features. For details about these commands, refer to chapter 4 of the *RT–11 System User's Guide.*

Exiting

Exiting is the process of terminating a program and returning control to the monitor. All user programs should exit correctly when they have completed processing. They should not be allowed to loop infinitely or halt the system.

After your program has properly exited, you can proceed with other work.

Exiting from MACRO–11 Programs

To terminate a MACRO–11 program, you use the .EXIT programmed request. Your program can perform either a normal exit or a RESET operation (if you are using the SJ monitor), depending on the contents of R0 when you issue the .EXIT request.

- If R0 is set to a value other than zero, a normal exit is performed. Mark time requests are cancelled and I/O operations are allowed to complete.
- If R0 is set to zero, a RESET operation is performed. Marked time requests are cancelled and I/O operations are aborted. Under SJ, this operation is performed by the PDP–11 RESET instruction.

If your program recognizes that a significant error has occurred during execution, it should clear R0 on exit so that the program cannot be restarted.

Exiting from FORTRAN IV Programs

You can exit from a FORTRAN IV program in two ways: first, by issuing a STOP command, which prints a message like:

```
STOP 'text'
```

at the terminal when the program exits.

The second way is to call the EXIT subroutine. This subroutine is in the FORTRAN IV subroutine library, which is usually combined with SYSLIB. EXIT does not display any termination messages. The format is:

```
CALL EXIT
```

The type of exit performed by a FORTRAN IV program is determined by the FORTRAN IV OTS. A normal exit is performed unless the OTS recognizes that a fatal error has occurred, in which case the program stops with an error message.

Passing Commands to KMON

When a background program exits, control returns to KMON, which is then ready to accept more monitor commands, either from the keyboard or from any active indirect file. Your program can optionally pass one or more monitor commands to KMON when it exits. These commands are executed before any more commands are read from the keyboard or indirect file.

Passing Commands from MACRO–11 Programs

To pass command lines to KMON when your program exits, perform the following steps:

1. Move the command lines into locations 512 to 777 (octal). Each command must be terminated by a null byte (that is, an ASCIZ string).
2. Place a count of the number of bytes in the command lines into the word (not byte) at location 510 (octal).
3. Set bit 11 of the JSW (the "pass line to KMON bit") immediately before the .EXIT request.
4. Issue the .EXIT with R0 = 0.

Passing Commands from FORTRAN IV Programs

The SETCMD programmed request is used to pass a command line to KMON from a FORTRAN IV program on exit. Note the following points:

1. Only one command line can be passed to KMON from FORTRAN IV programs. If you want to pass more than one command, make an indirect command file and pass a command to KMON to execute this file.

2. If you pass any command lines to KMON, any indirect files that were active at the time your program was invoked are aborted.

3. The argument to SETCMD can be either a quoted string, or a variable or array name. The command line must be terminated by a zero byte (ASCII null). (This is the equivalent of a MACRO–11 ASCIZ string.) If you use the normal FORTRAN IV OTS input statements (READ, ACCEPT) to read in the command line, then you must put this byte in the string yourself. There are also programmed requests for terminal input which automatically place a zero byte at the end of the string. These are discussed in chapter 13, "Using Multiterminal Input/Output."

Chaining

When you design applications programs, you may find it necessary or desirable to split a programming task into two or more programs. You may want to do this if one program performing the complete task is too large to fit into memory, or if some parts of a program are not needed every time the program is run.

If you create two or more related background jobs to do a single programming task, you will find it useful to make one job capable of starting the other without operator action. It is also useful for the first job to be able to pass information to the second without having to write the information to disk.

RT–11 allows you to transfer control from one program to another by a process called chaining. The area that contains the information you pass from one job to another is called the chain information area, and it occupies locations 500 to 777 (octal). You should not rely on any other locations being preserved from one job to the next. In virtual programs run under the XM monitor, locations 500 to 777 are not saved, so jobs cannot be chained. Privileged programs under XM can, however, chain.

Notice that location 1000 (octal), which is the top of the chain area, is also the default initial value of the stack pointer in background jobs. If your program needs a lot of stack space and, in addition, you want to place information into the chain area at compile time, this might result in the stack and the chain information overlapping. To avoid this, assign more space to the stack, using the methods described in chapter 1, "Developing Programs in MACRO–11 and FORTRAN IV," of *Programming with RT–11, Volume 1.*

Chaining in MACRO–11 Programs

To enable an outgoing job to chain to an incoming job and pass information to it, you should perform the following steps. Place the file specification (in RAD50 format) for the incoming job into bytes 500 to 507 (octal) of the outgoing job. (Read the section, "Using RAD50 File Descriptors," later in this chapter to learn how to do this.) Then, move the information that you want to pass to the incoming job into locations 510 to 777 (octal). You should use the .PEEK and .POKE programmed requests to access locations in the chain area. In the following example, the code moves the data from BUFF into the chain area. The format of the file specification is explained later.

EXAMPLE

```
        MOV     #BUFF,R1          ;Buffer address to R1
        MOV     #500,R2           ;Start of chain area
10$:    .POKE   #AREA,R2,(R1)     ;Put word of data from
        TST     (R2)+             ;Buff into chain area
        TST     (R1)+             ;Increment pointer
        BNE     10$               ;Back if not done yet
           .
           .
           .
```

```
BUFF:    .RAD50  /DK FRED   SAV/ ;RAD50 file spec for
                                 ;DK:FRED.SAV
         .WORD   /1,2,3,4,...,0/ ;Data for next job
```

Finally, issue the .CHAIN request, as follows:

```
         .CHAIN                  ;Chain to next program
```

In the incoming job, test bit 8 of the JSW (the chain bit). It will have been set only if the program has been successfully chained. In this case, retrieve the information passed by the outgoing job from locations 510 to 777 (octal). Use .PEEK to access this information.

The incoming job does not have to accept information passed by the outgoing job. If you want to set up the chain area with constant data in your program, you must set bit 8 in word 44 of block 0 in the program's save image file. If you do not set this bit and the chaining occurs, bytes 500 to 777 are saved from the job that issued the .CHAIN instead of being loaded from the save image file. To set bits or change other locations in your save image file, use a .ASECT directive in your source program.

EXAMPLE

```
         .ASECT          ;Word 44 (JSW)
         .=44
JSW:     .WORD    400    ;Set bit 8 to protect
                         ;chain area (all other
                         ;bits are cleared)
         .=500           ;Start of chain area
CAREA:      .
            .
            .
         .PSECT
```

This code sets bit 8 of word 44 (octal) and places information in the chain area to assure that these locations are loaded from the save image file. You can use the same method to set other bits, or a combination of bits, in the load image.

Chaining in FORTRAN IV Programs

To enable the outgoing job to chain to an incoming job and pass information to it, you should place the file specification (in RAD50 format) for the incoming job into a four-word area of the outgoing job. (Read the section, "Using RAD50 File Descriptors," later in this chapter to learn how to do this.) Then place the information (up to 60 words) that you want to pass to the incoming job, into an array or sequence of variables in a COMMON block. The first variable must start on a word boundary. Finally, call the CHAIN subroutine. The format is:

```
CALL CHAIN (dblk,var,wcnt)
```

In this statement:

dblk   is the address of the area containing the device and file specification of the incoming job

var   is the address of the first variable containing the information to be passed

wcnt   is the number of words of information (beginning at var) to be passed, which must not exceed 60

In the incoming job, call the RCHAIN subroutine to see if this job has been chained to the outgoing job. The format is:

```
CALL RCHAIN (flag,var,wcnt)
```

If it has been chained, the integer variable "flag" is set to −1 (true). If not, "flag" is 0 (false).

If the program has been chained, RCHAIN also retrieves any information passed by the outgoing job. "var" is the address of the first of a sequence of variables where you want the information to be stored. You should set "wcnt" to the number of words to be moved.

Using RAD50 File Descriptors

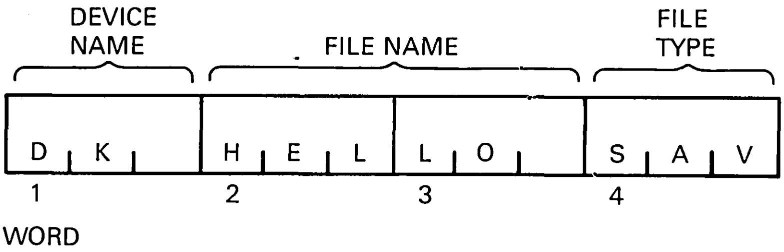
A MACRO–11 or FORTRAN IV program which chains to another job must specify the incoming job's file specification in a four-word area called the file descriptor block. This block contains the following information:

- The device type (3 characters)
- The file name (6 characters)
- The file type (3 characters)

This information must be stored in Radix 50 (RAD50) format. RAD50 code allows three characters to be stored in each word instead of the usual two. You can use any uppercase alphabetic characters, numbers, and spaces (<SPACE> characters) in the file descriptor block. Figure 31 shows an example of how the file specification DK:HELLO.SAV would be stored. Notice that each field must be extended to its full length, with trailing spaces if necessary. This file specification would be stored as:

DK<SPACE>	3 characters for the device name
HELLO<SPACE>	6 characters for the file name
SAV	3 characters for the file type

Figure 31.
RAD50 File Descriptor Block for DK:HELLO.SAV



If, at the time you write your program, you know the file specification for the job to which you want to chain, you can set up the file description as constant data. In MACRO–11 programs you can do this with the .RAD50 assembler directive. In FORTRAN IV programs you can use the R specifier in a DATA statement.

If, for example, you get the file specification at run time by reading it from the terminal, you will have to convert it from ASCII to RAD50 code and store it in the file descriptor block. Conversion routines are provided in the system subroutine library.

RAD50 in MACRO–11 Programs

The .RAD50 assembler directive encodes text in RAD50 format and reserves one word of storage for every three characters in the text string. If the text is not a multiple of three characters, the directive automatically appends trailing spaces. The format of the directive is:

```
FDB:   .RAD50   /text/
```

Here /(slash) may be any delimiter that does not appear in the text.

You can specify the complete file descriptor block in a single .RAD50 directive, or you can use a separate directive for each field.

EXAMPLE

You can create a file descriptor block for DK:HELLO.SAV in two ways:

```
FDB1:     .RAD50   /DK HELLO SAV/  ;File descriptor
                                   ;block
FDB2:     .RAD50   /DK/            ;Device name
          .RAD50   /HELLO/         ;File name
          .RAD50   /SAV/           ;File type
```

In the first method, you have to remember to include spaces in the text string to keep the correct word positions for the field. In the second method, the directive appends trailing spaces to the fields as necessary. If the file name is three characters or fewer, you must add enough spaces to make sure that two words of storage (not just one) are reserved.

RAD50 in FORTRAN IV Programs

You can store the file descriptor block in any set of four sequential words. You can use the single four-word variable (REAL*8, COMPLEX, or DOUBLE PRECISION) that is easiest to initialize. If you want to be able to refer to each word separately, use a four-element INTEGER*2 array.

To store the file descriptor information in RAD50 code, use the R format specifier in a DATA statement.

EXAMPLE

```
REAL*8 FDB
DATA FDB/12RDK HELLO SAV/
```

If you want, you can define each field separately.

EXAMPLE

```
INTEGER*2 IFDBLK(4)
DATA IFDBLK/2RDK,3RHEL,2RLO,3RSAV/
```

In the first example, you must include spaces as necessary so that each field occupies its full length. In the second example, you must define each element of the array separately. In either case, if the text string is not a multiple of three characters, the compiler appends enough trailing spaces to make it so.

Practice 10–1

1. If you are programming in MACRO–11, create the following programs:

Program 1 (PR1001.MAC)

```
        .TITLE  PR1001
;
;  PR1001       Prompt the user to enter a command,
;               read it, and exit preventing the
;               program from being REENTERed.
;
        .MCALL .GTLIN .EXIT
        .ENABL LC
MSGBFR: .BLKB  83.             ;Input text buffer
PROMPT: .ASCII "Enter command: "<200>
        .EVEN
START:  .GTLIN #MSGBFR,#PROMPT ;Prompt for and get
                               ;input string
        CLR    R0              ;Clear R0 for hard
        .EXIT                  ;exit and exit
        .END   START
```

Program 2 (PR1002.MAC)

```
        .TITLE  PR1002
;
; PR1002  Announce that program 2 has
;         started and then exit.
;
;
        .MCALL  .PRINT .EXIT
        .ENABL  LC
HELLO:  .ASCIZ  "HI THERE! THIS IS PROGRAM 2."
        .EVEN
START:  .PRINT  #HELLO    ;Display message
        .EXIT             ;and exit
        .END    START
```

Tailor the programs to use the extended memory features under the XM monitor. Use the .PEEK and .POKE programmed requests when you need to gain access to absolute locations in memory.

2. Assemble, link, and execute program 1 at your terminal. The system prompts you to type a monitor command; you can respond by typing a command like:

   ```
   DATE 01-JUN-84
   ```

   Try to reenter the program. KMON should give you an error message because the program, as written, cannot be reentered.

3. Copy program 1 into a file called PR1003.MAC. Modify the program in PR1003 so that it can be reentered and, on termination, displays a message reminding you to reenter the program. When you reenter, move the monitor command that you typed into the chain area and chain to program 2 (PR1002). Assemble and link program 2.

   Execute program 1 (PR1003) and type in a monitor command. The program should then stop. Reenter it, and check that you get a message announcing that program 2 has started.

4. Copy program 2 into a file called PR1004.MAC. Modify it so that it checks whether it has been chained. If it has not, it should issue an error message and stop. If it has, it should pick up the monitor command from the chain area, move it into the parameter area for KMON, and pass the command to KMON on exit.

5. Modify program 1 to chain to PR1004. Execute program 2 and check to see that it gives the error message. Now execute program 1 and reenter it. Check to see that the monitor command you typed in is passed to KMON by program 2 and executed correctly.

SYSLIB Routines for RAD50 Conversion

If you do not know the file specification until run time, you will have to convert it from ASCII to RAD50 code using the conversion routines available in SYSLIB. These conversion routines are available to both MACRO–11 and FORTRAN IV programmers:

- IRAD50 converts a specified number of ASCII characters to RAD50
- RAD50 converts six ASCII characters to RAD50

Practice 10–2

1. If you are programming in FORTRAN IV, create the following programs:

Program 1 (PR1001.FOR)

```
      PROGRAM PR1001
C
C     Prompt the user to enter a monitor command
C     and then exit.
C
      BYTE MSGBFR(80)
      TYPE 100                  ! Ask for a command
100   FORMAT (1H$,'Enter command: ')
      ACCEPT 101,MSGBFR         ! Read command line
101   FORMAT (80A1)
      STOP 'END OF PROGRAM'     ! Exit with mesaage
      END
```

Program 2 (PR1002.FOR)

```
      PROGRAM PR1002
C
C     Announce that program 2 has started and then
C     exit without a message.
C
      TYPE 100
100   FORMAT (1H ,'HI THERE! THIS IS PROGRAM 2.')
      CALL EXIT
      END
```

2. Compile, link, and execute program 1 at your terminal. It prompts you to type a monitor command; you can respond by typing a command like:

```
DATE 01-JUN-84
```

Notice the message displayed when the program terminates. Now modify the program so that it does not display any termination messages.

3. Copy program 1 to PR1003.FOR and modify it so that after it reads the monitor command, it chains to program 2 (PR1002.FOR) passing the monitor command in the chain area. Compile and link program 2.

   Execute program 1 and type in a monitor command. Check to see that you get a message announcing that program 2 has started.

4. Copy program 2 to PR1004.FOR and modify it so that it checks to see if it has been chained. If it has not, issue an error message and stop. If it has, pick up the monitor command from the chain area, pass the command to KMON, and then stop the program.

   Execute program 2 and check to see that it gives the error message. Now modify program 1 to chain to PR1004. Execute program 1 and check to see that the monitor command you type in is passed to KMON by program 2 and executed correctly.

References

RT-11 Programmer's Reference Manual. Chapter 2 discusses the .EXIT and .CHAIN programmed requests. Chapter 3 covers the SETCMD requests, CHAIN and RCHAIN subroutines, and RAD50 and IRAD50 conversion routines.

RT-11 Software Support Manual. Chapter 2 describes bit 8 of JSW.

RT-11 System User's Guide. Chapter 4 describes monitor commands.

11

11

Using Input/Output Systems

Almost all programs interact in some way with one or more peripheral devices. This interaction is normally for the purpose of inputting or outputting data. Support for input/output (I/O) operations in RT-11 is provided by means of programmed requests which allow you to perform I/O operations in one of three modes—synchronous, asynchronous, or event driven.

Synchronous I/O is processed in sequence with the program; that is, the program must wait for the I/O operation to be completed before it can continue. Asynchronous I/O, one form of nonsynchronous I/O, is processed independent of the program; that is, the program can continue without waiting for I/O completion. When the program needs the results of the I/O request, it must wait. Event-driven I/O, the other form of nonsynchronous I/O, enables you to specify a routine to be executed when the I/O is completed.

This chapter discusses these modes of performing I/O as well as the support for terminal, file, and queued I/O. It also describes the FORTRAN IV object time system (OTS) I/O support. The discussion is designed to help you select which form of I/O to use in your programs.

The MACRO-11 programmed requests introduced in this chapter are: .TTYIN, .TTINR, .TTYOUT, .TTOUTR, .PRINT, .GTLIN, .READ, .READW, .READC, .WRITE, .WRITW, .WRITC, .FETCH, .ENTER, .LOOKUP, .CLOSE, .PURGE, .RELEAS, .CSIGEN, .CSISPC, and .SPFUN. The FORTRAN IV requests introduced are: ITTINR, ITTOUR, PRINT, GTLIN, IREAD, IREADW, IREADC, IREADF, IWRITE, IWRITW, IWRITC, IWRITF, IFETCH, IGETC, IENTER, LOOKUP, CLOSEC, PURGE, IFREEC, ICSI, ISPFN, ISPFNW, ISPFNC, and ISPFNF.

I/O Operations

Almost every program has to perform some input or output operation; I/O operations are among the services most frequently used. The I/O method you choose affects both the ease of program writing and the execution speed.

I/O operations are divided into two classes, synchronous and nonsynchronous. Synchronous I/O is executed serially in a program; that is, when the program issues an I/O request, it must wait until the I/O operation has been completed before it can continue processing. Nonsynchronous I/O operations execute in parallel with the program, which is more efficient because the CPU can continue processing while a device is performing an I/O transfer. Thus, I/O and CPU processing can overlap, decreasing the total processing time.

Figure 32 shows a program using synchronous I/O. The numbers indicate the following events:

1 The program issues a read request and stops processing. The system processes the read request.

2 When data is available the read operation is completed and control returns to the user program.

3 The program issues a write request and stops processing. The system processes the write request.

4 The write operation is completed and control returns to the user program.

Figure 32.
Synchronous Processing

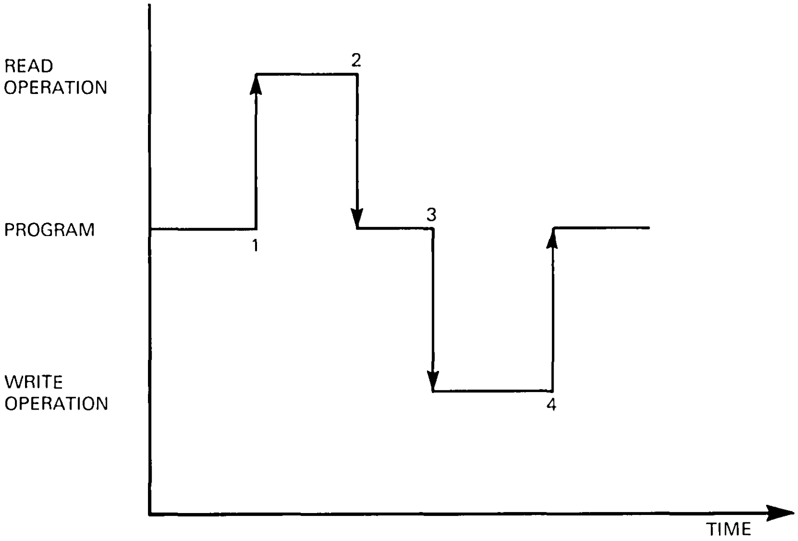


Figure 33 shows a program using nonsynchronous I/O. The numbers indicate the following events:

1 The program issues a read request and continues processing. The system processes the read request in parallel.

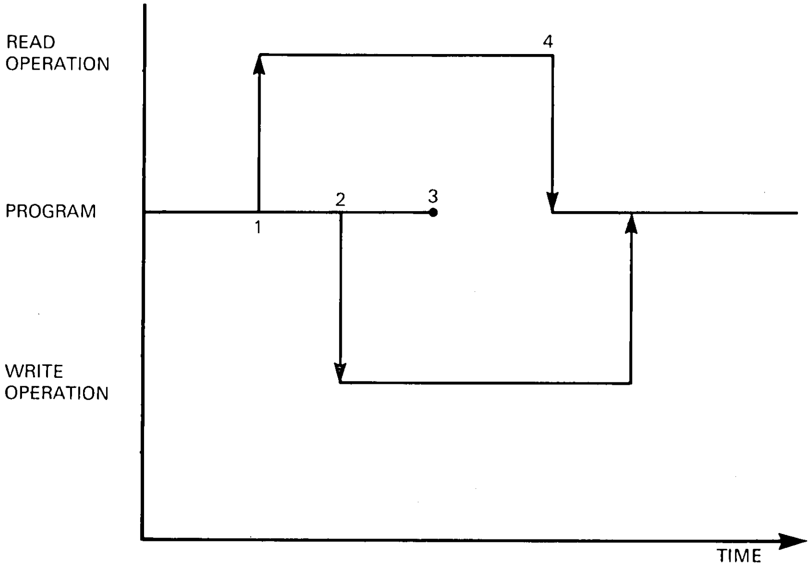
2 The program issues a write request and continues processing. The system processes the write request in parallel to the program and the read request.

3 The program cannot continue until the read is completed, so it waits for I/O completion of the read request.

4 The read is completed and the system informs the program, which continues.

The processing continues, with the program issuing nonsynchronous I/O requests and being informed of I/O completions as they occur.

Figure 33.
Nonsynchronous Processing



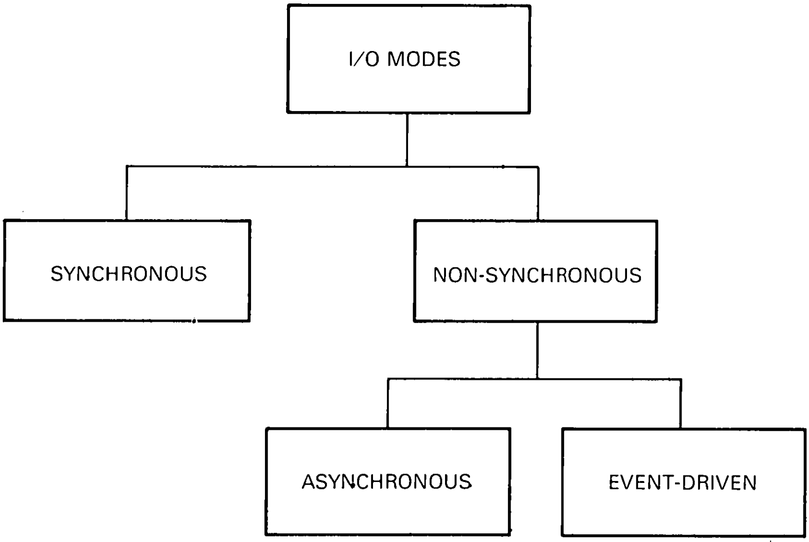
I/O Modes

RT–11 programmed requests support both synchronous and nonsynchronous I/O operations. Nonsynchronous operations can be further divided into asynchronous and event-driven modes. Thus, the two classes of RT–11 I/O operations include three different modes. These modes are related as shown in figure 34. Using these processing modes you can maximize performance under many different conditions.

Synchronous mode

When a synchronous I/O operation is requested, program execution is suspended until the I/O transfer is completed. This is the easiest mode to program, but it is the most inefficient method because there is no overlap of CPU and I/O processing.

Figure 34.
RT–11 Input/Output Modes



Asynchronous mode

When an asynchronous I/O operation is requested, program execution is suspended only until the monitor has queued the request. The program then continues execution, possibly before the I/O has been completed. It continues processing until it needs to synchronize with the I/O operation, for example, to use the data received. At this time the program must test for I/O completion with the .WAIT request.

Event-driven mode

When a program requests an event-driven I/O operation, the monitor suspends program execution until the request has been queued. The program then continues executing, possibly before the I/O has been completed. When the I/O is completed, the monitor interrupts the main program and executes a completion routine, specified in the original I/O request. This routine can be used to perform any processing that had to wait for I/O completion.

The event-driven mode is the most efficient method of overlapping I/O and CPU processing because the program does not have to test for I/O completion before executing code that must be synchronized with the I/O transfer. It is, however, the most difficult to implement.

Selecting an I/O Mode

The careful selection of I/O modes will help you achieve high performance from a program that includes large amounts of both I/O operations and CPU processing. Nonsynchronous I/O is most effective when the program is divided into a number of processes. For example, reading a buffer, analyzing the data, and writing a buffer can each be a separate process. These processes can run concurrently, but they need to be synchronized. The program must not try to analyze the data before it has been read into the buffer. Throughput is most efficient when there is a maximum overlap of process activity.

The following suggestions should help you select the best I/O mode to use for your applications programs:

1. If performance is not important, use synchronous mode because it is the simplest to program.
2. If one process needs much more time than the others, use synchronous mode because the potential overlap is small.
3. If system performance is important and the task can be divided into a number of processes whose timing is random, use event-driven mode. This gives the most efficient overlap of I/O and processing.
4. If the time required for a process is comparatively long, use event-driven I/O. For example, consider an application with a number of input devices. The application software is designed to accept input from any of them, process the data, and output the results

to a data base. At any given time, a device may have no operator or may not be working. The system must not hang waiting for input from any of the devices, so event-driven I/O is the only practical choice.

5. If none of the above apply, consider using asynchronous mode to overlap the longest processes. The I/O and .WAIT requests should be arranged to keep these processes as active as possible.

Terminal I/O

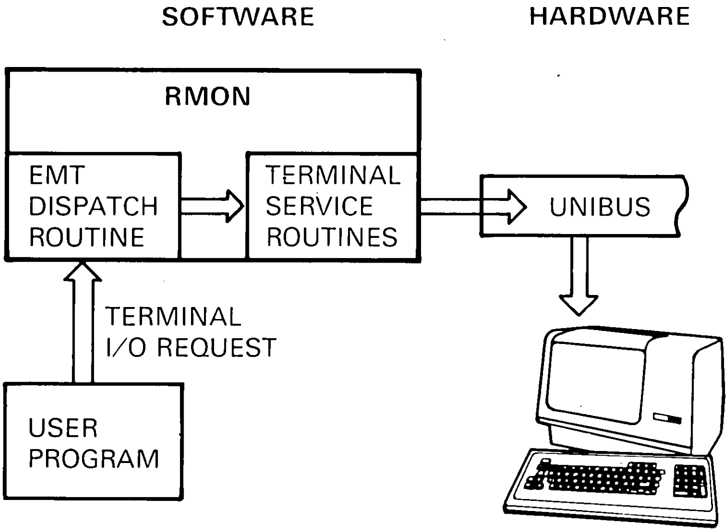
The console terminal is the most frequently used peripheral device because almost all programs send and receive data through it. RT–11 has a set of programmed requests developed expressly for console terminal I/O. These requests are listed in table 8. The system code for all these requests (except .GTLIN or GTLIN) resides within RMON. (The .GTLIN or GTLIN requests require the USR.) This means that terminal I/O does not need any disk access to load additional system software.

The terminal I/O system structure is shown in figure 35. When a terminal I/O request is issued, the EMT dispatch routine passes the request to the terminal service

Table 8.
Terminal I/O Requests

Operation	MACRO-11 Requests	FORTRAN IV Requests
Input character from terminal	.TTYIN .TTINR	ITTINR
Output character to terminal	.TTYOUT .TTOUTR	ITTOUR
Input line from terminal	.GTLIN	GTLIN
Output line to terminal	.PRINT	PRINT

Figure 35.
Processing Terminal Input/Output Requests



routines. These routines gain access to the terminal device registers directly.

When your program needs to communicate with the terminal, first consider using the terminal I/O requests since they are fast and flexible. Also, they do not greatly increase the size of your program because the code needed to execute them is always in memory, as part of RMON. The disadvantages of using the terminal I/O requests are:

- The requests are device specific. If you later decide to send your output to a line printer, you must rewrite part of your program. If you want to be able to change the I/O device easily, use one of the other I/O systems.
- Terminal I/O transmits data in character format only. Your program must perform any conversions, for example, binary to ASCII. If a FORTRAN IV program is to send out numeric data, you should consider using I/O statements in FORTRAN IV format.

Table 8 lists a complete set of programmed requests that permit multiterminal I/O. These requests are discussed in chapter 13, "Using Multiterminal Input/Output."

Queued I/O

RT–11 provides a standard programming interface for accessing the supported devices (such as storage disks and printers) and for special programs (called device handlers) which control each device. The interface takes I/O requests from user programs and puts them on a queue to be processed by the appropriate device handler. This process is called queued I/O. The requests used to perform input and output through queued I/O are listed in table 9.

The system device handler and system code reside in memory under all three monitors. The handler for the console terminal is resident in memory under the FB and XM monitors. Under the SJ monitor, it resides on the system disk and is loaded by the USR when needed. To decrease memory requirements, all other device handlers reside on the system disk and are loaded into memory when needed.

The common code needed to transfer information between a peripheral and a running program is in the Resi-

Table 9.
Queued I/O Requests

Operation	MACRO-11 Requests	FORTRAN IV Requests
Input from peripheral device	.READW .READ .READC	IREADW IREAD IREADC IREADF
Output to peripheral device	.WRITW .WRITE .WRITC	IWRITW IWRITE IWRITC IWRITF

dent Monitor RMON. When a queued I/O request is issued, RMON passes this request on to the appropriate device handler, using a data structure called an I/O queue element. Figure 36 shows the flow for queued I/O requests.

The flexible options of the queued I/O system make it the logical choice for I/O to most devices. As a programmer, you do not need to know very much about the device being used. The programs written using queued I/O can be device independent, and you can determine the actual device to be called at execution time.

File I/O

Certain set-up operations are required before you perform queued I/O to a file, and some clean-up operations are also required once I/O is completed. Table 10 shows the sequence of operations for MACRO-11 and FORTRAN IV files.

These requests need the file and device specifications in a device block in RAD50 format (see chapter 10). Pro-

Figure 36.
Processing Queued Input/Output Requests

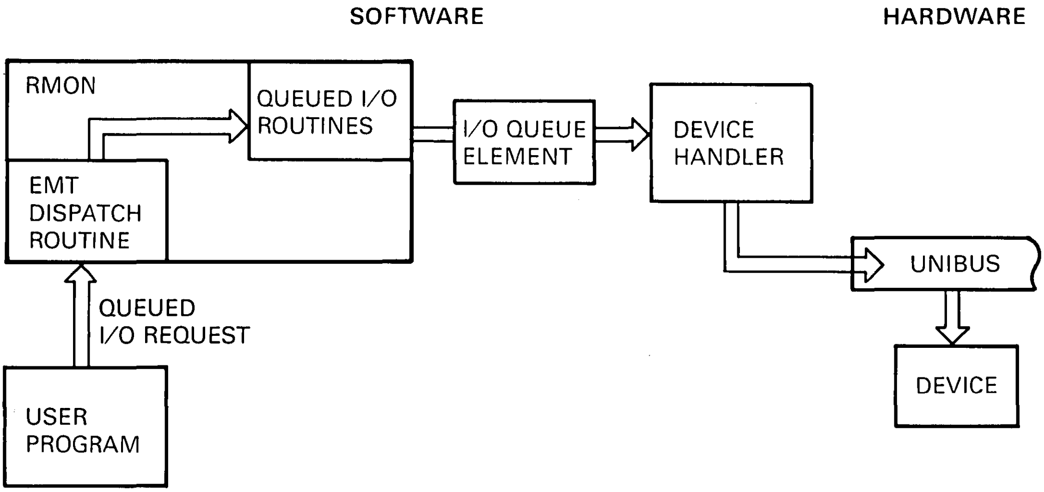


Table 10.
Sequence of Operations and Requests for File I/O

Operation	MACRO-11 Request	FORTRAN IV Request
1. Load the device handler	.FETCH	IFETCH
2. Allocate a channel to a device		IGETC
3. Open the channel	.ENTER	IENTER
	.LOOKUP	LOOKUP
4. Perform I/O	.READ	IREAD
	.WRITE	IWRITE
5. Close or purge the channel	.CLOSE	CLOSEC
	.PURGE	PURGE
6. Free the channel		IFREEC
7. Release the handler	.RELEAS	

grams that need to read ASCII file specifications from the console terminal or indirect command files can use the Command String Interpreter (CSI). Table 11 lists the requests used to process an ASCII string in CSI format. These set-up and clean-up requests are executed by the USR monitor component. The USR may be either memory or disk resident. Swapping the USR is covered in chapter 18, "Using Memory."

Table 11.
Requests for Processing ASCII Strings in CSI Format

Operation	MACRO-11 Request	FORTRAN IV Request
Load handlers and open channels	.CSIGEN	
Return RAD50 device blocks	.CSISPC	ICSI

Special Function I/O

RT–11 provides a special function I/O request that permits a program to perform device-dependent operations in all three I/O modes, on magnetic tapes and on some disks.

If you are programming in MACRO–11, you should use the request .SPFUN and specify the I/O mode in an argument to the macro call. If you are programming in FORTRAN IV, you can use one of four functions, ISPFN, ISPFNW, ISPFNC, or ISPFNF, for each of the different I/O modes.

FORTRAN IV OTS I/O

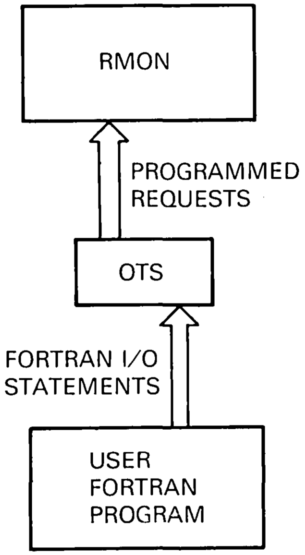
The FORTRAN IV language provides support for I/O through the statements, READ, WRITE, TYPE, ACCEPT, and PRINT. Because they are a defined part of the language, with no special subprogram calls needed, the following features allow for easy handling of data input and output:

- The division of I/O information into records
- Automatic translation between character and internal representation using formatted I/O
- Direct or random access I/O

Most common I/O operations to the terminal and other devices can be performed using these statements. However, these statements do not provide all of the capabilities of the RT–11 programmed requests. Because of the way in which the FORTRAN IV OTS performs I/O, the use of programmed requests can increase the execution speed and decrease the size of your program. The OTS I/O mechanism is shown in figure 37.

When a FORTRAN IV program executes an I/O statement, it calls some OTS routines. The OTS responds by executing RT–11 programmed requests. These may be terminal I/O or queued I/O requests, depending on the device

Figure 37.
Performing Input/Output through FORTRAN IV OTS



assigned. The OTS does not perform direct I/O itself but passes all requests to RT–11.

If your program uses programmed requests, you are able to bypass the I/O routines, communicating directly with RT–11. This reduces the time required to process OTS I/O routines, so your program should execute faster. If you remove all uses of one or more classes of FORTRAN IV I/O (for example, all formatted or direct I/O statements), the corresponding OTS routines will not be linked with your program, making it smaller.

Reference

RT–11 Programmer's Reference Manual. Chapter 1 discusses input/output operations, terminal input, output, and multiterminal requests.

12

12

Using Terminal Input/Output

In most applications, you interact with a program through the console terminal. The program sends messages, questions, warnings, and prompts to the terminal and receives responses to questions, choices of options, and commands from the terminal.

Information is transferred in both directions as a series of characters. Digital's systems use the seven-bit version of the American Standard Code for Information Interchange (ASCII). Most ASCII characters are standard printing characters such as numbers, letters, and punctuation marks. Some special nonprinting characters are used for controlling the terminal and sending special signals to the processor.

The basic operations of terminal I/O are single-character input and output. There are also line-oriented I/O functions in which a whole line is transferred by one programmed request. Other features offered by RT–11 for terminal I/O include the ability to enable or suppress echo printing and handle the special characters ⟨CTRL/C⟩ *and* ⟨CTRL/O⟩.

This chapter describes how to use the programmed requests: .TTINR, .TTYIN, .TTOUTR, .TTYOUT, .PRINT, .GTLIN, .SCCA, and .RCTRLO for MACRO–11 and ITTINR, ITTOUR, PRINT, GTLIN, SCCA, and RCTRLO for FORTRAN IV.

When you have completed this chapter, you will be able to write code to transfer a line of text from the terminal to a buffer in memory, using the single-character input requests .TTYIN for MACRO–11 and ITTINR for FORTRAN IV. You will also learn to perform the following functions: input text from the terminal; print text from a buffer in memory on a terminal, using the output requests .TTYOUT and .PRINT for MACRO–11 or ITTOUR and PRINT for FORTRAN IV; prevent ⟨CTRL/C⟩ *from aborting a program; and reset* ⟨CTRL/O⟩ *under program control.*

Terminal I/O Buffers

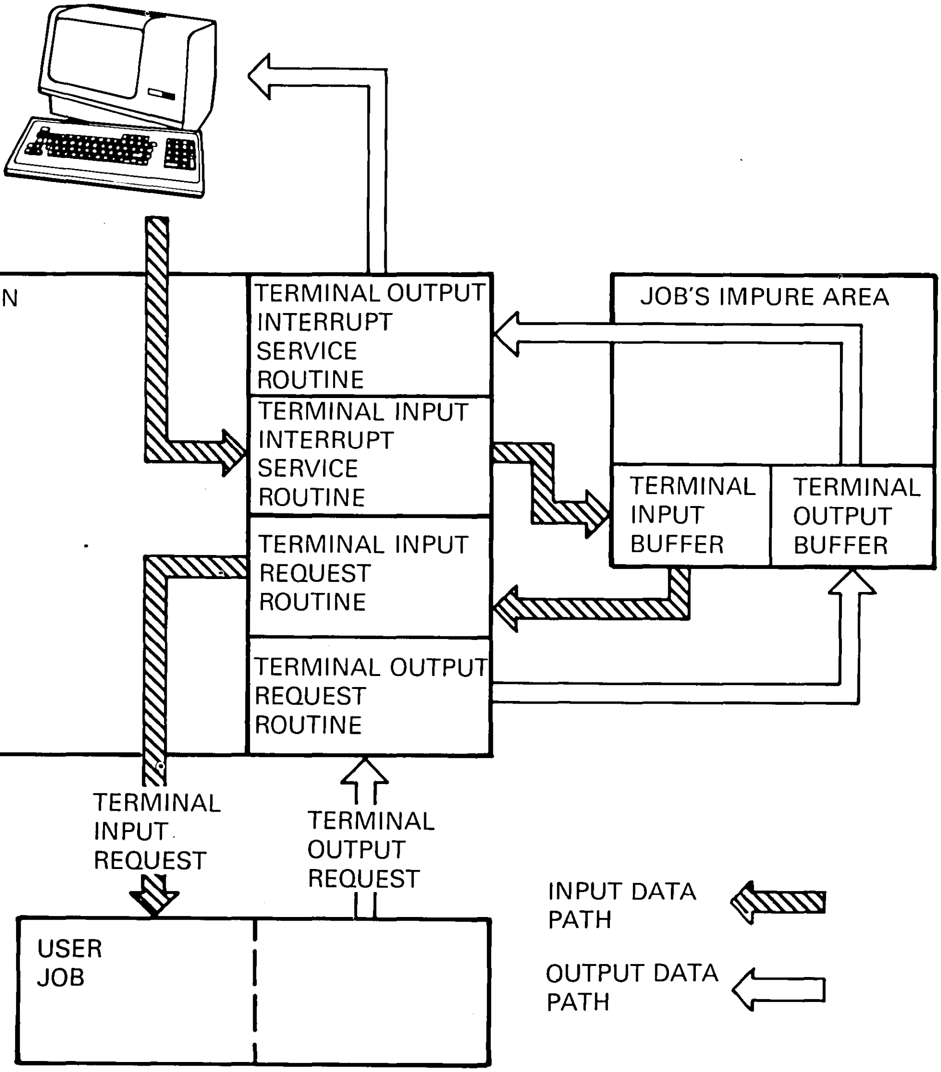
Terminal I/O in RT–11 is performed through a set of buffers known as the terminal I/O buffers. Each job has two buffers, one for input and one for output, both located in the job's impure area. By using the terminal I/O requests, you can transfer characters between a program and its terminal I/O buffers.

Interrupt service routines in RMON handle the actual character transfer between the I/O buffers and the terminal. By handling the transfer on an interrupt basis, RMON allows terminal I/O to run in parallel with the user program. On input, parallel processing means that you can type in characters before the program asks for them. This is known as type-ahead. On output, parallel processing means that the program can issue a number of output requests and can continue running while the characters are being printed. Thus, the program can run more quickly since it does not need to wait for all the characters to be printed before it proceeds.

Normally RMON does not make input characters available to your program until you type a line-terminating character, such as ⟨RETURN⟩, ⟨LINEFEED⟩, ⟨CTRL/Z⟩, or in some cases ⟨CTRL/C⟩. This means that the characters are held in a buffer, and you can correct typing errors, using the line-editing characters ⟨DELETE⟩ and ⟨CTRL/U⟩.

The internal structure of the terminal I/O system is shown in figure 38. The arrows indicate the transfer of a character between the terminal and the program. Notice that the path for input is independent of the path for output.

Figure 38.
Internal Structure of the Terminal Input/Output System



Sp.ecial Characters

Certain characters are not passed through the terminal input buffer by the terminal service routines. These characters include:

- Line-editing characters ⟨DELETE⟩ and ⟨CTRL/U⟩
- Terminal control characters ⟨CTRL/O⟩, ⟨CTRL/S⟩, and ⟨CTRL/Q⟩
- Characters to direct terminal input under the FB and XM monitors ⟨CTRL/F⟩ and ⟨CTRL/B⟩
- Characters for use when system job support is enabled ⟨CTRL/X⟩
- Interrupt character ⟨CTRL/C⟩

By default, all of these are handled by RMON. Techniques for inhibiting ⟨CTRL/C⟩ and resetting ⟨CTRL/O⟩ are discussed later in this chapter.

Another input character that is handled specially is the carriage return. When you press ⟨RETURN⟩, the input terminal service routine always inserts both a carriage return and a line feed into the terminal input buffer.

Single-Character I/O

Some of the programmed requests supported by RT–11 for terminal I/O will only operate on a single character. These requests can be used to retrieve one character from the terminal input buffer or add one character to the terminal output buffer.

MACRO–11 Requests: .TTINR, .TTOUTR, .TTYIN, and .TTYOUT

.TTINR moves a single ASCII character from the terminal input buffer to the low-order byte of register R0. .TTOUTR moves a single character from the low-order byte of R0 into the terminal output buffer. .TTINR and .TTOUTR do not take any arguments.

These requests each have a single possible error return. .TTINR returns an error if no character is available in the terminal input buffer. .TTOUTR returns an error if there is no room in the terminal output buffer for the character. The purpose of these two requests is to attempt the I/O operation, and return an error immediately if the operation is not possible at the time of the request.

The more common form of terminal I/O is the use of .TTYIN and .TTYOUT. These requests are similar to .TTINR and .TTOUTR, except that they loop, repeating the request until a terminator has been typed and the operation can be completed. These requests take an optional argument; for .TTYIN, the address of where to put the character, and for .TTYOUT, the address of the character to be output.

FORTRAN IV Requests: ITTINR and ITTOUR

ITTINR gets a character from the terminal input buffer and returns the ASCII code for the character as its function code. A negative value indicates that no character is available in the terminal input buffer. The function is called using the format:

```
ICHAR=ITTINR()
```

There are no arguments to ITTINR, but the parentheses are required so the compiler can tell that ITTINR is a function reference, not a variable reference.

The result of ITTINR must first be placed in an INTEGER variable to check for an error return. If there is no error, you can transfer it to a BYTE (or LOGICAL*1) variable. This sequence is necessary because for negative numbers the PDP–11 sets the highest bit (bit 15) to 1. To test whether the result of ITTINR is negative, the processor tests bit 15 of the returned variable. If you place the result in a BYTE variable, only the low-order eight bits are stored. The high-order bits, including bit 15, are lost. If you then try to test for errors by seeing whether the BYTE variable is negative, you will receive incorrect results.

The following example shows the correct use of ITTINR and the test of its returned value:

EXAMPLE

```
        BYTE CHRTRS(80)
        INTEGER RETVAL
           .
           .
        DO 10, I=1,80
100     RETVAL=ITTINR()
        IF (RETVAL.LT.0) GO TO 100
        CHRTRS(I)=RETVAL
10      CONTINUE
           .
           .
```

ITTOUR is the programmed request to output single characters. It takes a single-byte variable as an argument and transfers the character in that byte to the terminal output buffer. A return value of one from ITTOUR indicates an error, namely, that there is no room in the terminal output buffer for the character.

Error Handling under the SJ Monitor

The SJ and FB monitors handle terminal I/O error conditions differently. The SJ monitor always returns an error code to the user program if an error condition exists. On input the only error is "no character available in the buffer," and on output the only error is "no room in the buffer." Your program must check the error indicator and take appropriate action. Usually the appropriate action is to retry the request by looping again until the transfer is successful. In

MACRO–11 programs you can do this by using the .TTINR or .TTOUR request.

EXAMPLE

```
        .MCALL  .TTINR
           .
           .
1$:     .TTINR          ;Try to read a character
        BCS     1$      ;Repeat if none ready
           .
           .            ;Continue when success
```

The programmed requests .TTYIN and .TTYOUT perform the looping for you. Neither proceeds until the I/O request has executed error-free. Also, with these requests you can give an argument that specifies the location of the data to be transferred. You should use .TTYIN and .TTYOUT unless you want to do some other processing before retrying the request.

In FORTRAN IV programs you can test for errors and retry the request by examining the return value from the ITTINR and ITTOUR functions, and by looping if an error exists.

EXAMPLE

```
5       ICHAR=ITTINR()
        IF (ICHAR.LT.0) GO TO 5
```

or

```
10      IF (ITTOUR(CHAR).NE.0) GO TO 10
```

Always use a loop like one of these if you do not want your program to proceed until the I/O request has executed successfully.

Error Handling under the FB Monitor

Normally, when you issue a terminal I/O request, you do not want the program to continue before the transfer is complete. Under the SJ monitor you can ensure completion of transfer by looping to repeat the request until it returns a success indication. Looping works well for a system like SJ where only one job can run at a time. Under the FB monitor, however, you can run two jobs at a time. When one job is waiting for I/O completion, the other can be allowed to run. If the I/O request loops, as under SJ, then the monitor cannot tell that it could be running the other job. Therefore, the FB monitor uses a different technique for handling I/O requests.

A job waiting for some specific I/O condition to occur is said to be blocked. When a job is blocked the FB monitor switches control to a job that can run. Jobs are assigned relative priorities, and FB always selects the highest priority job that is not blocked.

The terminal I/O service routines under FB automatically block a job that issues a terminal I/O request if the request cannot be completed immediately. The job remains blocked until the condition is cleared and the FB monitor can schedule it to run.

One effect of this blocking technique is that the .TTINR or ITTINR and .TTOUTR or ITTOUR requests do not return an error code. The job is blocked until the transfer is complete. This means that under the FB monitor .TTINR behaves the same as .TTYIN, and .TTOUTR behaves the same as .TTYOUT. If you use .TTINR or ITTINR or .TTOUTR or ITTOUR, you should check for the error code to ensure that the program will run correctly, even under the SJ monitor.

Overriding Job Blocking under the FB Monitor

You may not want your job to be blocked by the monitor when you issue a .TTINR/ITTINR or .TTOUR/ITTOUR programmed request. If you want to get an error code and program your own response, as under the SJ monitor, set bit 6 of the job status word (word 44 in the system communication area) before issuing the I/O request.

EXAMPLE

In FORTRAN IV use the command:

```
CALL IPOKE ("44,IPEEK("44).OR."100)
```

In MACRO–11 use the statement:

```
BIS      #100,@#44   ;Set bit 6 of JSW
```

Leave bit 6 of the JSW set only as long as you need to; then clear it.

EXAMPLE

In FORTRAN IV use the command:

```
CALL IPOKE ("44,IPEEK("44).AND..NOT."100)
```

In MACRO–11 use the statement:

```
BIC      #100,@#44   ;Clear bit 6 of JSW
```

Do not use this technique if you intend to loop (or in MACRO–11 use .TTYIN or .TTYOUT), because it prevents any background job from running.

You should reset ⟨CTRL/O⟩ after setting or clearing any bits in the JSW, and before issuing the first terminal I/O request. This forces the monitor to update all the terminal data structures with the new status. The programmed request to reset ⟨CTRL/O⟩ is discussed later in this chapter.

Table 12 shows the RT–11 processing schemes for terminal I/O when transfer of a character cannot take place. Error handling under the SJ monitor is not affected by the value of bit 6 of the JSW. Table 13 shows how to program terminal I/O, either to wait for completion of error process-

Table 12.
Terminal I/O Error Processing Schemes

Monitor	.TTYIN .TTYOUT	.TTINR .TTOUTR	ITTINR ITTOUR
SJ,FB,XM with bit 6 of JSW set	C-bit is set Program loops	C-bit is set Program continues processing Program must check for error	
FB,XM with bit 6 of JSW clear	Job blocked Control passes to lower priority job	Job blocked Control passes to lower priority job	

ing, or to allow the program to continue if error occurs. If you leave bit 6 of the JSW clear, the job will be blocked until I/O completion. You should still test for errors and loop to retry the request, so that the program can run correctly under the SJ monitor. You leave bit 6 set only when you

Table 13.
Techniques for Handling Terminal I/O Errors

Monitor	Language	Program Waits until Error Condition Is Cleared	Program Continues if an Error Occurs
SJ	MACRO-11	Use .TTYIN and .TTYOUT	Use .TTINR and .TTOUTR
	FORTRAN IV	Use ITTINR and ITTOUR; loop until error condition clears	Use ITTINR and ITTOUR
FB,XM	MACRO-11	Clear bit 6 of JSW; Use .TTYIN and .TTYOUT	Set bit 6 of JSW; use .TTINR and .TTOUTR
	FORTRAN IV	Clear bit 6 of JSW. use ITTINR and ITTOUR	Set bit 6 of JSW; use ITTINR and ITTOUR

need to continue processing, even if .TTINR and .TTOUTR or ITTINR and ITTOUR returns an error. You should then clear bit 6. Do not use a tight loop to handle errors with bit 6 set because that prevents other jobs from running. The techniques listed are for MACRO–11 and FORTRAN IV under the SJ and FB monitors.

Setting Up and Using I/O Buffers

When you transfer character strings, you normally store the text in buffers in your program. Input buffers do not need to be initialized. In MACRO–11 programs, use .BLKB to reserve space for an input buffer. In FORTRAN IV programs, you must use an array of data type BYTE or LOGICAL*1. The buffer must have enough space for the largest message the program expects to receive. If you overflow the buffer, you will corrupt your program code or data areas at run time.

Output buffers must be initialized before you output any data. You can either build a message at run time by moving text into the buffer, or define the text of a fixed message at compile time. The MACRO–11 requests .ASCII or .ASCIZ define text messages. .ASCIZ terminates the string with a byte containing binary zero (null byte). Some string manipulation routines in SYSLIB expect strings in the .ASCIZ format.

The FORTRAN IV DATA statement defines the contents of a LOGICAL*1 array. This statement is suitable only for short character strings. The string manipulation routines in SYSLIB make it easier to define strings, but the text has to be moved into the buffers at run time.

To illustrate the ways to use the terminal I/O requests discussed so far, we have chosen some common applications for you to study. The programs (PR1201.MAC and PR1201.FOR) which follow store multiple lines of input from the terminal. They use a null byte to terminate each line of text within the buffer, which means the carriage return and line feed characters are not stored. This saves one byte per line, and also means that the messages can be used in the SYSLIB string manipulation routines.

PR1201.MAC

```
        .TITLE  PR1201  -- TERMINAL I/O EXAMPLE
        .MCALL  .TTYIN  .TTYOUT .TTINR  .PEEK   .POKE
        .MCALL  .RCTRLO .EXIT
;       Data Defintions
AREA:   .BLKW                   ;EMT argument block
PROMPT: .ASCIZ  "Enter messages: "
MSGBFR: .BLKB   100.            ;Input buffer
        .EVEN
;       Program Code
START:  MOV     #PROMPT,R1      ;Point to prompt buffer
10$:    .TTYOUT (R1)+           ;Print (w/wait) 1 char
        TSTB    (R1)            ;End of message?
        BNE     10$             ;Loop if not
        MOV     #MSGBFR,R1      ;Point to input buffer
        MOV     #100.,R2        ;Load maximum char count
GET:    .TTYIN  (R1)+           ;Read (w/wait) 1 char
        DEC     R2              ;Decrement char count
        BEQ     OFLO            ;Branch if buffer full
        CMPB    #15,R0          ;Was char a <CR>?
        BNE     GET             ;Branch if not
LINE:   CLRB    -1(R1)          ;Yes, store null byte
        .TTYIN                  ;Get <LF> char
        TSTB    -2(R1)          ;Was last line blank?
        BNE     GET             ;Branch if not for more
        BR      PRINT           ;Otherwise, print buffer
;       Buffer full.  Flush terminal input buffer
OFLO:   CLRB    -(R1)           ;Append null bytes
        CLRB    -(R1)           ; to mark end of text
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R1           ;Move to R1
        BIS     #100,R1         ;Inhibit TT wait
        .POKE   #AREA,#44,R1    ;Update JSW
        .RCTRLO                 ;Reset Control/O
10$:    .TTINR                  ;Read (wo/wait) 1 char
        BCC     10$             ;Branch if char read
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R1           ;Move to R1
        BIC     #100,R1         ;Enable TT wait
        .POKE   #AREA,#44,R1    ;Update JSR
        .RCTRLO                 ;Reset Control/O
PRINT:  MOV     #MSGBFR,R1      ;Load buffer address
10$:    .TTYOUT (R1)+           ;Print (w/wait) 1 char
        TSTB    (R1)            ;Is next byte null?
        BNE     10$             ;Branch if not
        .TTYOUT #15             ;Otherwise print <CR>
        .TTYOUT #12             ; and <LF>
        INC     R1              ;Skip over null byte
        TSTB    (R1)            ;Is next byte null?
        BNE     10$             ;Branch if not
        .EXIT                   ;Otherwise, exit
        .END    START
```

PR1201.FOR

```
        PROGRAM PR1201
        BYTE MSGBFR(100),PROMPT(80) ! MESSAGE BUFFERS
        CALL SCOPY('ENTER MESSAGES: ',PROMPT)
C
C       Output prompt.
C
        DO 10,I=1,80
        IF (PROMPT(I) .EQ. 0) GO TO 20
5       IF (ITTOUR(PROMPT(I)) .NE. 0) GO TO 5
10      CONTINUE
C
C       Now input lines of text terminated by two <CR>s.
C
20      DO 40 I=1,100
25      IERR=ITTINR()           ! ACCEPT CHAR
        IF (IERR .LT. 0) GO TO 25 ! LOOP UNTIL READ
        MSGBFR(I)=IERR          ! STORE CHAR IN BUFFER
        IF (MSGBFR(I) .NE. "15) GO TO 40
        MSGBFR(I)=0             ! CHANGE <CR> TO NULL
30      IF (ITTINR() .LT. 0) GO TO 30
        IF (I .NE. 1 .AND. MSGBFR(I-1) .EQ. 0) GO TO 100
40      CONTINUE
C
C       Buffer overflowed (more than 100 chars typed)
C       Read & lose remaining chars in input buffer.
C
60      CALL IPOKE("44,IPEEK("44).OR."100)
        CALL RCTRLO             ! INHIBIT TT WAIT
65      IF (ITTINR() .GE. 0) GO TO 65
        CALL IPOKE("44,IPEEK("44) .AND. .NOT. "100)
        CALL RCTRLO             ! ENABLE TT WAIT
        MSGBFR(99)=0            ! ADD 2 NULLS TO BUFFER
        MSGBFR(100)=0
C
C       Now output messages entered.
C
100     DO 130 I=1,100
        IF (MSGBFR(I) .NE. 0) GO TO 110
102     IF (ITTOUR("15) .NE. 0) GO TO 102 ! OUTPUT <CR>
103     IF (ITTOUR("12) .NE. 0) GO TO 103 !  AND <LF>
        IF (MSGBFR(I+1) .EQ. 0) GO TO 150 ! STOP IF 2 NULLS
        GO TO 130
110     IERR=ITTOUR(MSGBFR(I))  ! OUTPUT A CHAR
        IF (IERR .NE. 0) GO TO 110 ! LOOP UNTIL OUTPUT
130     CONTINUE
150     CALL EXIT               ! EXIT
        END
```

As you study the MACRO–11 or FORTRAN IV program, look for:

- The input process
- Multiple-line buffering
- The response if too many characters are entered
- The procedure for printing multiple lines

The programs do not store carriage return or line feed. Carriage return is stored as a null byte, and line feed is discarded. The programs contain code to clear the system's terminal input buffer if the operator types in more than 100 characters. They set bit 6 of the JSW, so that an error indication is returned when there are no more characters in the buffer (all type-ahead has been cleared). They then loop, reading the buffer until they receive the error code. This indicates that the buffer is empty, and the programs clear bit 6 again. If they did not empty the buffer, the type-ahead characters would still exist, and would be read later by the system. KMON would then try to interpret the data as a monitor command.

Practice 12–1

In this exercise you will write programs to perform terminal I/O. You can write the programs either in MACRO–11 or FORTRAN IV, using the programmed requests discussed in this chapter. Write the programs so that they can be run under the FB or SJ monitor.

1. Write a program that prompts "PLEASE TYPE IN YOUR NAME." After the user types a name, your program should respond:

   ```
   WELCOME TO RT-11, User's name
   ```

2. Write a program that does the following:

   a. Prompts the user to input a message. The user should type a single line of text, terminated with a carriage return.

b. Loops, printing the user's text repeatedly on the same line. When the output reaches column 80, the program should output a carriage return and line feed and continue the output on the next line. The program repeats this until the user presses ⟨RETURN⟩, ⟨LINEFEED⟩, or ⟨CTRL/Z⟩. Other characters are ignored.

c. Repeats steps a. and b. until, during step a., the user types a blank line (responds to the prompt with ⟨RETURN⟩). Then the program exits.

Terminal Special Mode

When using .TTYIN, .TTINR, or ITTINR, I/O data is normally buffered until the line is terminated by ⟨RETURN⟩, ⟨LINEFEED⟩, ⟨CTRL/Z⟩, or ⟨CTRL/C⟩. You can edit a line using ⟨DELETE⟩ or ⟨CTRL/U⟩, and each character is echoed as typed with no special action performed by the user program. When the line terminator is received, all the characters in the buffer are passed to the program one at a time.

Under terminal special mode, characters are made available to a program as soon as they are typed in. There is no delay caused by waiting for a line terminator. In this mode the normal ⟨DELETE⟩ and ⟨CTRL/U⟩ actions are disabled. These characters are passed to the program, to be handled as you want. Terminal echo is also disabled under terminal special mode, except for ⟨CTRL/C⟩ and ⟨CTRL/O⟩. If you want the input echoed, you must code the output commands yourself.

Terminal special mode, which is enabled by setting bit 12 of JSW, is used for:

- Password entry, with echo suppressed.
- Single-character responses to program prompts. For example, Yes or No type questions can be answered with Y or N.

- Single-character unechoed input. For example, provide function keys to control the video display in the video editor KED.

Line-oriented Output

In addition to programmed requests that perform single-character terminal I/O, RT–11 supports I/O operations that transfer a whole line or message using a single request. For example, you can use the PRINT request to issue a prompt to the user, indicating what information is needed next. The PRINT request causes the contents of the specified buffer to be printed on the terminal. The last byte in the buffer must contain either 0 (NULL) or 200 (octal).

- If NULL, the monitor adds a carriage return and line feed to the end of the message.
- If 200 (octal), the monitor leaves the print head (or cursor on a display terminal) at the next character position after the last character printed.

If you include a carriage return and line feed in your message, they are printed, making the message multiline.

Using .PRINT in MACRO–11

The MACRO–11 programmed request has the format:

```
.PRINT   addr
```

In this format, "addr" is the address of the message buffer.

EXAMPLE

```
.PRINT  #MESS
    .
    .
    .
```

```
MESS:   .ASCII  /THIS IS THE FIRST LINE OF THE MESSAGE/
        .BYTE   15,12
        .ASCIZ  /THIS IS THE SECOND LINE OF THE MESSAGE/
```

.PRINT from a Foreground Job

If two jobs share a terminal and the foreground job issues a .PRINT request, the message is printed immediately. If the foreground job uses .TTYOUT or ITTOUR requests, the message is delayed until the background job finishes typing its current line. For this reason, you should use .PRINT rather than .TTYOUT or ITTOUR for critical messages from a foreground job.

When a foreground job and a background job are running, the job producing output is indicated by a B> or F> preceding the output. These markers are printed only when the job producing the output changes. When a system job prints a message, the logical job name is used as the identifier.

Using PRINT in FORTRAN IV

The FORTRAN IV request has the format:

CALL PRINT (message)

In this format, "message" can be a quoted string, passed as a single line, or an array containing characters, terminated with a null byte or 200 (octal). To place the value 200 (octal) in a string, you can use the SYSLIB subroutine CONCAT.

EXAMPLE

```
BYTE MESSAG(80)                        !MESSAGE ARRAY
CALL CONCAT ('TYPE IN A NUMBER: ', 200,MESSAG)
CALL PRINT (MESSAG)
```

Line-oriented Input

The GTLIN request inputs a complete line of text and stores it in the specified buffer in ASCIZ format. The maximum number of characters allowed for the input line is 80. You should, however, allocate 81 bytes to the buffer to make room for the trailing null byte. The GTLIN request uses the USR to input the line but does not check the syntax of the input text. The form of the MACRO-11 request is:

```
.GTLIN    addr
```

In this format, "addr" is the address of the input buffer. The FORTRAN IV call is:

```
CALL GTLIN(buffer)
```

Here, "buffer" is an array or variable.

EXAMPLE

```
LOGICAL*1 IOBUF(81)  !HOLDS UP TO 80 CHARACTERS
CALL GTLIN(IOBUF)
```

GTLIN has another form, which allows you to print a prompt on the terminal before reading the input line. The MACRO-11 request for this is:

```
.GTLIN    baddr,paddr
```

In this request, "baddr" is the address of the input buffer, and "paddr" is the address of the prompt buffer. The FORTRAN IV call is:

```
CALL GTLIN(BUFFER,PROMPT)
```

Each argument is an array or variable.

The GTLIN prompt string has the same format as the PRINT output string. A null byte at the end of the string causes a carriage return and line feed to be printed after the prompt. The value 200 (octal) at the end of the string leaves the print head (or cursor) on the same line. Prompts usually end with 200 (octal), so that the input is on the same line as the prompt.

GTLIN converts lowercase letters to uppercase, unless bit 14 of the JSW is set. This bit controls lowercase to uppercase conversion for terminal input requests.

Input from Indirect Command Files

The GTLIN request accepts data from an indirect command file if one is active. Otherwise, input is from the terminal. The .TTYIN requests can accept input from the terminal only. Thus, you should use the .TTYIN requests if you want to make sure that data arrives from the terminal.

If you use .GTLIN for input, and the program is run from an indirect command file, the input data must be taken into consideration as the command file is created. Lines of data can be included in the command file, to be read by the .GTLIN requests in the program. Sometimes you may want to switch the input data stream from the command file to the terminal. For example, to read a quantity of known data, which can be entered in a command file, and then ask the user to type in responses on-line, you must indicate in the program data that you want to switch the input to the terminal.

1. Set bit 3 of the job status word. This enables the input stream to be switched when requested. It does not switch the stream automatically.
2. Insert the characters ^C (a circumflex followed by a C) in the command file at the point where you want to enable data input from the terminal.

Normally, a job using .GTLIN to read data treats ^C in an indirect command file as if it were the ⟨CTRL/C⟩ character, and the job is aborted. But if bit 3 of the JSW is set, all further input for the job comes from the terminal. You cannot switch the input data stream back to the command file until the job terminates. If you clear bit 3 from the program, the next .GTLIN request aborts the job, as if ⟨CTRL/C⟩ had been typed.

If the monitor command SET TT NOQUIET is in effect, data read by .GTLIN from a command file is echoed on the terminal. If SET TT QUIET is specified, no echoing is performed.

Handling ⟨CTRL/C⟩

⟨CTRL/C⟩ is the normal way for you to return control to the RT–11 monitor from a program. Pressing ⟨CTRL/C⟩ once aborts the program the next time it requests terminal input, and pressing ⟨CTRL/C⟩ twice aborts the program immediately.

You can disable ⟨CTRL/C⟩ to prevent the user from aborting your program, but you should do so only for thoroughly debugged code. If you disable ⟨CTRL/C⟩ and the program goes into an infinite loop, the only way to stop it is to reboot the system. When ⟨CTRL/C⟩ is disabled, any single ⟨CTRL/C⟩ passes as an ASCII character (octal value 3). You disable ⟨CTRL/C⟩ using the .SCCA request as follows:

```
.SCCA   area, flagaddr      (MACRO–11)
CALL SCCA (FLAG)            (FORTRAN IV)
```

In MACRO–11, "area" is the address of a two-word parameter block, and "flagaddr" is the address of a terminal status word (flag word). In FORTRAN IV "FLAG" is an integer variable to be used as the flag word.

If ⟨CTRL/C⟩ is pressed twice, RMON sets bit 15 of the flag word. If you want to detect another double ⟨CTRL/C⟩, your program must clear this bit. To reset to normal ⟨CTRL/C⟩ action, use:

```
.SCCA   area,#0       (MACRO–11)
CALL SCCA             (FORTRAN IV)
```

Handling ⟨CTRL/O⟩

⟨CTRL/O⟩ inhibits output to the console terminal until another ⟨CTRL/O⟩ is received, or until the program resets the ⟨CTRL/O⟩ switch.

The program continues to be executed, but RMON does not perform terminal output requests while ⟨CTRL/O⟩ is in effect. All the output requests terminate successfully immediately, and the program executes much faster than if it had to wait for I/O completion.

⟨CTRL/O⟩ is most useful when you want to examine a small part of a long text file. You can suppress the output of sections that you do not need to see. In this way you can scan the file very rapidly.

When writing a program, you may want to be sure that data is printed on the console terminal. You can use the RCTRLO request to reset ⟨CTRL/O⟩ if it is in effect.

EXAMPLE

The MACRO–11 form of this request is:

```
.RCTRLO    ;Reset ⟨CTRL/O⟩
```

The FORTRAN IV form is:

```
CALL RCTRLO
```

When this request is called, printing on the console terminal is enabled regardless of the status of ⟨CTRL/O⟩. It is recommended that you issue a .RCTRLO or RCTRLO request after setting or clearing any bits in the JSW.

Practice 12–2

Modify the second program you wrote for practice 12–1 so that it uses the .GTLIN or GTLIN request to accept the initial input.

Use .PRINT or PRINT to output the text. Print each output message on a separate line. Continue until the user presses ⟨RETURN⟩, ⟨LINEFEED⟩, or ⟨CTRL/Z⟩, then reset ⟨CTRL/O⟩ and repeat the input prompt.

References

RT–11 Programmer's Reference Manual. Chapter 1 reviews the SYSLIB string manipulation routines for MACRO–11 and FORTRAN IV programs. Chapter 2 discusses the .GTLIN, .SCCA, .RCTRLO, .TTYIN, .TTINR, .TTYOUT, and .TTYOUTR requests in MACRO–11 programs. Chapter 3 describes the GTLIN, SCCA, RCTRLO, ITTINR, and ITTOUR requests in FORTRAN IV programs.

RT–11 Software Support Manual. Chapter 3 examines RT–11's terminal input/output system.

Multiterminal Support

- **Hardware Configuration**
- **The System Console**

Multiterminal Programming

- **Terminal Control Blocks**
- **Programming Multiterminal I/O**
- **Debugging a Multiterminal Application**
- **Asynchronous Terminal Status**

Reference

13

13

Using Multiterminal Input/Output

RT–11 provides support for 1 to 16 additional terminals. You can select multiterminal support at system generation time. It is available for the SJ, FB, and XM monitors.

RT–11 provides special programmed requests for multiterminal I/O. The input and output requests are similar to the terminal I/O requests discussed in chapter 12, "Using Terminal Input/Output." There are additional multiterminal requests to attach, detach, obtain, and set status information about specific terminals.

This chapter describes the features provided by the multiterminal support option and the possible hardware configurations. You will learn the significance of the system console in a multiterminal system, as well as how to change the system console, how to run foreground and system jobs with specific terminals as their consoles, and how to write programs for multiterminal applications.

Multiterminal Support

RT–11 provides support for up to 16 additional terminals through a feature which can be selected during system generation. The multiterminal option is available for all three monitors. Multiple-terminal support does not provide a multiuser system, because RT–11 supports only one terminal at a time as the system console. You can transfer the system console from the initial terminal to any of the local terminals. The extra terminals are only I/O devices controlled by applications programs.

Hardware Configuration

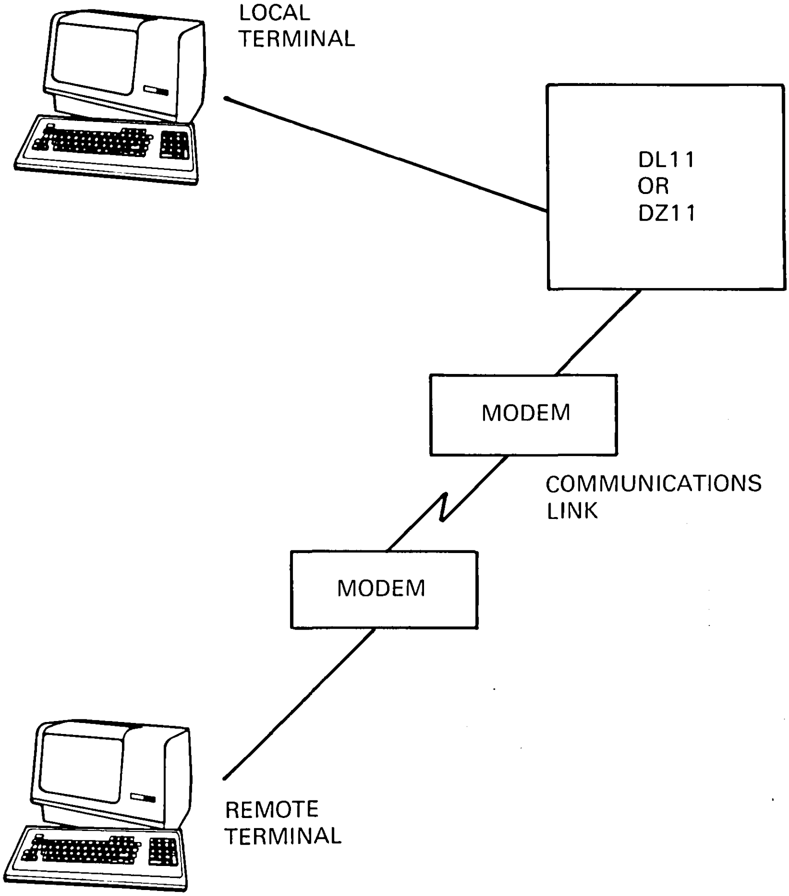
The multiple-terminal feature supports terminals connected through DL11 and DZ11 serial line interfaces (DLV11 and DZV11 for an LSI–11). The terminals connected to these interfaces can be either local or remote. A local terminal is connected directly to the DL or DZ interface. A remote terminal is connected to the DL or DZ interface by a modem and a communication link. Figure 39 shows a local terminal and a remote terminal connected to a DL11 or a DZ11. During system generation, you must specify which terminals are local and which are remote.

The System Console

The system or background console is the terminal used to enter monitor commands and communicate with the background job. KMON prompts you on this terminal, and the terminal I/O programmed requests (discussed in chapter 12, "Using Terminal Input/Output") communicate with this terminal from background jobs. By default, the terminal I/O requests from foreground and system jobs also use this terminal.

When running a foreground or system job, you can use the /TERMINAL option with the FRUN or SRUN command to assign a terminal for the exclusive use of the job (see

Figure 39.
Local and Remote Terminal Connections



chapter 2, "Executing Programs"). Such a terminal cannot be shared by another job. When you assign a terminal to a job, you must use that terminal to enter data to the job and to abort the job with ⟨CTRL/C⟩ ⟨CTRL/C⟩.You cannot communicate with the job by using ⟨CTRL/F⟩ or ⟨CTRL/X⟩ on the system console. You can terminate a foreground or system job that has an assigned terminal, by issuing the ABORT command at the system console.

You can move the system console to any local terminal connected to your system (except a terminal assigned to a foreground or system job) by using the command:

```
SET TERM CONSOL=n
```

Here "n" is the logical unit number of the new console terminal. After you press ⟨RETURN⟩ to terminate this command, RT–11 prints its next prompt on the new system console.

RT–11 does not allow you to set a remote terminal as the system console. A patch to enable you to use a remote terminal as the system console is given in appendix D of the *RT–11 System Generation Guide.*

Multiterminal Programming

An application program can communicate with up to 16 terminals, in addition to the system console, by using the multiterminal programmed requests listed in table 14. If you assign a terminal using FRUN/TERMINAL or SRUN/TERMINAL, you do not need multiterminal requests to communicate with that terminal.

Table 14.
Multiterminal Programmed Requests

MACRO-11 Request	FORTRAN IV Request	Function
.MTATCH	MTATCH	Attach a terminal
.MTIN	MTIN	Input character(s)
.MTOUT	MTOUT	Output character(s)
.MTPRNT	MTPRNT	Output a character string
.MTRCTO	MTRCTO	Reset ⟨CTRL/O⟩
.MTSTAT	MTSTAT	Get multiterminal system status
.MTGET	MTGET	Get terminal status
.MTSET	MTSET	Set terminal status
.MTDTCH	MTDTCH	Detach a terminal

The input and output requests are similar to the console terminal I/O requests .TTYIN, .TTYOUT, and .PRINT. Before you can communicate with a terminal using these requests, you must reserve that terminal, using the .MTATCH or MTATCH request. When you have finished using a terminal, you must release it, using the .MTDTCH or MTDTCH request.

Terminal Control Blocks

When you request multiple-terminal support during system generation, the SYSGEN procedure creates a terminal control block (TCB) for each terminal you specify. The TCBs are assigned to the terminals, starting with the hardware console, and continuing in the following order:

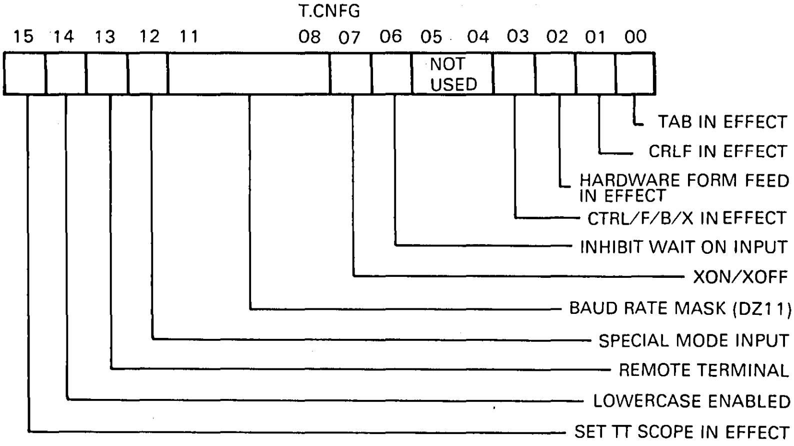
1. Local DLs
2. Remote DLs
3. Local DZs
4. Remote DZs

The number of the TCB assigned to a terminal becomes the logical unit number (LUN) of that terminal, with the hardware console having LUN 0. The TCBs are linked into RMON and form a contiguous table. The .MTSTAT or MTSTAT request returns information that enables you to access the TCBs directly, using the .GVAL request. The programmed requests .MTGET and MTGET retrieve information from the TCB. The requests .MTSET and MTSET allow you to change this information.

The first word in the TCB is the terminal configuration word, shown in figure 40. Bits 6, 12, and 14 of this word have a similar effect to the corresponding bits in the JSW, that is:

- Bit 6 is the inhibit wait bit. If this bit is set, the program does not wait for I/O to complete.

Figure 40.
TCB Configuration Word



- Bit 12 is the special mode bit. If this bit is set, input is in special mode, characters are not echoed, and so on.
- Bit 14 is the lowercase bit. If this bit is clear, all characters are converted to uppercase. If this bit is set, lowercase characters are passed.

In multiterminal applications, you can set these bits for the system console either in the JSW or in the TCB. Setting the bits in either place results in both words having those bits set.

Programming Multiterminal I/O

The sequence of events and requests which you should use when programming multiterminal applications is given below:

1. Examine the system status. To determine if the system under which the job is running has multiter-

minal support, check the SYSGEN features word (RMON fixed offset 372).

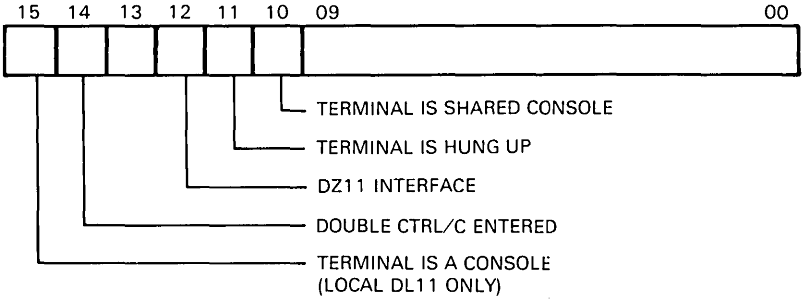
2. Attach a terminal. A job must attach a terminal before it can communicate with that terminal by means of the multiterminal I/O requests. Use the request .MTATCH or MTATCH, and specify the logical unit number of the terminal to be attached.

   Once a job has attached a terminal, no other job can attach or communicate with it until the job issues a .MTDTCH request or terminates.

   As an optional argument, you can specify the address of an asynchronous terminal status word. If you specify this argument, the system automatically notifies your job of certain changes in the terminal's status. This word is described later.

3. Initialize the terminal characteristics. Use the .MTGET or MTGET request to obtain complete status information about the terminal you have attached. The status block returned by this request contains the first six bytes of the TCB, including the terminal configuration word (figure 40). Byte 7 is the terminal state byte, shown in figure 41. Byte 8 is the carriage width, indicating the maximum number of characters on a line.

Figure 41.
Terminal State Byte



To change any of the characteristics, you must modify the data in the status block received by .MTGET or MTGET, and return the new values using the .MTSET or MTSET request.

You can get status information about terminals that are not attached to your job; however, you can set characteristics for attached terminals only.

4. Communicate with the terminal. A job can get characters from an attached terminal using the .MTIN or MTIN request. This is equivalent to the .TTYIN or ITTINR request, except that you can specify the number of characters to be received.

   A job outputs characters using the .MTOUT or MTOUT request, and character strings, using the .MTPRNT request. These are equivalent to the .TTYOUT or ITTOUT and .PRINT or PRINT requests.

   If you want to enable asynchronous I/O, special mode I/O, or lowercase I/O, first set the appropriate bits in the terminal configuration word using .MTSET or MTSET.

   Use the .MTRCTO or MTRCTO request to reset the effect of a ⟨CTRL/O⟩. This is equivalent to the .RCTRLO or RCTRLO request.

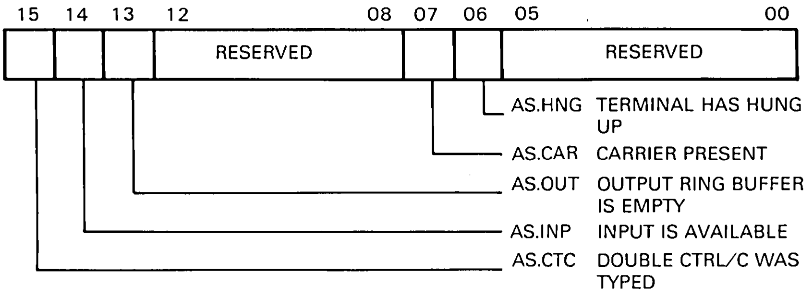
   You should issue a .MTRCTO or MTRCTO request after setting or clearing any bits in the job's JSW or any TCB, and before issuing the first terminal I/O request. This forces the monitor to update all the terminal data structures with the new status.

5. Release the terminal. When the job has finished, it should detach the terminal to make it available for use by other jobs. Use the .MTDTCH or MTDTCH request.

Debugging a Multiterminal Application

Use VDT, the virtual debugging technique, to debug a multiterminal application program.

Figure 42.
Asynchronous Terminal Status Word


15
14
13
12
08
07
06
05
00
RESERVED
RESERVED
AS.HNG TERMINAL HAS HUNG UP
AS.CAR CARRIER PRESENT
AS.OUT OUTPUT RING BUFFER IS EMPTY
AS.INP INPUT IS AVAILABLE
AS.CTC DOUBLE CTRL/C WAS TYPED

Asynchronous Terminal Status

If you select the asynchronous terminal status feature during system generation, the multiterminal interrupt service code automatically notifies a job of certain changes in terminal status after it has been attached. An optional argument to the .MTATCH or MTATCH request specifies the location to be used as the asynchronous terminal status word. Without this feature, a job must issue a .MTGET or MTGET request to detect any changes in status. Figure 42 shows the format of the asynchronous terminal status word.

Practice 13–1

You may write the following multiterminal program in either MACRO–11 or FORTRAN IV, to run under the monitor you prefer. The program should:

1. Check whether or not the system has multiterminal support. If not, it should print an error message on the console and exit.
2. Attach one of the available terminals, other than the system console.
3. Enable lowercase I/O at that terminal.

4. Display (on that terminal) a prompt asking for the user's name, for example:

```
Who are you?
```

5. Read the user's name and then display a response like:

```
Welcome to Multiterminal RT-11, Ann
```

6. Release the terminal and exit.

References

RT–11 Programmer's Reference Manual. Chapter 2 discusses multiterminal input/output requests in MACRO–11 programs. Chapter 3 describes multiterminal input/output requests in FORTRAN IV programs.

RT–11 Software Support Manual. Chapter 5 contains descriptions of the terminal configuration word and the asynchronous terminal status word.

RT–11 System Generation Guide.

14

14

Using Queued Input/Output

Chapter 11, "Using Input/Output Systems," discussed briefly the basic types of I/O available: terminal I/O, queued I/O, and OTS I/O.

The mode of I/O operation presented in this chapter is called synchronous because program execution is suspended until the requested I/O is completed. Asynchronous I/O and event-driven I/O, on the other hand, return control to the program before the transfer is completed. These two I/O modes are discussed in chapter 15, "Using Nonsynchronous Queued Input/Output."

The MACRO–11 programmed requests described in this chapter include: .READW, .WRITW, .FETCH, .ENTER/.LOOKUP, .CLOSE/.PURGE, .RELEAS, .CDFN, .QSET, and .DSTATUS. The FORTRAN IV requests covered include: IREADW, IWRITW, IFETCH/IGETC, IENTER/LOOKUP, CLOSEC/PURGE, IFREEC, ICDFN, IQSET, and IDSTAT.

When you have completed this chapter, you will be able to use synchronous queued I/O requests to read data stored in a file, create a new file and use synchronous queued I/O requests to write data to it, and use synchronous queued I/O requests to read data from or write data to a non-file-structured device.

Concepts of Queued I/O

The RT–11 queued I/O system is used for most I/O to supported I/O devices, including I/O to file-structured and non-file-structured devices. Queued I/O allows device-independent programming and is implemented using the following software components:

- RMON, which receives the programmed requests governing queued I/O operations and passes them on to other components. RMON also keeps track of I/O activity.
- USR, which connects jobs to peripheral devices and handles all access to file directories on volumes.
- Device handlers, which are software routines containing code to handle the details of device specific I/O operations.

Queued I/O is the method used by RT–11 to keep track of pending I/O operations. The choice of the next I/O operation to be performed by the operating system is made by looking at a list, or queue, of requests. Each I/O request results in RMON passing a data structure called a "queue element" to the device handler. The queue element contains all the information needed by the device handler to perform the requested operation.

Using I/O Channels

Although there may be almost any number of devices and files in a given RT–11 system, a specific program uses only a limited number at any given time. Access to devices and files is controlled through a set of I/O "channels." Once a channel is open to a device or a file, the program may call for I/O to be performed to the device or file to which the channel is connected.

Standard Sequence of Requests

The following programs (PR1401.MAC and PR01402.FOR) show how queued I/O requests are performed. The program must perform a basic sequence of activities if successful queued I/O is to occur. These activities are:

1. Make sure that the device handler is in memory.
2. Open a channel to the file or device.
3. Read from and/or write to the channel.
4. Close the channel after all I/O has been completed.
5. Release the device handler from memory.

Performing Queued I/O

Before any queued I/O operations can be performed on a device, the device handler must be resident in memory. The device handler can be loaded either by the console LOAD command or by the program through a programmed request.

Fetching Device Handlers

The MACRO–11 programmed request to load a device handler is .FETCH. The FORTRAN IV system subroutine is IFETCH. These requests bring the specified handler into memory. The form of the MACRO–11 .FETCH programmed request is:

```
.FETCH    addr,devnam
```

In this request, "addr" is the address at which the handler is to be loaded, and "devnam" is the address of a RAD50 word containing the device name.

PR1401.MAC

```
        .TITLE  COPY1   I/O EXAMPLE PROGRAM
;
;       Program copies one file to another and exits.
;
        .MCALL  .EXIT   .FETCH  .LOOKUP .ENTER  .PRINT
        .MCALL  .READW  .WRITW  .CLOSE  .SRESET
EMTARG: .BLKW   6               ;EMT argument block
INFILE: .RAD50  /DK TRAN1 XYZ/  ;Copy from DK:TRAN1.XYZ
OUTFIL: .RAD50  /DK TRAN2 XYZ/  ; to DK:TRAN2.XYZ
LIMITS: .LIMIT                  ;Generate program limits
BUFFER: .BLKW   256.            ;File I/O Buffer
ERROR:  .BYTE                   ;Error status byte
ANNCE:  .ASCIZ  "Program copies TRAN1.XYZ to TRAN2.XYZ"
FCH1MS: .ASCIZ  "Error on FETCH of output handler"
FCH2MS: .ASCIZ  "Error on FETCH of input handler"
LKPMES: .ASCIZ  "Error on LOOKUP of input file"
ENTMES: .ASCIZ  "Error on creation of output file"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine sets up files for I/O.
;       File specifications are fixed in this version.
;       Routine returns with C-Bit SET on error.
;
SETUP:  MOV     R1,-(SP)        ;Save register
        .PRINT  #ANNCE          ;Announce program
;       Fetch device handlers
        MOV     LIMITS+2,R1     ;Load free memory address
        .FETCH  R1,#OUTFIL      ;Get output device handler
        BCS     FCH1ER          ;Branch on FETCH error
        MOV     R0,R1           ;Copy free address
        .FETCH  R1,#INFILE      ;Get input device handler
        BCS     FCH2ER          ;Branch on FETCH error
;       Open input and output files
        .LOOKUP #EMTARG,#3,#INFILE ;Open input file
        BCS     LKPERR          ;Branch if failed
        MOV     R0,R1           ;Save input file length
        .ENTER  #EMTARG,#0,#OUTFIL ;Create output file
        BCC     DONE            ;Return if no error
;       Error Routines
        .PRINT  #ENTMES         ;Issue error creating output
        BR      ERDONE          ; file message and return
LKPERR: .PRINT  #LKPMES         ;Issue failed to open input
        BR      ERDONE          ; file message and return
FCH2ER: .PRINT  #FCH2MS         ;Issue FETCH error message
        BR      ERDONE          ; and return
FCH1ER: .PRINT  #FCH1MS         ;Issue FETCH error message
ERDONE: SEC                     ;Indicate error occurred
```

PR1401.MAC (continued)

```
DONE:   MOV     (SP)+,R1        ;Restore R1 (save C-bit)
        RETURN                  ;Return to caller
        .SBTTL  CPYRTN  -- Synchronous Copy (Single Buffer)
;
;       Routine assumes that the input file is opened
;       on channel 3 and the output on channel 0.
;       Returns with C-BIT SET on error.
;
;       Note: All registers except R0 are preserved.
;
CPYRTN: MOV     R1,-(SP)        ;Save register
        CLR     R1              ;Clear block number
        CLRB    ERROR           ;Init error flag
1$:     .READW  #EMTARG,#3,#BUFFER,#256.,R1
        BCC     2$              ;Branch if read succeeded
        TSTB    @#52            ;End-Of-File reached?
        BEQ     EXIT            ;Branch if so
        BR      RDERR           ;Otherwise, process error
2$:     .WRITW  #EMTARG,#0,#BUFFER,#256.,R1
        BCS     WERR            ;Branch on write failure
        INC     R1              ;Update block number
        BR      1$              ;And read next block
RDERR:  .PRINT  #RERRMS         ;Issue read error message
        BR      EREXIT          ;And finish up
WERR:   .PRINT  #WERRMS         ;Issue write error message
EREXIT: DECB    ERROR           ;Set error flag
EXIT:   MOV     (SP)+,R1        ;Restore saved register
        TSTB    ERROR           ;Error? (and clear C-BIT)
        BEQ     1$              ;Branch if not
        SEC                     ;Otherwise, set C-BIT
1$:     RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Cleanup For Copy Program
CLSCHN: .CLOSE  #3              ;Close input file
        .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch if succeeded
        .PRINT  #PRTCT          ;Output file is protected
PRGCHN:                         ;Purge files (.SRESET)
RESET:  .SRESET                 ;Reset system
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Get file names
        BCS     1$              ;Branch if failed
        CALL    CPYRTN          ;Copy the input to output
        BCS     1$              ;Branch if failed
        CALL    CLSCHN          ;Close the channels
        BR      2$
1$:     CALL    PRGCHN          ;Purge the channels
2$:     .EXIT                   ;Exit
        .END    START
```

PR1402.FOR

```
      PROGRAM COPY1
C
C     Program performs a file to file copy and then
C     exits.
C
      LOGICAL*1 SETUP,CPYRTN  ! Declare functions
      LOGICAL*1 ERROR
C
      ERROR = SETUP()         ! Open files
      IF (ERROR) GO TO 20     ! Stop on error
      ERROR = CPYRTN()        ! Copy file
      IF (ERROR) GO TO 20     ! Stop on error
      CALL CLSCHN             ! Close files and exit
      GO TO 30
20    CALL PRGCHN             ! Purge channels
30    CALL EXIT
      END
      FUNCTION SETUP
C
C     This routine sets up the files for I/O.
C     File specifications are fixed in this version.
C
C     Function returns .TRUE. if an error occurred.
C
      LOGICAL*1 SETUP
      INTEGER*2 INCHN,OUTCHN
      COMMON /CHNNLS/ INCHN,OUTCHN
C
C     Channel numbers are common because they are
C     used by CPYRTN, CLSCHN, and PRGCHN.
C
C     Input (DK:TRAN1.XYZ) and output (DK:TRAN2.XYZ)
C     file specifications:
C
      INTEGER*2 INFILE(4),OUTFIL(4)
      DATA INFILE/2RDK,3RTRA,2RN1,3RXYZ/
      DATA OUTFIL/2RDK,3RTRA,2RN2,3RXYZ/
C
C     Output introductory message and allocate channels.
C
      CALL PRINT('Program copies TRAN1.XYZ to TRAN2.XYZ')
      INCHN = IGETC()
      OUTCHN = IGETC()
C
C     Fetch needed device handlers.
C
      IF (IFETCH(OUTFIL(1)) .NE. 0) GO TO 101
      IF (IFETCH(INFILE(1)) .NE. 0) GO TO 102
C
C     Open input file.
C
```

PR1402.FOR (continued)

```
        LENGTH = LOOKUP(INCHN,INFILE)
        IF (LENGTH .LT. 0) GO TO 103
C
C       Create output file.
C
        IF (IENTER(OUTCHN,OUTFIL,LENGTH) .LT. 0) GO TO 104
        SETUP = .FALSE.         ! No error
        RETURN
C
C       ERROR ROUTINES
C
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.          ! Error
        RETURN
        END
        FUNCTION CPYRTN
C
C       Single buffered, synchronous copy routine.
C
C       Function returns .TRUE. on error.
C
        LOGICAL*1 CPYRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFFER(256),BLOCK
        BLOCK = 0               ! Init block number
C
C       Read/write loop.
C
20      IERR = IREADW(256,BUFFER,BLOCK,INCHN)
        IF (IERR .GE. 0) GO TO 30 ! Read successful
        IF (IERR .EQ. (-1)) GO TO 150 ! End of File?
        GO TO 100               ! Error
C
C       Write out buffer just read.
C
30      IF (IWRITW(256,BUFFER,BLOCK,OUTCHN) .LT. 0)
     1    GO TO 101
        BLOCK = BLOCK+1         ! Update to block
        GO TO 20                ! Read next block
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
```

PR1402.FOR (continued)

```
101     CALL PRINT('Write error, copy aborted')
140     CPYRTN = .TRUE.
        RETURN
C
C       Successful return.
C
150     CPYRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1    ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        RETURN
        END
```

The best place to put a device handler is immediately above the memory your program is using—the job's high limit. You can keep track of the program's high limit by using the macro directive .LIMIT.

EXAMPLE

```
        .FETCH  LIMIT+2,#LPNAM
        BCS     ERROR
        MOV     R0,LIMIT+2
          .
          .
          .
LPNAM:  .RAD50  /LP/
LIMIT:  .LIMIT
```

When fetching multiple device handlers, fetch the next one into the area directly above the preceding one. This is easy because after each .FETCH, R0 points to the word above the device handler that was last fetched. By copying R0 into another location, you can refer to that value in implementing the next fetch. Do not do your next fetch using R0 as the addr argument to the .FETCH macro because the contents of R0 are changed in the macro expansion before it is referenced.

In FORTRAN IV programs you would use the IFETCH routine, which has the form:

IERR=IFETCH(devnam)

In this routine, "devnam" is a variable that contains the RAD50 code for the device handler to be fetched. The device handler is positioned within the FORTRAN IV OTS workspace.

EXAMPLE

```
INTEGER HVAR
DATA HVAR/3RDK /
IERR=IFETCH(HVAR)
```

Selecting a Channel

After you have fetched the device handler into memory, you can open I/O channels to the device. Channels are referred to by number. By default, channel numbers 0 to 15 (decimal), or 0 to 17 (octal), are available. If your program is not overlaid, you can use any of these numbers. If it is overlaid, do not use channel 15 (octal 17), as this is the channel used by the overlay process. If necessary, you can write code to check bit 9 of the JSW to see if the program is overlaid.

If a FORTRAN IV program does not use the FORTRAN IV OTS I/O routines, the rules above apply without change. If the program does use these facilities, it is up to you to make sure that the channel numbers you select are not being used for OTS I/O. OTS routines use RT–11 programmed requests to perform FORTRAN IV I/O, so some channels may be occupied when you try to gain access to them. To get a channel for your own purpose, use the IGETC request.

EXAMPLE

```
ICHAN=IGETC()
```

This call asks the OTS to supply you with an available channel and mark the channel "in use" so that the OTS does not try to use the channel itself. IGETC returns the number of the channel. When your program stops using a channel, you must return the channel to the OTS by calling the IFREEC routine.

EXAMPLE

```
IERR=IFREEC(ICHAN)
```

Remember, you have not disconnected a channel until you have closed it by using the CLOSEC routine (discussed later).

Opening a Channel

Having selected a channel, you can connect it to a device or to a file on a file-structured device. To connect the channel, issue either a LOOKUP or an ENTER request. Both take a channel number and the address of a four-word block containing the device name, file name, and file type of the file in RAD50 format. LOOKUP connects the channel to an old file. ENTER connects the channel to a new file.

If the device to which you are connecting the channel is not file structured, the file name and file type are ignored by the request and can be left zero. For a file-structured device, the LOOKUP request searches the directory for the specified file; the ENTER request creates a new file with the name given. If the device is file structured and you perform a LOOKUP without specifying a file name, RT–11 opens the device as one large file. This is called a non-file-structured LOOKUP. An ENTER request to a file-structured device requires a specific file name.

When you create a new file using ENTER, it is referred to as tentative. The characteristics of the tentative status are as follows:

- The status of the directory entry for the file is flagged as tentative. (When the file is closed correctly, its status is flagged as permanent.)
- A tentative length is recorded in the directory. It may be that not all the allocated space is used by the operations performed during the program run, but the allocated space is reserved for possible use by the file until the channel is closed.

When the file is closed correctly, the following events take place:

- The tentative status changes to permanent.
- The length of the file is updated to record only the actual space used. The USR uses the fourth word of the channel table to store this information.

- Any other file on the same volume with the same name and file type is deleted.

In general, when you use ENTER to create a file which has the same specification as an old file, the old file is deleted when the new one is closed. This does not occur, however, if the old file is protected. You can prevent an unprotected file from being deleted by accident. Before issuing an ENTER request, perform a LOOKUP operation to see if there is a file with that name. If the LOOKUP operation fails because the file is not found, it is safe to perform an ENTER. A protected file is never deleted because of an ENTER. If you issue an ENTER request using a name assigned to a protected file, the ENTER request returns an error.

It may happen that when you try to close a tentative file, the USR finds a protected file that was not there when the ENTER request was issued. If this situation occurs, the CLOSE request returns an error but the file is closed correctly. Then there are two files with the same name on the device.

The ENTER request has one argument that the LOOKUP request does not have—the length you want to allocate to the file. The length is one of the following values:

- A positive number giving the length of the file in blocks. The USR finds the first empty area on the device that is large enough and allocates the specified number of blocks to the tentative file.
- The value −1. The USR allocates the largest empty area available on the volume.
- The value 0. The USR allocates the larger of either: half of the largest empty space or all of the second largest space.

The form of the MACRO–11 request .ENTER is:

```
.ENTER    area,chan,file,length
```

EXAMPLE

```
          .ENTER   #AREA,#0,#FNAM,#0
          ...
AREA:     .BLKW 4
FNAM:     .RAD50 /DK FILE  TYP/
```

The form of the FORTRAN IV request IENTER is:

length = IENTER (chan,file,length)

EXAMPLE

```
LEN=IENTER(ICHAN,FNAME,LENGTH)
```

The LOOKUP request is used to access a permanent file on the device. The MACRO–11 request for lookup has the form:

.LOOKUP   area,chan,filename

FORTRAN IV programmers use the LOOKUP system subroutine which has the form:

length = LOOKUP(chan,filename)

The LOOKUP and ENTER requests, in either MACRO–11 or FORTRAN IV, both return the actual number of blocks allocated to the file. The value is returned in R0 for MACRO–11.

Synchronous I/O Requests

While synchronous I/O is being performed, control does not return to the job that issued the request until the I/O operation is complete. Other jobs can be executed while the job is waiting for a return of control.

To perform synchronous I/O you use the READW and WRITW requests. The arguments to these requests are:

channel number — Any channel referred to in the requests must have previously been opened using ENTER or LOOKUP.

buffer — The memory buffer is the source of data for a write operation, or the destination of data for a read operation.

word count — The number of words to be transferred (use null bytes if necessary to fill words).

block number — In file I/O, the relative block number within the file at which the data transfer is to start. The first block in the file is always block zero.

In non-file-structured I/O to a file-structured device, the block number refers to the physical block on the device, starting with block 0.

The READW and WRITW requests cause data to be read from and written to the device. The W means "wait for completion." For MACRO–11 programmers, the form of these requests is:

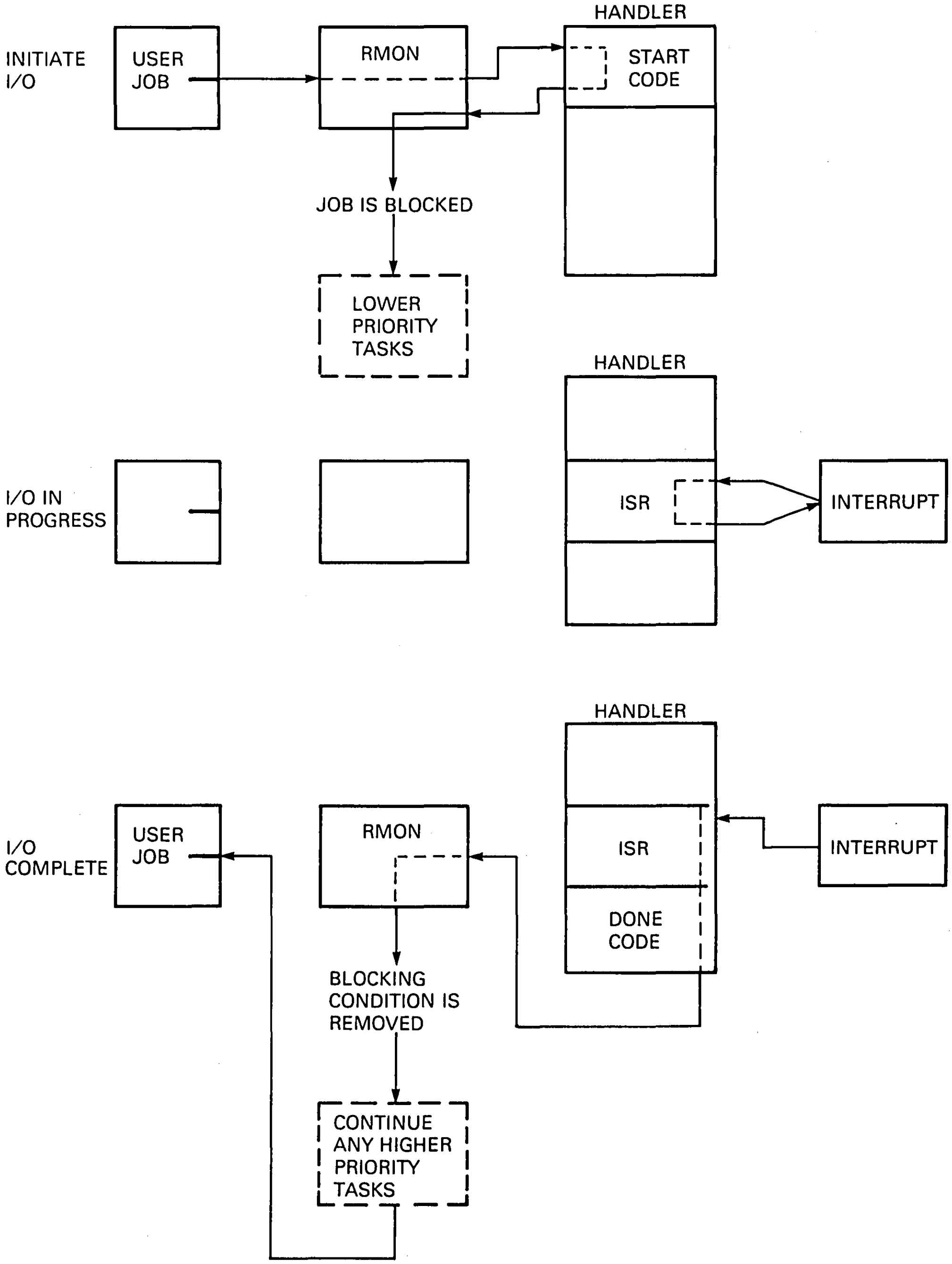
```
.READW/.WRITW    area,channel,buffer,wordcnt,block
```

For FORTRAN IV programmers, the system subroutines are called with the form:

```
IERR=IREADW/IWRITW(wcnt,buff,block,chan)
```

When a job issues a synchronous I/O request, a specific sequence of events is initiated. Refer to figure 43 while reading the following description of this sequence.

Figure 43.
Flow of Control in Synchronous Input/Output



1. Using the arguments provided by the request, RMON builds an I/O queue element and passes it to the appropriate device handler.
2. The device handler starts the I/O operation and returns control to RMON.
3. RMON blocks the job that issued the request, and starts a scheduling search that results in control being passed to any lower priority jobs waiting to run.
4. The I/O is controlled either by direct memory access (DMA) or interrupt processing, while other jobs, if any, execute. DMA is a mode of memory access by which a device can access memory locations directly with no help from the CPU.
5. When the device handler recognizes that the I/O operation is complete, it informs RMON and passes back the queue element.
6. The job that issued the I/O request is set runnable by RMON. Another scheduling pass is done and the job continues execution when control is returned to it.

Transfers to Block Replaceable Devices

When using block replaceable devices such as disks, remember that all I/O transfers start at the beginning of a block. So if you read ten words from block 0 and then issue a request for another ten words from block 0, you read the same ten words again. Also, remember that if a write operation leaves a part of a block empty, the remainder of the last block is filled with zeros.

Because of these block replaceable device features, you should always transfer data in units of a block (256 words). If you must update data in the center of a block, you should read the block, update the data in memory, and write the modified block.

When you issue a read or write request on a channel, RMON checks the request to make sure that the block number is within the file. If the block number is outside the file, RMON returns an end-of-file error and no data is transferred.

If the first block of the transfer is inside the file, RMON checks to make sure that the last block of the transfer is also in the file. If the file is too short for the transfer, RMON adjusts the word count to make the transfer fit the file. So, on block replaceable devices, an end-of-file error is returned only if the block number given in the request is past the end of the file.

Transfers to Sequential Devices

The end-of-file processing for sequential devices is compatible with that for block replaceable devices. If the data to be read is not at the end of the file, as much data as possible is transferred and no I/O error is generated. An end-of-file error is generated only if the device is at the end of the file when you make the request.

Closing a Channel

You must always close an open channel when you want to use that channel for another file or when the channel was opened using ENTER and you want to keep the data written to it.

You may exit without closing a channel if the channel was opened with LOOKUP, or if the channel was opened by ENTER but was used as a scratch file for data to be discarded. If you exit without closing the file, the file will be erased.

The CLOSE request closes a file correctly. If it was opened with an ENTER request, the file has its name entered in the directory. The channel is disconnected from the file and cannot be used until it is opened again.

The PURGE request for MACRO–11 and FORTRAN IV programmers is provided for use when you have used ENTER to open a file that is not to become permanent. The

PURGE request disconnects the channel from the file, and the file is lost. The MACRO–11 requests have the form:

```
.CLOSE/.PURGE    chan
```

The FORTRAN IV calls have the form:

```
CALL CLOSEC/PURGE(chan)
```

Releasing a Device Handler

Releasing a device handler makes its memory space available for program use. This function is available only to MACRO–11 programs; there is no similar call in FORTRAN IV.

To reuse space efficiently, release device handlers in reverse order to their fetching. If you are releasing only some of the device handlers, find the address of the lowest one by using the .DSTATUS request (discussed later) before releasing it. Remember to subtract the size from the pointer returned by the .DSTATUS request. To release all of the device handlers in one operation, use the .SRESET command. To release a specific device handler from memory, the MACRO–11 programmer uses the .RELEAS request in the form:

```
.RELEAS    dnam
```

In this request, "dnam" is the address of the RAD50 device name.

Data Structures

The programmed requests that allow you to use the RT–11 queued I/O system require a number of different data items. You must provide some of these in your program; others are generated by the operating system.

User-created File Specifications

You must specify device names to be used in the fetching operation, and file specifications to open channels. Device and file names are specified in one- and three-word data blocks, encoded in the RAD50 format discussed in chapter 10, "Controlling Program Execution." You may combine these items into one four-word block.

System-maintained Channels

The system maintains a five-word data block for each channel, with 16 channels available to each job by default. Figure 44 is a schematic diagram of this five-word data block. The data block for a specified channel is filled in when the channel is opened. The data block is accessed by the system when:

- A read or write request is issued. The data is checked for trying to read or write past the end of the file and

Figure 44.
Input/Output Channel Data Block

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

then used to build the I/O queue element for the request (discussed later).

- A new file is closed. The data block is used to update the directory.

C.USED is undefined if the channel is opened by a LOOKUP request. If opened by an ENTER, the value of C.USED is the number of the highest block written.

If your program needs more than 16 channels, you can use the programmed request .CDFN to request the total number of channels you want. Each .CDFN request supersedes any previous .CDFN request. The MACRO–11 form of the .CDFN request is:

```
.CDFN   area,addr,num
```

In this request:

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

addr — is the address of an area you have reserved in your program for use as channel tables. The size of the area must be 5*num words.

num — is the total number of channels you want (maximum of 255).

EXAMPLE

```
.CDFN   #EMTBLK,#AREA,#26
```

You use the FORTRAN IV system subroutine ICDFN in the form:

```
IERR=ICDFN(num)
```

Here "num" is the total number of channels you want. The memory for the channel tables is taken from the FORTRAN IV OTS workspace.

System-maintained I/O Queue Elements

The system maintains a seven-word queue element under the SJ and FB monitors, and a ten-word element under the XM monitor. One queue element is created for each job. The contents of an I/O queue element are shown in figure 45. The queue element is used in the following ways:

- It is filled in by RMON in response to a read or write request.
- RMON passes the queue element to the device handler, which uses it while servicing a read or write request.
- When the device handler has completed the execution of the I/O request, it passes the queue element back to RMON.

When not in use, a job's queue elements are kept in a list of available elements. When a job issues a request that needs a queue element, one is removed from the list. I/O requests, inter-job communication requests, and timer requests all require queue elements. When an operation using a queue element completes, the queue element is returned to the list.

As previously discussed, one queue element is available to each job. If your program needs more than one queue element, you can use the .QSET request. This condition will be discussed in chapter 15, "Using Nonsynchronous Queued Input/Output."

In the .QSET request, you specify the number of queue elements you want added to the list, rather than the total number of elements. The MACRO–11 request form is:

```
.QSET addr, num
```

In this request, "addr" is a block of memory you have reserved to be used for queue elements. The size of this area must be 7*num under the SJ or FB monitor and 10*num under the XM monitor. Here "num" is the number of additional queue elements you want to reserve.

Figure 45.
Structure of an Input/Output Queue Element

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

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

EXAMPLE

```
NUM=3
          . . .
          .QSET     #QADD,#NUM
          . . .
QADD:     .BLKW     10.*NUM
```

The name of the FORTRAN IV request is IQSET which has the form:

IERR=IQSET(num)

Here "num" is the number of additional queue elements to be allocated. The memory for these additional elements is taken from the FORTRAN IV OTS workspace.

Device Handlers

A device handler is a software routine that performs the operations necessary to allow a device to respond to I/O requests as planned. Device handlers may be either permanently resident in memory, or installed and removed as needed by your program. Before your program can request I/O to any device, the device handler for that device must have been placed in memory. You can do this in three different ways:

1. Permanently resident. The handler for the system device (SY:) is always resident; the TT: handler is resident in FB and XM systems.
2. Loaded from KMON, using the LOAD command.
3. Fetched by the program during execution, using a programmed request.

Device handlers to be used by foreground programs must be loaded. Background programs can either load or fetch handlers from within the program. Loaded handlers stay in memory until unloaded from KMON or removed as a result of a system reboot.

Loading a device handler means that your program does not have to include instructions to fetch the handler; on the other hand, loading from KMON requires the help of an operator. Even if you expect that the device handler will be loaded before your program is run, you can assure success by performing either of the following:

1. Do a fetch (background jobs only); a fetch for resident handler returns immediately, indicating success.
2. Use the DSTATUS (device status) request in your program to check if the handler is resident. Provide code to print an error message and exit if it is not.

.DSTATUS returns four words of information about the specified device. You supply the logical or permanent device name. The MACRO–11 form of .DSTATUS is:

```
.DSTATUS    infblk,device
```

In this request, "infblk" is the address of a four-word block to which the information is returned and "device" is the address of a RAD50 word containing the device name.

The FORTRAN IV system subroutine name for the device status request is IDSTAT. Its form is:

```
IERR=IDSTAT(device,infblk)
```

Here "device" contains a RAD50 word for the device, while "infblk" is a four-word area to store the returned information.

You use the DSTATUS or IDSTAT request to:

- Check for the correct physical device, if it is important that your program access a specific physical device.
- Check device characteristics. For example, you may want to make sure that your program will not try to write to a read-only device.
- Check the size of the handler before a fetch.
- Check if a handler is resident.

Figure 46 shows the information returned by DSTATUS or IDSTAT. The following list elaborates the DSTATUS function:

1. DEVICE CHARACTERISTICS. Refer to the previous list on uses of DSTATUS or IDSTAT.
2. DEVICE CODE. A unique number for every physical device. A list of these assignments is available in chapter 2 of the *RT–11 Programmer's Reference Manual.*
3. HANDLER SIZE. This number represents the number of bytes.

4. POINTER TO HANDLER. If the handler is resident, this number is the load point plus 6. If the handler is not resident, the value is 0.
5. DEVICE SIZE. In 256-word blocks. If the device is sequential, the value is 0.

Figure 46.
DSTATUS Information Block

<table>
<tr><td>DEVICE CHARACTERISTICS</td><td>DEVICE CODE</td></tr>
<tr><td colspan="2">HANDLER SIZE</td></tr>
<tr><td colspan="2">POINTER TO HANDLER</td></tr>
<tr><td colspan="2">DEVICE SIZE</td></tr>
</table>

Practice 14–1

1. Type the program PR1401.MAC or PR1402.FOR into a file.
2. Run the program. When it runs successfully, copy it to a file called PR1801.MAC or PR1802.FOR for use in practice 18-1.
3. Change the program so that it copies from a different file.
4. Change the program so that it reads and writes 512 words per access instead of 256.

Reference

RT–11 Programmer's Reference Manual. Chapter 2 discusses the READW and WRITW requests in detail.

15

15

Using Nonsynchronous Queued Input/Output

The architecture of the PDP–11 allows a large number of I/O operations to occur in parallel with computation. Thus, the PDP–11 enables real-time applications to operate with high-speed I/O devices. A slow application program can miss high-speed data input or lose control of a high-speed output device.

In chapter 14, "Using Queued Input/Output," you learned how to use a class of I/O requests called synchronous I/O requests. When you use these requests your program does not proceed with its execution until the requested I/O completes and, therefore, you do not take full advantage of the PDP–11's capability. In this chapter we will discuss another class of I/O requests—nonsynchronous I/O. Nonsynchronous I/O allows you to make use of the I/O architecture by requesting I/O transfers that run concurrently with other transfers and with computation. Careful use of these requests can greatly increase the execution speed of your program.

The MACRO–11 requests discussed in this chapter are: .READ, .WRITE, .WAIT, .READC, and .WRITC. The FORTRAN IV requests discussed are: IREAD, IWRITE, IWAIT, IREADC, IWRITC, IREADF, and IWRITF.

When you have completed this chapter, you will be able to select the best mode(s) of I/O to use, given input/output specifications for a program. The specifications may include the average data rate of each I/O channel, the amount by which the data rates change, the tradeoff considerations of throughput against ease of programming, and the expected ratio of computing time to I/O duration.

You will also be able to determine whether throughput can be improved by overlapping input, computation, or output, given specifications for a program. You will then learn to design and implement an algorithm that includes such overlapping. Given specifications of a program with multiple, independent input channels, you will also learn to design and implement the program, using event-driven I/O.

Nonsynchronous I/O

Any I/O transfer includes three important events:

1. Starting the transfer. Your program issues a READ or WRITE programmed request. Control leaves the job and enters the monitor. The "request count" byte in the channel table is incremented.
2. Returning control to your program.
3. Assuring completion of the I/O transfer. This means that requested input information is available for processing, or that information in an output buffer has been transferred and the buffer can be used again.

When using nonsynchronous I/O, you must take special steps within your program to check for or wait for completion of the I/O transfer. This makes programming with nonsynchronous I/O more difficult than with synchronous I/O.

I/O Modes

RT–11 supports I/O requests in three modes. Each mode has its own set of programmed requests, is executed in a different way, and is more useful in certain types of applications. Matching your programming needs to the best I/O mode makes the most efficient use of your RT–11 system and your programming time.

You have learned about synchronous I/O. The two nonsynchronous modes are asynchronous and event-driven. Event-driven I/O is also called I/O with completion routines. Both nonsynchronous modes return control to your program immediately and therefore, allow you to perform other operations while I/O is in progress. They differ, however, in the actions that RT–11 takes when the I/O transfer completes, and they also differ in the way that you write your program in order to use them.

Use asynchronous I/O in operations where you may want to perform other activities after starting the I/O. This can be other processing or another transfer request. When your program reaches the point where it must wait for I/O completion, your job issues a WAIT request. When a WAIT request is issued, control leaves the job and RMON does not continue execution of that job until all I/O has completed on the channel you specified in the WAIT request. RMON implements the WAIT using the request count of the channel table. When that request count returns to zero, RMON once again allows the job to run.

Asynchronous I/O is somewhat similar to synchronous I/O. In synchronous I/O the job issues a request and immediately asks to be blocked until that request completes. In asynchronous I/O, issuing the I/O request is separated from the blocking operation, allowing the job to perform other activities before being blocked. Asynchronous I/O is most often used:

- To allow I/O on two or more devices to proceed concurrently. When control returns from one I/O request, the job issues another.

- To allow I/O to proceed concurrently with computation. The job issues I/O requests and then proceeds with computations on data held in memory.

Event-driven I/O allows you to specify the operations that you want performed when I/O completes, while allowing your program to do other useful work. When your program issues an event-driven I/O request, you specify a routine that you have written and ask for the routine to be identified as the completion routine for this request. When I/O completes, the job is interrupted, the completion routine is run, and control returns to the job at the point of interruption.

Asynchronous I/O is the better choice when your program has only a limited number of operations to perform before it must wait and when it is only waiting for the completion of I/O on a specific channel. Event-driven I/O is more appropriate if:

- Your job can or must continue processing while waiting for I/O to complete. For example, if your job is controlling a real-time process using parameters that change because of input information, the job must continue to use the old parameters until new ones come in. It cannot stop and wait for the new input.
- Your job must wait for a specific I/O request (if multiple requests have been issued on the same channel).
- Your job must wait for some combination of requests to complete. For example, you may want the job to wait until one of a group of selected I/O requests completes.

Using I/O Queue Elements

The use of I/O queue elements is transparent for synchronous I/O. For nonsynchronous I/O, however, if you do not make sure you have enough I/O queue elements, your job can become blocked, making your program slower.

A request for an I/O transfer is an outstanding request from the time it is issued until the time it completes. While a request is outstanding, the device handler servicing it has exclusive use of the queue element for that request. For event-driven I/O, this queue element is also kept in use while the completion routine is running.

If you use only synchronous I/O, the one queue element (automatically provided) is enough, because at any given time only one I/O request is outstanding. If you use nonsynchronous I/O, the number of outstanding I/O requests is unlimited. To get the best performance from your program, estimate the maximum number of I/O requests that will be outstanding at any given time, then use .QSET or IQSET to make sure you have enough I/O queue elements. If you issue an I/O request and no queue element is available, your job is blocked until one becomes available.

Asynchronous Requests

The MACRO–11 format for the asynchronous I/O requests is:

```
.READ   area,chan,buff,wcnt,block
.WRITE  area,chan,buff,wcnt,block
```

The FORTRAN IV format is:

```
IERR=IREAD (wcnt,buff,blk,chan)
IERR=IWRITE (wcnt,buff,blk,chan)
```

The arguments are identical to those of the synchronous requests.

You may also use the .WAIT or .IWAIT request in the following formats:

```
MACRO–11:     .WAIT  chan
FORTRAN IV:   IERR=IWAIT (chan)
```

When you use .WAIT or IWAIT, you have to wait for all I/O on a selected channel to complete. If you want to wait for a specific request to complete, do one of the following:

- Use a synchronous request.
- Use an asynchronous request and do not issue any other request on that channel before issuing the .WAIT or IWAIT.
- Use event-driven I/O and have a completion routine run when that specific I/O request completes.

Using Asynchronous I/O to Implement Multiple Buffering

Sometimes a program can be handled as three operations:

- Input: the transfer of data from a peripheral device to memory
- Computation: the production of new data in memory using data in memory
- Output: the transfer of data from memory to a peripheral device

The advantage of using a synchronous I/O is that these operations can take place concurrently, although they are logically sequential. Data must be input before computations can be performed, and computations must be complete before results can be written out. Thus, there are two opposing needs: to have the operations take place concurrently to increase speed, and sequentially, for the logic of the program.

This problem can be solved by multiple buffering. Input data is read into a buffer. When that buffer is full, data is read into a second buffer while computation starts on the first. Each buffer goes through the necessary operations in sequence, but simultaneously, one buffer can be in use for input, one for output, and one or more for computation.

Benefits of Multiple Buffering

To see how multiple buffering helps, let's look at a simple copy operation which omits the computation step from the read/compute/write cycle. Figure 47 shows a synchronous, single-buffered copy operation. The box represents the buffer. An arrow pointing into the buffer represents a read

Figure 47.
Synchronous Single-buffered Copy Sequence

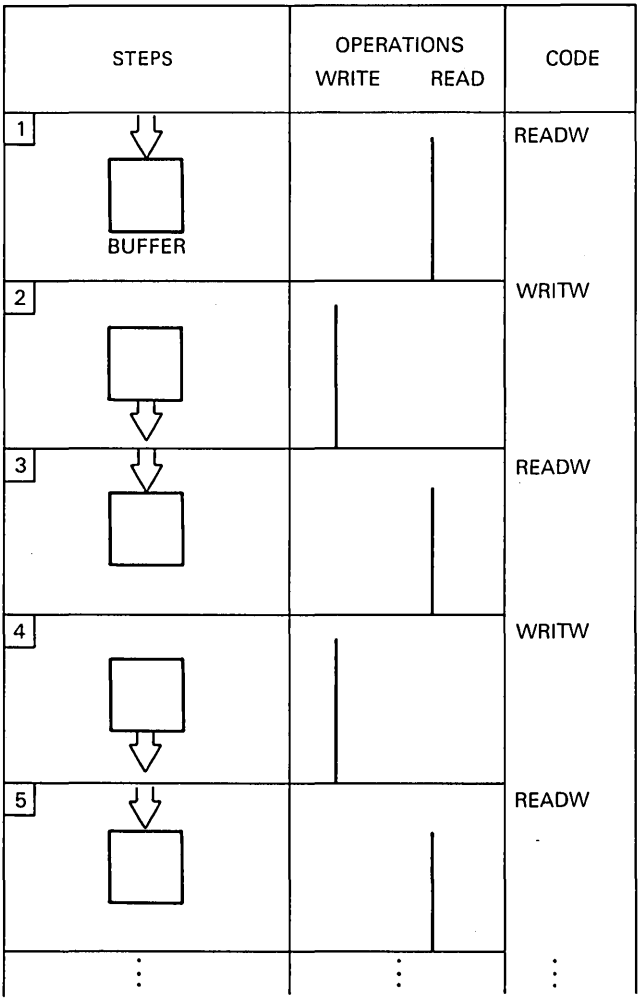
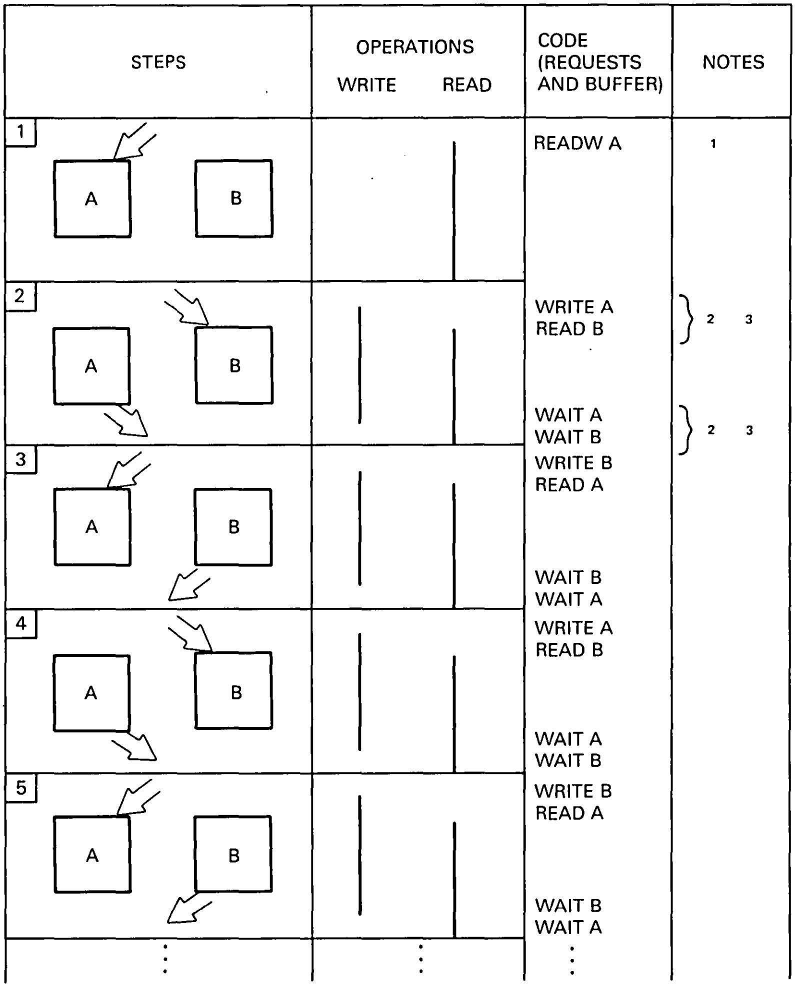


Figure 48.
Double-buffered Copy Sequence



STEPS	OPERATIONS WRITE READ	CODE (REQUESTS AND BUFFER)	NOTES
1 A B		READW A	1
2 A B		WRITE A READ B WAIT A WAIT B	2 3 2 3
3 A B		WRITE B READ A WAIT B WAIT A	
4 A B		WRITE A READ B WAIT A WAIT B	
5 A B		WRITE B READ A WAIT B WAIT A	
⋮	⋮	⋮	

operation, and another arrow pointing out of the buffer represents a write. The figure indicates how time is divided among the read and write operations over three cycles of the program.

Figure 48 shows a double-buffered copy. By using two buffers, the READ and the WRITE can execute concurrently. This figure shows the overlap of these operations. As you study the figure, please read the following notes:

1 Instead of using a READW, you could use a READ followed by a WAIT. This may make the program easier to write.

2 The order in which you issue the READ, WRITE, and WAIT requests generally makes no significant difference if the input and output devices are of approximately the same speed. For example, if you know that the output device is slower than the input device, issue the WRITE on that channel first and the WAIT on that channel last.

3 If you switch either the order of the READ and WRITE, or the order of the WAITs from that shown in figure 48, you can change the second I/O request to synchronous mode and remove the first WAIT. There will be a small increase in speed because you are performing only one programmed request instead of two. The gains are probably not significant, however.

Figure 49 compares single buffering with double buffering. The longer the programs run, the more time is saved. In the long run, execution time can be decreased by up to 50 percent using this method.

Figure 50 shows a flowchart of a double-buffered copy program. End-of-file and error conditions have been omitted to keep the flowchart simple. Following this figure, are MACRO–11 and FORTRAN IV programs that carry out this double-buffered copying method.

Two conditions have been assumed in figures 47 through 50 and the two programs (PR1501.MAC and PR1502.FOR):

1. The input and output devices are not the same. If they were, a program using two buffers would run no faster than a single-buffered program, because the device handler is able to service only one request at a time.

   If a device handler is to control a number of devices that can, in fact, operate independently, you will need to write a device handler that queues I/O requests internally and thereby avoids the normal serialization of the queued I/O system. Only if you are working with this sort of device handler, can you overlap operations that are being performed by it.

2. The time graphs show approximately equal input and output times. This is the condition in which a double-buffered copy program is more efficient. The greater the difference between the speed of the input device and that of the output device, the smaller the benefit of a double-buffered copy over a single-buffered copy.

Figure 49.
Comparison of Single- and Double-buffered Copy

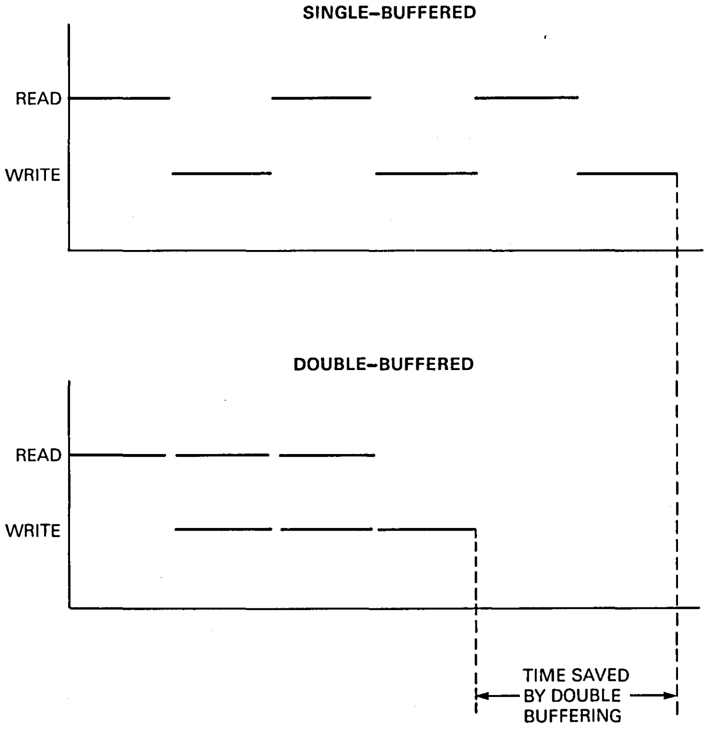
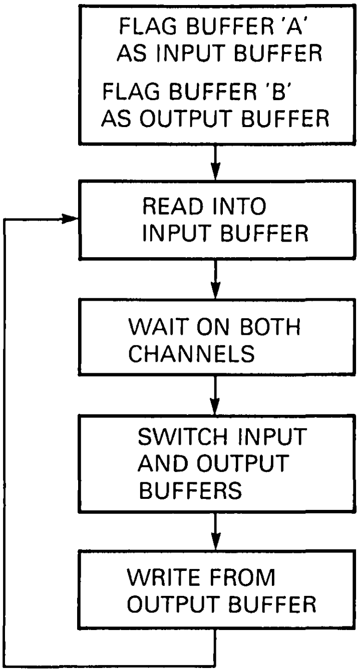


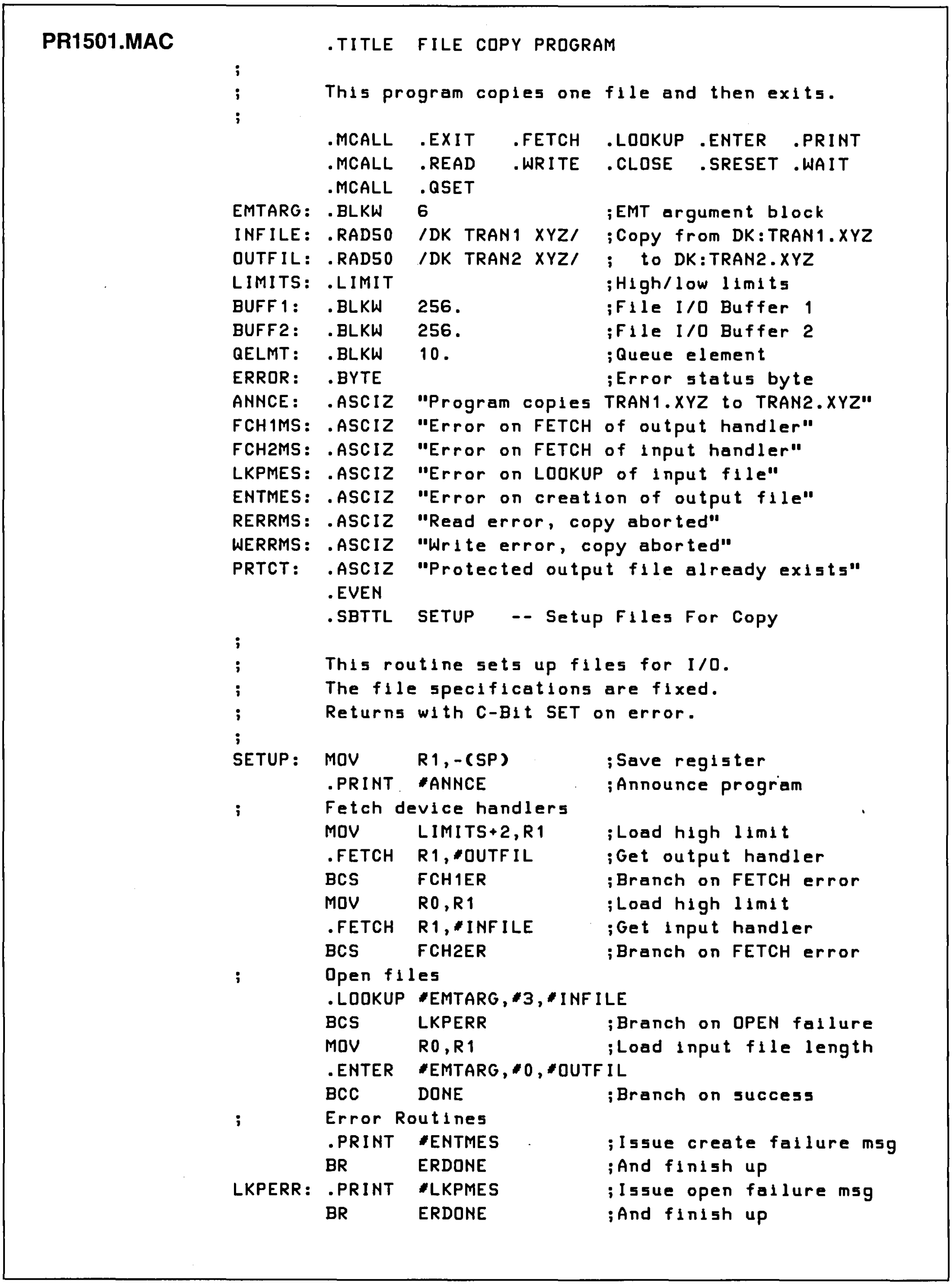
Figure 50.
Flowchart for a Double-buffered Copy Program


FLAG BUFFER 'A' AS INPUT BUFFER
FLAG BUFFER 'B' AS OUTPUT BUFFER
READ INTO INPUT BUFFER
WAIT ON BOTH CHANNELS
SWITCH INPUT AND OUTPUT BUFFERS
WRITE FROM OUTPUT BUFFER

Using Multiple Buffering with Computation

Input and output operations on the same device cannot be overlapped with each other, but both I/O operations can be overlapped with computation. Buffering arrangements differ depending on which operations overlap.

Some types of computation need the input data to be kept until the computation is complete. For this type of computation, you need another buffer to hold results. Other computations can store results by writing over the input data, so you do not need to allocate a separate output buffer. Whether you use separate input and output buffers, of course, will affect the multiple buffering algorithm that you use.



PR1501.MAC

```
        .TITLE  FILE COPY PROGRAM
;
;       This program copies one file and then exits.
;
        .MCALL  .EXIT   .FETCH  .LOOKUP .ENTER  .PRINT
        .MCALL  .READ   .WRITE  .CLOSE  .SRESET .WAIT
        .MCALL  .QSET
EMTARG: .BLKW   6               ;EMT argument block
INFILE: .RAD50  /DK TRAN1 XYZ/  ;Copy from DK:TRAN1.XYZ
OUTFIL: .RAD50  /DK TRAN2 XYZ/  ;  to DK:TRAN2.XYZ
LIMITS: .LIMIT                  ;High/low limits
BUFF1:  .BLKW   256.            ;File I/O Buffer 1
BUFF2:  .BLKW   256.            ;File I/O Buffer 2
QELMT:  .BLKW   10.             ;Queue element
ERROR:  .BYTE                   ;Error status byte
ANNCE:  .ASCIZ  "Program copies TRAN1.XYZ to TRAN2.XYZ"
FCH1MS: .ASCIZ  "Error on FETCH of output handler"
FCH2MS: .ASCIZ  "Error on FETCH of input handler"
LKPMES: .ASCIZ  "Error on LOOKUP of input file"
ENTMES: .ASCIZ  "Error on creation of output file"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine sets up files for I/O.
;       The file specifications are fixed.
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     R1,-(SP)        ;Save register
        .PRINT  #ANNCE          ;Announce program
;       Fetch device handlers
        MOV     LIMITS+2,R1     ;Load high limit
        .FETCH  R1,#OUTFIL      ;Get output handler
        BCS     FCH1ER          ;Branch on FETCH error
        MOV     R0,R1           ;Load high limit
        .FETCH  R1,#INFILE      ;Get input handler
        BCS     FCH2ER          ;Branch on FETCH error
;       Open files
        .LOOKUP #EMTARG,#3,#INFILE
        BCS     LKPERR          ;Branch on OPEN failure
        MOV     R0,R1           ;Load input file length
        .ENTER  #EMTARG,#0,#OUTFIL
        BCC     DONE            ;Branch on success
;       Error Routines
        .PRINT  #ENTMES         ;Issue create failure msg
        BR      ERDONE          ;And finish up
LKPERR: .PRINT  #LKPMES         ;Issue open failure msg
        BR      ERDONE          ;And finish up
```

PR1501.MAC (continued)

```
FCH2ER: .PRINT  #FCH2MS          ;Issue FETCH error msg
        BR      ERDONE           ;And finish up
FCH1ER: .PRINT  #FCH1MS          ;Issue FETCH error msg
ERDONE: SEC                      ;Indicate error occurred
DONE:   MOV     (SP)+,R1         ;Restore R1 (save C-bit)
        RETURN                   ;Return to caller
        .SBTTL  CPYRTN  -- Double-Buffered Copy
;
;       Routine copies data from the opened input file
;       on channel 3 to the output file opened on
;       channel 0.
;       Returns with C-BIT set on error.
;
;       Note: All registers except R0 are preserved.
;
CPYRTN: MOV     R1,-(SP)         ;Save registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        .QSET   #QELMT,#1        ;Allocate a queue element
        CLR     R1               ;Initialize block number
        MOV     #BUFF1,R2        ;R2 ==> input buffer
        MOV     #BUFF2,R3        ;R3 ==> output buffer
        CLRB    ERROR            ;Clear error flag
1$:     .READ   #EMTARG,#3,R2,#256.,R1
        BCC     2$               ;Branch if read succeeded
        TSTB    @#52             ;End-Of-File (EOF)?
        BEQ     EXIT             ;Branch if so
        BR      RDERR            ;Otherwise, process error
2$:     .WAIT   #0               ;Wait for write to finish
        BCS     WERR             ;Branch on error
        .WAIT   #3               ;Wait for read to finish
        BCS     RDERR            ;Branch on error
        MOV     R2,R4            ;Switch input & output
        MOV     R3,R2            ;  buffers
        MOV     R4,R3
        .WRITE  #EMTARG,#0,R3,#256.,R1
        BCS     WERR             ;Branch on write error
3$:     INC     R1               ;Update block number
        BR      1$               ;And read next block
RDERR:  .PRINT  #RERRMS          ;Issue read error msg
        BR      EREXIT           ;And finish up
WERR:   .PRINT  #WERRMS          ;Issue write error msg
EREXIT: DECB    ERROR            ;Set error flag
EXIT:   MOV     (SP)+,R4         ;Restore saved registers
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        .WAIT   #0               ;Wait for last output
        BCC     1$               ;Branch if successful
```

PR1501.MAC (continued)

```
        .PRINT  #WERRMS         ;Issue write error msg
        BR      2$
1$:     TSTB    ERROR           ;Error? (Clear C-BIT)
        BEQ     3$              ;Branch if not
2$:     SEC                     ;Otherwise, set C-BIT
3$:     RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Cleanup For Copy Program
CLSCHN: .CLOSE  #3              ;Close input file
        .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch if succeeded
        .PRINT  #PRTCT          ;Issue protected file msg
PRGCHN:                         ;Purge files
RESET:  .SRESET                 ;Reset system & purge
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Open input & output files
        BCS     1$              ;Branch if failed
        CALL    CPYRTN          ;Copy the input to output
        BCS     1$              ;Branch if failed
        CALL    CLSCHN          ;Close the files
        BR      2$              ;Exit
1$:     CALL    PRGCHN          ;Purge the files
2$:     .EXIT                   ;Exit
        .END    START
```

PR1502.FOR

```
        PROGRAM COPY1
C
C       Program performs a file to file copy and then
C       exits.
C
        LOGICAL*1 SETUP,CPYRTN
        LOGICAL*1 ERROR
C
        ERROR = SETUP()         ! Open files
        IF (ERROR) GO TO 20     ! Stop on setup error
        ERROR = CPYRTN()        ! Copy input to output file
        IF (ERROR) GO TO 20     ! Stop on error
        CALL CLSCHN             ! Success, close channels
        GO TO 30
20      CALL PRGCHN             ! Error, purge channels
30      CALL EXIT
        END
        FUNCTION SETUP
C
C       This routine sets up the files for I/O.
C       The file specifications are fixed in this version.
C
C       Function returns .TRUE. if an error occurred.
C
        LOGICAL*1 SETUP
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
C
C       Channel numbers are common because they are used
C       by CPYRTN, CLSCHN, and PRGCHN.
C
        INTEGER*2 INFILE(4),OUTFIL(4)
        DATA INFILE/2RDK,3RTRA,2RN1,3RXYZ/ ! Input & output
        DATA OUTFIL/2RDK,3RTRA,2RN2,3RXYZ/ !  file specs
C
C       Output introductory message and allocate channels.
C
        CALL PRINT('Program copies TRAN1.XYZ to TRAN2.XYZ')
        INCHN = IGETC()
        OUTCHN = IGETC()
C
C       Fetch device handlers.
C
        IF (IFETCH(OUTFIL(1)) .NE. 0) GO TO 101
        IF (IFETCH(INFILE(1)) .NE. 0) GO TO 102
C
C       Open input file.
C
        LENGTH = LOOKUP(INCHN,INFILE)
        IF (LENGTH .LT. 0) GO TO 103
```

PR1502.FOR (continued)

```
C
C       Create output file.
C
        IF (IENTER(OUTCHN,OUTFIL,LENGTH) .LT. 0) GO TO 104
        SETUP = .FALSE.         ! Return success
        RETURN
C
C       ERROR ROUTINES
C
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.          ! Return with error
        RETURN
        END
        FUNCTION CPYRTN
C
C       Double buffered copy routine.
C
C       Function returns .TRUE. on error.
C       NOTE: Some severe errors will abort the program.
C
        LOGICAL*1 CPYRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFFER(256,2),BLOCK,INPTR,OUTPTR
        LOGICAL*1 FRSTTM        ! "First time through" flag
        DATA FRSTTM/.TRUE./
        IF (.NOT. FRSTTM) GO TO 10 ! Do QSET only once
        IF (IQSET(1) .NE. 0) STOP 'No room for queue element'
        FRSTTM = .FALSE.
10      BLOCK = 0               ! Initialize block number
        INPTR = 1               ! Initialize input buffer #
        OUTPTR = 2              ! Initialize output buffer #
C
C       Read/write loop.  Begin by reading into first buffer.
C
20       IERR = IREAD(256,BUFFER(1,INPTR),BLOCK,INCHN)
        IF (IERR .GE. 0) GO TO 30 ! Successful read
        IF (IERR .EQ. (-1)) GO TO 150 ! End of File
        GO TO 100               ! Otherwise, error
C
C       Wait for read and previous write to finish.
C
30      IF (IWAIT(INCHN) .NE. 0) GO TO 100 ! Error on read
        IF (IWAIT(OUTCHN) .NE. 0) GO TO 100 ! Error on write
```

PR1502.FOR (continued)

```
C
C       Switch read and write buffers.
C
        ITMP = INPTR
        INPTR = OUTPTR
        OUTPTR = ITMP
C
C       Write out buffer just read.
C
        IF (IWRITE(256,BUFFER(1,OUTPTR),BLOCK,OUTCHN) .LT. 0)
     1     GO TO 101
        BLOCK = BLOCK+1            ! Update block number
        GO TO 20                   ! Read next block
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
101     CALL PRINT('Write error, copy aborted')
140     CPYRTN = .TRUE.
        RETURN
C
C       Wait for output to complete and return success.
C
150     IF (IWAIT(OUTCHN) .NE. 0) GO TO 101
        CPYRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        RETURN
        END
```

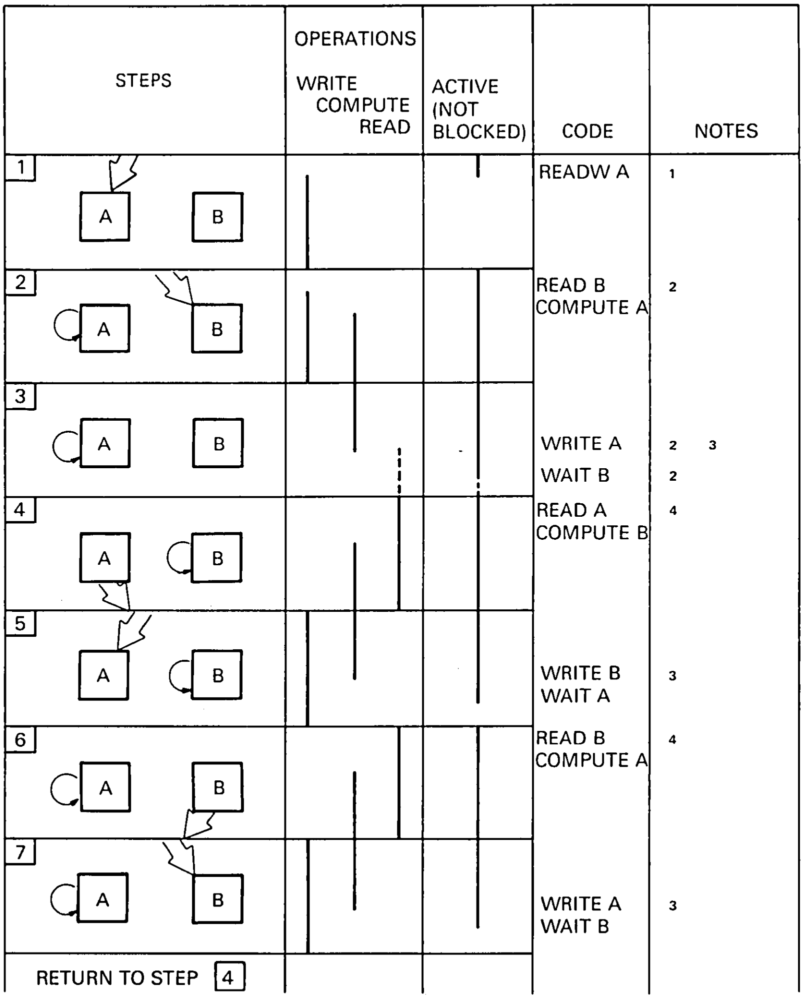
While examining the following programs (PR1503.MAC and PR1504.FOR) and designing multiple buffering algorithms of your own, remember these rules:

1. For maximum speed, start I/O as early as possible to achieve the most overlap.
2. Do not start I/O on a buffer until all other activity on that buffer has completed:
   - Do not start writing from a buffer until the computations that produce the data in the buffer have finished.
   - Do not read into a buffer until the computations that need the old data in the buffer have finished.
   - If an input buffer is also being used as an output buffer, do not read into that buffer until the previous write operation on that buffer has completed.
3. Do not start computation on input data until you are sure that the READ request into that buffer has completed.
4. The WAIT request waits until all I/O on a channel is done. If you are using asynchronous I/O and you want to wait for a specific I/O transfer to complete, do not start a new transfer on that channel until you first execute a WAIT.

Figure 51 shows the sequence of operations in a read/compute/write program. In this example the input and output devices are the same, so the program does not try to overlap input and output. In addition, the computation can store its results in the input buffer. The circular arrow represents the computation operation.

In addition to showing the steps in the operation, figure 51 shows when the program is active—not waiting for I/O to complete. The actual timing depends on the relative speeds of the I/O device and the computation. In the figure, the timing and spacing is exaggerated to emphasize the connections among the different events and to show potential overlaps. Overlap is greatest when the total I/O time is about the same as the computation time. Then you can save

Figure 51.
Double-buffered Read/Compute/Write Sequence



STEPS	OPERATIONS WRITE COMPUTE READ	ACTIVE (NOT BLOCKED)	CODE	NOTES
1			READW A	1
2			READ B COMPUTE A	2
3			WRITE A WAIT B	2 3 2
4			READ A COMPUTE B	4
5			WRITE B WAIT A	3
6			READ B COMPUTE A	4
7			WRITE A WAIT B	3
RETURN TO STEP 4				

as much as 50 percent of the time. The program is blocked only when it must wait for the latest read operation to complete. The numbered notes in figure 51 are described below.

1 Instead of a READW, you can use a READ followed by a WAIT. This makes the program easier to write.

2 Assuming the READ operation takes approximately half the time of the computation, the READ B operation in frame 2 completes while the COMPUTE A is in progress. Therefore, the following WRITE A starts executing immediately and the WAIT B is not necessary—it would return successfully right away. The dotted lines indicate this. WAIT B is included, and the solid lines are positioned to show the sequence of requests and the pattern of the operations that are set up soon after the startup.

3 Each of these WRITEs starts to execute as soon as the previous READ completes.

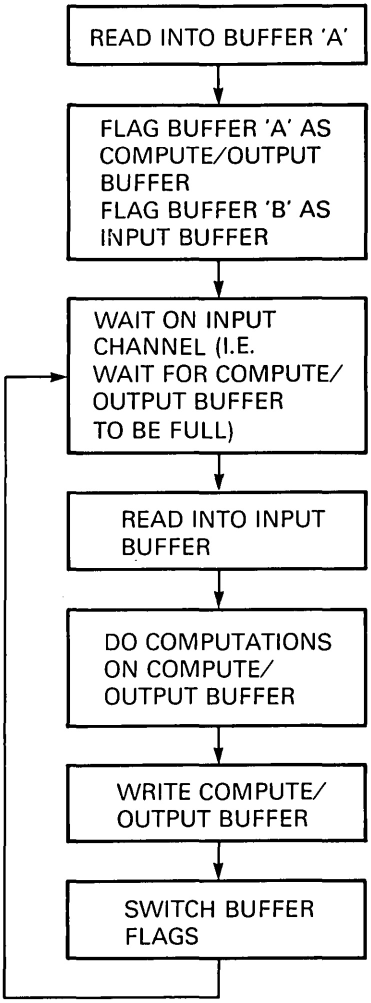
4 Each of these READs starts to execute as soon as the previous WRITE completes.

Figure 52 is a flowchart for the double-buffering method shown in figure 51. The two programs (PR1503.MAC and PR1504.FOR) which follow are examples of this double-buffering method. The programs read a specified file from DK:, sort the bytes of each block, and write the results to another file on DK:. Figure 53 is a flowchart for a program whose computation needs a separate output buffer. To overlap operations here, you need four buffers. At any given time, one pair is being used for I/O (read into the input buffer and write from the output buffer) and one pair is being used in computation. This flowchart also allows for different input and output devices.

The use of the WAIT request is probably the most difficult part of writing a program which uses multiple buffers. In this type of program, the WAIT on the input channel must precede the start of computation on that buffer.

If the input and output devices are not the same, a wait on output channel is needed before the read into input buffer, to make sure that the old buffer contents are written out before the new contents are read in. If the devices are the same (as assumed in these examples), the device handler does not start the READ until it has completed the WRITE.

Figure 52.
Flowchart for Double-buffered Read/Compute/Write Program


READ INTO BUFFER 'A'
FLAG BUFFER 'A' AS COMPUTE/OUTPUT BUFFER FLAG BUFFER 'B' AS INPUT BUFFER
WAIT ON INPUT CHANNEL (I.E. WAIT FOR COMPUTE/ OUTPUT BUFFER TO BE FULL)
READ INTO INPUT BUFFER
DO COMPUTATIONS ON COMPUTE/ OUTPUT BUFFER
WRITE COMPUTE/ OUTPUT BUFFER
SWITCH BUFFER FLAGS

PR1503.MAC

```
        .TITLE  COMPUTATION & I/O EXAMPLE PROGRAM
;
;       This program demonstrates asynchronous double
;       buffered computation.  The program reads data
;       from an input file, performs computation on
;       the data read, and writes the transformed
;       data to the output file.
;
        .MCALL  .EXIT   .FETCH  .LOOKUP .ENTER  .PRINT
        .MCALL  .READ   .WRITE  .CLOSE  .SRESET .WAIT
        .MCALL  .QSET
EMTARG: .BLKW   6               ;EMT argument block
INFILE: .RAD50  /DK TRAN1 XYZ/  ;Copy from DK:TRAN1.XYZ
OUTFIL: .RAD50  /DK TRAN2 XYZ/  ;  to DK:TRAN2.XYZ
LIMITS: .LIMIT                  ;High/low program limits
BUFF1:  .BLKW   256.            ;File I/O Buffer 1
BUFF2:  .BLKW   256.            ;File I/O Buffer 2
QELMT:  .BLKW   10.             ;Queue element
ERROR:  .BYTE                   ;Error status byte
EOF:    .BYTE                   ;End of File flag
ANNCE:  .ASCIZ  "Program copies TRAN1.XYZ to TRAN2.XYZ"
FCH1MS: .ASCIZ  "Error on FETCH of output handler"
FCH2MS: .ASCIZ  "Error on FETCH of input handler"
LKPMES: .ASCIZ  "Error on LOOKUP of input file"
ENTMES: .ASCIZ  "Error on creation of output file"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine sets up files for I/O.
;       The file specifications are fixed.
;
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     R1,-(SP)        ;Save register
        .PRINT  #ANNCE          ;Announce program
;       Fetch device handlers
        MOV     LIMITS+2,R1     ;Load high limit
        .FETCH  R1,#OUTFIL      ;Get output handler
        BCS     FCH1ER          ;Branch on FETCH error
        MOV     R0,R1           ;Load high limit
        .FETCH  R0,#INFILE      ;Get input handler
        BCS     FCH2ER          ;Branch on FETCH error
;       Open files
        .LOOKUP #EMTARG,#3,#INFILE
        BCS     LKPERR          ;Branch if open failed
        MOV     R0,R1           ;Load input file length
```

PR1503.MAC (continued)

```
        .ENTER  #EMTARG,#0,#OUTFIL
        BCC     DONE            ;Branch if successful
;       Error Routines
        .PRINT  #ENTMES         ;Issue create failure msg
        BR      ERDONE          ;And finish up
LKPERR: .PRINT  #LKPMES         ;Issue open failure msg
        BR      ERDONE          ;And finish up
FCH2ER: .PRINT  #FCH2MS         ;Issue FETCH error
        BR      ERDONE          ;And finish up
FCH1ER: .PRINT  #FCH1MS         ;Issue FETCH error
ERDONE: SEC                     ;Indicate error occurred
DONE:   MOV     (SP)+,R1        ;Restore R1 (save C-bit)
        RETURN                  ;Return to caller
        .SBTTL  CMPRTN  -- Computation Routine
;
;       Routine assumes the input file is opened on
;       channel 3 and the output on channel 0.
;       Returns with C-BIT SET on error.
;
;       Note: All registers except R0 are preserved.
;
CMPRTN: MOV     R1,-(SP)        ;Save registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
        .QSET   #QELMT,#1       ;Allocate a queue element
BEG:    .READ   #EMTARG,#3,#BUFF1,#256.,#0
        BCC     INIT            ;Branch if read ok
        TSTB    @#52            ;End-of-File?
        BEQ     EXIT            ;Branch if so, all done
        BR      RDERR           ;Issue read error
INIT:   MOV     #1,R1           ;Load read block number
        CLR     R5              ;Load write block number
        MOV     #BUFF2,R2       ;R2 --> input buffer
        MOV     #BUFF1,R3       ;R3 --> output buffer
        CLRB    ERROR           ;Clear error flag
        CLRB    EOF             ;Clear EOF flag
SLOOP:  .WAIT   #3              ;Wait for input to finish
        BCS     RDERR           ;Branch on error
        .READ   #EMTARG,#3,R2,#256.,R1
        BCC     COMP            ;Branch if read succeeded
        TSTB    @#52            ;End-Of-File (EOF)?
        BNE     RDERR           ;Branch if fatal error
        INCB    EOF             ;Set EOF encountered flag
;
;       Perform computation on previously read block.
;       Computation in this program consists of shifting
;       each data element one place to the right (a
;       simple scaling operation).
;
```

PR1503.MAC (continued)

```
COMP:   MOV     #256.,R0        ;Initialize counter
        MOV     R3,R4           ;Copy buffer address
LOOP:   ASR     (R4)+           ;Shift right one bit (/2)
        DEC     R0              ;Decrement loop counter
        BNE     LOOP            ;Branch if not done
;
;       Write out buffer on which computation has just
;       been performed.
;
        .WRITE  #EMTARG,#0,R3,#256.,R5
        BCS     WERR            ;Branch on write error
        TSTB    EOF             ;EOF on last read?
        BGT     EXIT            ;Branch if so
        MOV     R2,R4           ;Otherwise, switch input
        MOV     R3,R2           ;  and output buffer
        MOV     R4,R3
        INC     R1              ;Update input block #
        INC     R5              ;Update output block #
        BR      SLOOP           ;And continue
;
;       Error messages and cleanup.
;
RDERR:  .PRINT  #RERRMS         ;Issue read error msg
        BR      EREXIT          ;And finish up
WERR:   .PRINT  #WERRMS         ;Issue write error msg
EREXIT: DECB    ERROR           ;Set error flag
EXIT:   MOV     (SP)+,R5        ;Restore saved registers
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        .WAIT   #0              ;Wait for last output
        BCC     1$              ;Branch if successful
        .PRINT  #WERRMS         ;Issue write error msg
        BR      2$
1$:     TSTB    ERROR           ;Set C-Bit? (clear it)
        BEQ     3$              ;Branch if not
2$:     SEC                     ;Otherwise, set it
3$:     RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Cleanup For Copy Program
CLSCHN: .CLOSE  #3              ;Close input file
        .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch on success
        .PRINT  #PRTCT          ;Issue protected file msg
PRGCHN:                         ;Purge files
RESET:  .SRESET                 ;Reset (purge files)
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Open files
        BCS     1$              ;Branch on error
```

PR1503.MAC (continued)

```
        CALL    CMPRTN          ;Transfer the file
        BCS     1$              ;Branch on error
        CALL    CLSCHN          ;Close the files
        BR      2$
1$:     CALL    PRGCHN          ;Purge the files
2$:     .EXIT
        .END    START
```

PR1504.FOR

```
      PROGRAM ACOMP
C
C     Asynchronous Double-Buffered I/O w/Computation.
C
C     This program reads data from the input file,
C     performs computation on the data read, and
C     writes the transformed data to the output file.
C     This program uses asynchronous I/O to allow the
C     computation to occur during I/O operations.
C
      LOGICAL*1 SETUP,CMPRTN
      LOGICAL*1 ERROR
C
      ERROR = SETUP()           ! Open files
      IF (ERROR) GO TO 20       ! Stop on setup error
      ERROR = CMPRTN()          ! Copy file
      IF (ERROR) GO TO 20       ! Stop on error
      CALL CLSCHN               ! Close channels
      GO TO 30                  ! Exit
20    CALL PRGCHN               ! Purge channels
30    CALL EXIT
      END
      FUNCTION SETUP
C
C     This routine sets up the files for I/O.
C     The file specifications are fixed in the version.
C
C     Function returns .TRUE. if an error occurred.
C
      LOGICAL*1 SETUP
      INTEGER*2 INCHN,OUTCHN
      COMMON /CHNNLS/ INCHN,OUTCHN
C
C     Channel numbers in common because they are used
C     by CMPRTN, CLSCHN, and PRGCHN.
C
      INTEGER*2 INFILE(4),OUTFIL(4)
      DATA INFILE/2RDK,3RTRA,2RN1,3RXYZ/ ! Input & output
      DATA OUTFIL/2RDK,3RTRA,2RN2,3RXYZ/ !  file specs
C
C     Output introductory message and allocate channels.
C
      CALL PRINT('Program copies TRAN1.XYZ to TRAN2.XYZ')
      INCHN = IGETC()
      OUTCHN = IGETC()
C
C     Fetch needed device handlers.
C
      IF (IFETCH(OUTFIL(1)) .NE. 0) GO TO 101
      IF (IFETCH(INFILE(1)) .NE. 0) GO TO 102
```

PR1504.FOR
(continued)

```
C
C       Open input file.
C
        LENGTH = LOOKUP(INCHN,INFILE)
        IF (LENGTH .LT. 0) GO TO 103
C
C       Create output file.
C
        IF (IENTER(OUTCHN,OUTFIL,LENGTH) .LT. 0) GO TO 104
        SETUP = .FALSE.          ! Return success
        RETURN
C
C       ERROR ROUTINES
C
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.           ! Return error
        RETURN
        END
        FUNCTION CMPRTN
C
C       Double-buffered computation routine.
C
C       Function returns .TRUE. on error.
C       NOTE: Some severe errors will abort the program.
C
        LOGICAL*1 CMPRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFFER(256,2),BLOCK,INPTR,OUTPTR
        LOGICAL*1 FRSTTM         ! Once only flag
        DATA FRSTTM/.TRUE./
        IF (.NOT. FRSTTM) GO TO 10 ! Do QSET only once
        IF (IQSET(1) .NE. 0) STOP 'No room for queue element'
        FRSTTM = .FALSE.
C
C       Begin by reading into buffer 1.
C
10      IERR = IREAD(256,BUFFER(1,1),0,INCHN)
        IF (IERR .GE. 0) GO TO 20 ! Successful read
        IF (IERR .EQ. (-1)) GO TO 150 ! EOF means all done
        GO TO 100                ! Read error
C
C       Initialize block numbers and flags.
C
```

PR1504.FOR (continued)

```
20      BLOCK = 1               ! Initialize block number &
        INPTR = 2               !  input buffer number &
        OUTPTR = 1              !  output buffer number
C
C       Loop: Wait for input to complete, compute,
C             output.
C
30      IF (IWAIT(INCHN) .NE. 0) GO TO 100 ! Error on read
C
C       Read next block into input buffer.
C
        IERR = IREAD(256,BUFFER(1,INPTR),BLOCK,INCHN)
        IF (IERR .LT. (-1)) GO TO 100   ! Error on read
C
C       Perform computation on output buffer while read is
C       in progress.  Computation consists of dividing each
C       element of the buffer by 2 (scaling operation).
C       Normally, a subroutine would be called to do the
C       computation.
C
50      DO 60 I=1,256
        BUFFER(I,OUTPTR)=BUFFER(I,OUTPTR)/2
60      CONTINUE
C
C       Write out buffer on which computations have just
C       completed.
C
        IF (IWRITE(256,BUFFER(1,OUTPTR),BLOCK-1,OUTCHN) .LT. 0)
     1     GO TO 101            ! Error on write
C
C       Check if last read resulted in EOF.
C
        IF (IERR .EQ. (-1)) GO TO 150   ! Copy completed.
C
C       Otherwise, switch buffers and advance block number.
C
        ITMP = INPTR
        INPTR = OUTPTR
        OUTPTR = ITMP
        BLOCK = BLOCK+1
        GO TO 30                ! Repeat
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
101     CALL PRINT('Write error, copy aborted')
140     CMPRTN = .TRUE.
        RETURN
```

PR1504.FOR
(continued)

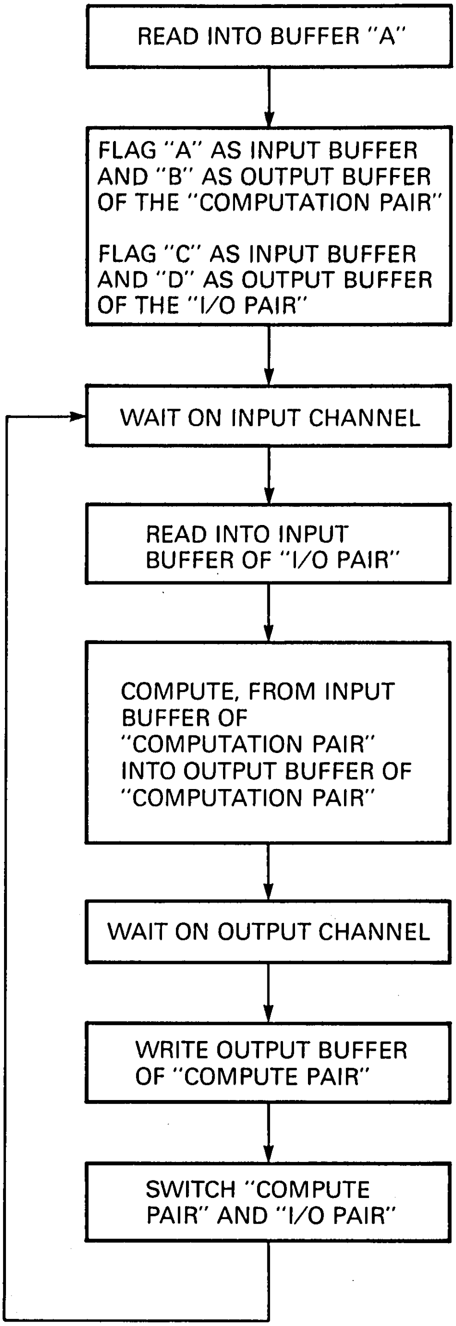
```
C
C       Wait for last output to complete and return.
C
150     IF (IWAIT(OUTCHN) .NE. 0) GO TO 101
        CMPRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        RETURN
        END
```

Event-driven I/O Requests

The formats of event-driven I/O requests for MACRO–11 are:

```
.READC   area,chan,buff,wcnt,crtn,blk
.WRITC   area,chan,buff,wcnt,crtn,blk
```

Figure 53.
Flowchart for Multiple Buffering with Separate Input and Output Buffers


READ INTO BUFFER "A"
FLAG "A" AS INPUT BUFFER AND "B" AS OUTPUT BUFFER OF THE "COMPUTATION PAIR"
FLAG "C" AS INPUT BUFFER AND "D" AS OUTPUT BUFFER OF THE "I/O PAIR"
WAIT ON INPUT CHANNEL
READ INTO INPUT BUFFER OF "I/O PAIR"
COMPUTE, FROM INPUT BUFFER OF "COMPUTATION PAIR" INTO OUTPUT BUFFER OF "COMPUTATION PAIR"
WAIT ON OUTPUT CHANNEL
WRITE OUTPUT BUFFER OF "COMPUTE PAIR"
SWITCH "COMPUTE PAIR" AND "I/O PAIR"

In these requests "crtn" is the address of the completion routine that is to be run when this I/O transfer completes. Other arguments are identical to the other queued I/O requests.

The formats of event driven I/O requests in FORTRAN IV are:

```
IERR=IREADC (wcnt,buff,blk,chan,crtn)

IERR=IWRITC (wcnt,buff,blk,chan,crtn)

IERR=IREADF (wcnt,buff,blk,chan,area,crtn)

IERR=IWRITF (wcnt,buff,blk,chan,area,crtn)
```

In these requests, "crtn" is the name of the completion routine to be run when this I/O transfer completes, and "area" is the name of a four-word area you must have in your program if you use IREADF or IWRITF. All other arguments are the same as for the other queued I/O requests.

Use IREADC and IWRITC if you are using completion routines written in MACRO–11. Use IREADF and IWRITF if you are using completion routines written in FORTRAN IV. If you use IREADF or IWRITF, remember to include the area argument. This may be either an array or a variable. Do not modify or use this argument again until the completion routine has started execution. The USR must not swap over the area argument.

The routine that issues an IREADC, IWRITC, IREADF, or IWRITF request must also issue an EXTERNAL statement for the name of the completion routine(s) being passed to any of these requests.

Completion Routines

The key to event-driven I/O is the completion routine. Completion routines are called as subroutines by the monitor. When a job issues an event-driven I/O request and that request completes, the completion routine for the request runs, regardless of the state of the job that issued the original request.

A completion routine can perform almost any operation needed when an I/O transfer completes. It can:

- Issue a new I/O request
- Set a flag indicating completion of the I/O request, for use by another routine of the job
- Check for end-of-file or hardware errors that may have occurred during the transfer
- Do limited processing on the input data

Later in this chapter, we will discuss what completion routines cannot do.

You should try to keep completion routines as short as possible. A long completion routine delays execution of the main job and may delay execution of other completion routines. When possible, have the completion routine set some flags, or record a few items of information, but leave any heavy computation to the main job.

Priority Levels and Scheduling

User-written routines under RT–11 have relative priorities that RMON uses when determining which routine to run next. In the SJ applications previously discussed, all routines within a job have the same priority, and a routine runs only if it is called or jumped to by another routine running in that job. Under the FB monitor, the routines of the foreground job have a higher priority than those of the background job. Routines in each job run independently of those in the other job, except that those in the foreground job can interrupt those of the background job.

When a job starts a completion routine, two priority levels exist for that job. The main-line routine or program, and any routines that the main line calls or jumps to, are said to execute at main level. The completion routine that you specify in an event-driven I/O request, and any routines that the completion routine may call or jump to, execute at completion level. Any routines executing at completion level have a higher priority than the main level of the job that issued the I/O request.

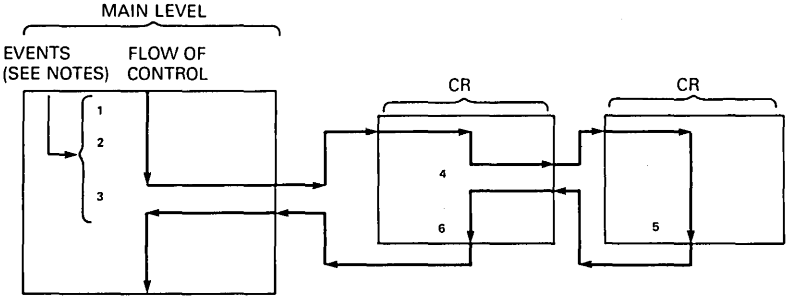
Scheduling Completion Routines under the SJ Monitor

Under the SJ monitor, a completion routine is called as soon as the I/O transfer completes, regardless of what was interrupted by the I/O completion (one completion routine can interrupt another). This arrangement is called a system of nested completion routines. One could say that completion routines under the SJ monitor are executed: last in—first out (LIFO).

If two or more outstanding event-driven I/O requests are using the same completion routine, the LIFO method used by the SJ monitor can cause that routine to be called again before it has returned and errors can occur. Figure 54 shows how this situation can occur. As you study the figure, please refer to the following notes which describe the sequence of events.

1	Main-level routine issues event-driven request with CR as completion routine.	Transfer is queued to device handler.
2	Main-level routine issues second event-driven request with CR as completion routine.	Transfer is queued to device handler.
3	First I/O request completes.	CR is called (first call).
4	Second I/O request completes.	CR is called (second call, reentrance occurs at this point and errors can occur).
5	Second call of CR completes, returns.	First call of CR continues at the point at which it was interrupted.
6	First call of CR completes, returns.	Main-level routine continues at the point at which it was interrupted.

Figure 54.
Reentrance of a Completion Routine under the SJ Monitor



Reentrance (as shown in figure 54) can cause execution errors if the reentered routine has not been written to allow for reentrance. If the second call of the routine (events 4 and 5) changes the values of any locations set by the first call (between event 3 and event 4), then, when control returns to the first call (event 5), the routine will be operating on incorrect data.

To prevent reentrance-caused errors, you can write MACRO–11 reentrant completion routines that return the same values as those on entry. To do this, save on the stack the old contents of any locations that are to be changed during the course of the routine, and restore them before returning from the routine. Another method is to use stack storage only.

FORTRAN IV routines cannot be made reentrant. If your completion routine is not reentrant, you must make sure that only one request for any given completion routine is outstanding at any given time. You can use a flag that is set while the completion routine is outstanding. Check the flag before issuing the I/O request. If the flag is clear, set it and issue the request. (If it is not clear, take any appropriate action.) The completion routine should clear the flag as its last action before returning.

Scheduling Completion Routines under the FB Monitor

When an event-driven I/O request completes under the FB monitor, the completion routine for the request is inserted into a completion queue, which is a list of completion routines to be run by the scheduler of RMON, according to the scheduling algorithm.

The completion queue is maintained by RMON using a linked list of structures called completion queue elements. A completion queue element is the I/O queue element from the I/O request, modified by the RMON before being inserted into the completion queue.

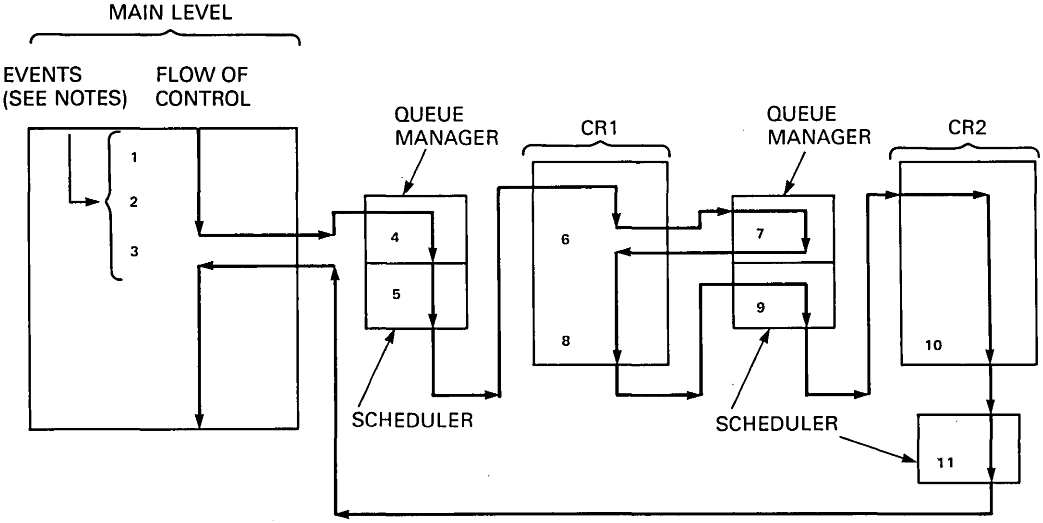
The completion queue for a job is maintained and used as a first in—first out (FIFO) queue, so the completion routines of each job run sequentially, in the order in which the I/O requests completed. Management of a job's completion queue is shared by two parts of RMON—the queue manager and the scheduler. The queue manager is called by a device handler when I/O completes. The queue manager changes the I/O queue element to a completion queue element and adds it to the completion queue. The scheduler is called when the system may need to shift execution from one priority level to another (for example, when a level becomes blocked or unblocked, or when a completion routine completes). The scheduler calls the completion routine as a subroutine.

Figure 55 shows the FIFO scheduling of completion routines under the FB monitor, including the functions of the queue manager and the scheduler. This figure shows the scheduling of completion routines for an event-driven I/O request that completes while another completion routine is running on the same job. Completion routines cannot interrupt one another under the FB monitor, as they can under the SJ monitor. As you study figure 55, read the following notes which describe the events shown in the figure.

1	Main-level routine issues event-driven request with CR1 as completion routine.	Transfer is queued to device handler.

2	Main-level routine issues event-driven request with CR2 as completion routine.	Transfer is queued to device handler.
3	First request completes.	Device handler informs queue manager (part of RMON) of I/O completion.
4	Queue manager inserts completion queue element for CR1 into completion queue.	
5	Scheduler recognizes outstanding completion routine CR1.	CR1 is called.
6	Second I/O request completes.	Device handler informs queue manager of I/O completion.

Figure 55.
FIFO Scheduling of Completion Routines under the FB Monitor



7	Queue manager inserts queue element into completion queue; recognizes that a completion routine (CR1) is in progress.	Control returns to CR1.
8	CR1 completes.	Control passes to scheduler.
9	Scheduler recognizes another outstanding completion routine, CR2.	CR2 is called.
10	CR2 completes.	Control passes to scheduler.
11	Scheduler recognizes that there are no more completion routines outstanding.	Control returns to main level.

In figure 55, there are two completion queues, one for each job. With two jobs active under the FB monitor, there are four possible priority levels at which a user-written routine may execute. These are shown in table 15. All background routines have lower priority than any foreground routine. Background completion level is at a lower priority than foreground main level.

A job can become blocked either at main level or at completion level. Even if the main level is blocked, its completion routines are allowed to run; however, the re-

Table 15.
Priority Levels in a Foreground/Background Monitor

Job	Level	Priority
Foreground	Foreground completion Foreground main	Highest
System jobs (1 to 6)	System job n completion System job n main	
Background	Background completion Background main	Lowest

verse is not true. If a job becomes blocked at completion level, RMON does not allow that job's main level to run. This assures that once a completion routine has started to execute, the main level does not run again until the completion routine has completed. In this way you are assured that the main level of a job never executes while some completion routine is performing operations.

With the single exception that, if its completion level is blocked, the main level of a job is not allowed to run, RMON always runs the highest priority level that is not blocked. RMON may shift control from one level to another when any of a number of events occur, including:

- The current executing routine issues a programmed request that causes the routine to become blocked.
- A new completion routine is queued at a priority level higher than the current executing routine.
- A completion routine returns, and there are no other completion routines at that level.
- A routine at a higher priority than the current executing routine becomes unblocked.
- The current executing job exits or is aborted.

Any of these events may start an operation of RMON known as a scheduling pass. In this operation, RMON examines the blocking conditions of jobs (in decreasing order of priority), identifies the highest priority level that can be run, and executes that level. Because the FB scheduler serializes the execution of completion routines, there is no chance of a routine being reentered under the FB monitor.

Conventions for Writing Completion Routines

Certain RT–11 conventions govern the writing of a completion routine. In MACRO–11 these conventions are:

- A completion routine is always called with a JSR PC and is returned with an RTS PC.

- Before calling a completion routine, RMON puts the value of the channel status word (CSW) for this request in R0, and the channel number of the request in R1. Your completion routine should check the CSW for indications of hardware error (bit 0) or end-of-file (bit 13). It can use the channel number to distinguish I/O requests if the same completion routine is used on more than one channel.
- A routine can use R0 and R1. Any other registers must be saved and restored.

In FORTRAN IV the conventions are:

- A completion routine is defined using a SUBROUTINE statement. Use RETURN to end execution of the routine.
- This completion routine may have up to two arguments, of INTEGER data type. The SUBROUTINE statement can define up to two arguments. RMON puts the value of the channel status word (CSW) in the first argument, and the channel number of the request in the second. Your completion routine should check the CSW for indications of hardware error or end-of-file. It can use the channel number to distinguish I/O requests if the same routine is used on more than one channel.
- Remember that the FORTRAN IV routine that issues IREADC, IWRITC, IREADF, or IWRITF must declare the names of any completion routines in an EXTERNAL statement.
- A completion routine for IREADC or IWRITC, written in MACRO–11, must obey all conventions that apply to assembly language completion routines.

Restrictions on Completion Routines

RT–11 places certain restrictions on completion routines because they execute independent of the main level. Execution errors may occur if you do not follow these restrictions. The following restrictions apply to both MACRO–11 and FORTRAN IV:

- A completion routine must not reside in the area into which the USR swaps. If the main level issues a request that needs the USR, and I/O completes while the USR is in memory, control does not transfer to the completion routine (which is swapped out) but to some part of the USR. USR swapping is discussed in chapter 18, "Using Memory."
- A completion routine must not execute any request that needs the USR, as the USR is not reentrant.
- Under the SJ monitor, either a completion routine must be reentrant, or you must make sure that only one request using that completion routine is active at any given time.
- Any routine callable from both main level and completion level must be reentrant.

The following restrictions apply to FORTRAN IV programs only:

- When using OTS I/O (FORTRAN IV READ and WRITE statements), the first I/O statement to any specific logical unit number, or the OPEN statement, causes the OTS to execute a programmed request that needs the USR. Therefore, neither of these types of statement should appear in a completion routine. Also, a completion routine should not contain a CLOSE statement or a call to the CLOSE subroutine.
- Do not call IGETC or IFREEC from a completion routine.
- The section of the OTS that supports FORTRAN IV subprogram calls and returns is not reentrant. This means that no subroutine or function written in FORTRAN IV should be callable both from main and

from completion level. It also means that under the SJ monitor, only one completion routine written in FORTRAN IV should be outstanding at any given time.

Applications for Event-driven I/O

Event-driven I/O is most useful when you are working with devices whose data rates are in some way random. Let us look at some examples.

Suppose that your program performs I/O on a number of channels and you have no way of knowing which channel will be the first to complete. Asynchronous I/O is of no use here because you cannot select any one channel to wait on, as a different channel may complete first. Instead, you can use a completion routine to indicate completion on any channel when it occurs, and to inform the main line.

You can use event-driven I/O to overlap input operations on different channels only if those channels use different device handlers, or if the handler queues requests internally. Otherwise, the I/O system serializes requests to the handler.

Now, consider a program handling input from a device that has a high, but variable, data rate. To make sure that no data is lost because of a delay in issuing a new I/O request after the old one completes, you must have an input request outstanding on the channel at all times. You can implement this by using event-driven requests and having the completion routine issue a new input request as its first action.

One problem presented by a device with high but variable data rate, is that the input data rate may temporarily exceed that of the computation routines. If this occurs, you may need a carefully controlled system of multiple buffers to store the input data until the data rate drops and the computation routines at main level have a chance to process the data.

A similar condition is one in which the rate of input and/or computation temporarily exceeds the rate of the output device. If there is more data to be output, this too, would need multiple buffers and a completion routine that issues a new output request as soon as the old one completes.

Finally, event-driven I/O is the only choice if the main level of your program has any operation it must perform continuously while waiting for I/O completion. For example, your program may be doing real-time process control using some set of control parameters. These parameters may change as a result of some input, but your program cannot issue synchronous requests, or use WAIT to wait for the input because it would then stop executing its control procedure. By using event-driven I/O, the main level can continue control using the old parameters, until new ones come in.

The applications described above usually apply to real-time I/O devices instead of the standard peripheral devices supported by RT-11. In fact, the major application of event-driven I/O is to support nonstandard devices.

Practice 15–1

1. Write a program that:

   a. Reads a data file from a disk one block at a time.

   b. Finds the maximum value in the block, interpreting each word as a signed integer.

   c. Writes a block to an output file that contains the difference between each word in the input data block, and the maximum of that block.

2. Use a multiple-buffering algorithm to give maximum throughput. For more practice designing your own algorithm, you can use a triple-buffered algorithm in order for input, computation, and output to all proceed concurrently.

Practice 15–2

For each programming problem described below, indicate which mode of I/O (synchronous, asynchronous, or event-driven) should be used. Justify your answer. In each problem, assume that standard device handlers are available for each of the devices mentioned.

1. Your program is engaged in process control. It is responsible for maintaining the temperature and pressure of a system within certain preset limits. A pressure sensing device PS:, and a temperature sensing device TS:, provide input data when the pressure or temperature changes. If either value goes past its present limit, your program is to respond by sending new control parameters to a control device CL:. What mode would you use for the input requests for devices PS: and TS:?
2. Your RT–11 system receives data from an input device IN:. It must assemble this data into packets that are to be sent across a communications line to another processor, which appears to your program as a device PR:. The data rate of IN: is expected to be very close to, but will not exceed that of PR:. Which I/O mode would you use?
3. Your program generates reports. It takes input from a file on a high-speed disk, formats it, and prints the formatted report on the terminal. Which I/O mode would you use?

References

RT–11 Programmer's Reference Manual. Chapter 1 describes conventions and restrictions on completion routines.

RT–11 Software Support Manual. Chapter 3 provides additional information on the completion queue element.

16

16

Scheduling and Blocking

A job can be waiting for an I/O operation to complete for much of its time, even when using nonsynchronous I/O, because peripheral devices operate much more slowly than the processor. Under the Single Job monitor, the CPU is idle during these waiting periods. Under the Foreground/Background monitor, however, you can make more efficient use of your processor by running one (background) job when a second (foreground) job is waiting for some external event. With the proper understanding of RT–11 job control, you can write application programs in ways that best utilize the foreground/background capabilities of the system.

This chapter explains the basic concepts of RT–11 job management and describes the programmed requests that can increase your control over job execution under the Foreground/Background monitor. The MACRO–11 requests discussed include: .SPND, .RSUM, .CNTXSW, .WAIT, and .TWAIT. The FORTRAN IV requests discussed are SUSPND and RESUME.

When you have completed this chapter, you will be able to write code to block a job until completion of a specific I/O request. You will also learn to design an application system, separating tasks for foreground and background jobs in a way that maximizes system throughput.

Scheduling

When you issue a KMON command to run a job, that job becomes active. It stays active until it exits, or is aborted by an error or ⟨CTRL/C⟩. The SJ monitor supports only one active job. A standard Foreground/Background monitor supports two.

The foreground/background scheduling concepts discussed in this chapter are the same for both Foreground/Background and Extended Memory monitors, so the term "foreground/background" is used to apply to both in this chapter.

The system job option, available through system generation, enables a Foreground/Background monitor to support up to six system jobs in addition to the foreground and background jobs.

Regardless of how many jobs are active under a monitor, only one can use the CPU at a time because the CPU can execute only one instruction at a time. The resident monitor RMON determines which job is given the use of the CPU. The process of allocating the CPU among active jobs is called scheduling. The three basic concepts of scheduling under the RT–11 Foreground/Background monitors are:

1. Priority. Each job has a fixed priority, which RMON uses to determine which job will get the CPU. RMON always runs (gives use of the CPU to) the highest priority job that is capable of running.
2. Blocking. When a job is not able to run because it needs some external event, such as completion of a read request, it is said to be blocked. RMON recognizes when a job is blocked and allows a lower priority job to run.
3. Context. To help keep track of active jobs, RMON keeps important data about each job. This data is the context in which the job is said to run. When RMON transfers use of the CPU from one job to another, it first performs an operation known as a context switch.

The next three sections explain these concepts in more detail.

Priorities

Under a Foreground/Background monitor, every job has a software priority that is fixed at the time that the job is initiated. The background job has the lowest priority, and the foreground job has the highest. If the monitor has system job support, all system jobs have priorities between those of the background and foreground jobs.

This system of software priorities is used by RMON to determine which job to run. These priorities are different from the hardware priorities used for servicing interrupts on PDP–11 computers.

If a job requests completion routines, it can be run at two levels: main level and completion level. The completion level has a higher priority than the main level (but a lower priority than the main level of the next higher priority job). This priority structure is shown in table 15.

The foreground and background job slots are for application and utility programs. The system jobs slots are reserved for special programs provided by Digital. The programs currently provided with RT–11 are the device queue package, error log programs, transparent spooler, and virtual communications program.

Blocking

Levels can be in one of two states, blocked or runnable. A blocked level is not able to proceed until some condition is met. A level not blocked is runnable. Usually RMON blocks a job when the job issues a programmed request that for some reason cannot be completed immediately. Some requests that cannot be completed return an error to the program. If the error is fatal, RMON aborts the program. Normally RMON blocks a job if there is a delay and aborts the program if there is a serious error. RMON unblocks the job when the conditions have changed, and it can complete the programmed request. Some examples of this type of blocking are:

- Terminal I/O requests. The job is blocked if no characters are available on input, or no space in the buffer is available on output. (If bit 6 of the job status word is set, the job is not blocked.)
- Synchronous I/O requests. The job is blocked until the transfer is complete.
- USR requests. If the USR is owned by another job, the requesting job is blocked until the USR is available again. This situation is discussed in chapter 18, "Using Memory."
- I/O requests. If a job issues an I/O request and no queue element is available, the job is blocked until a queue element is available.

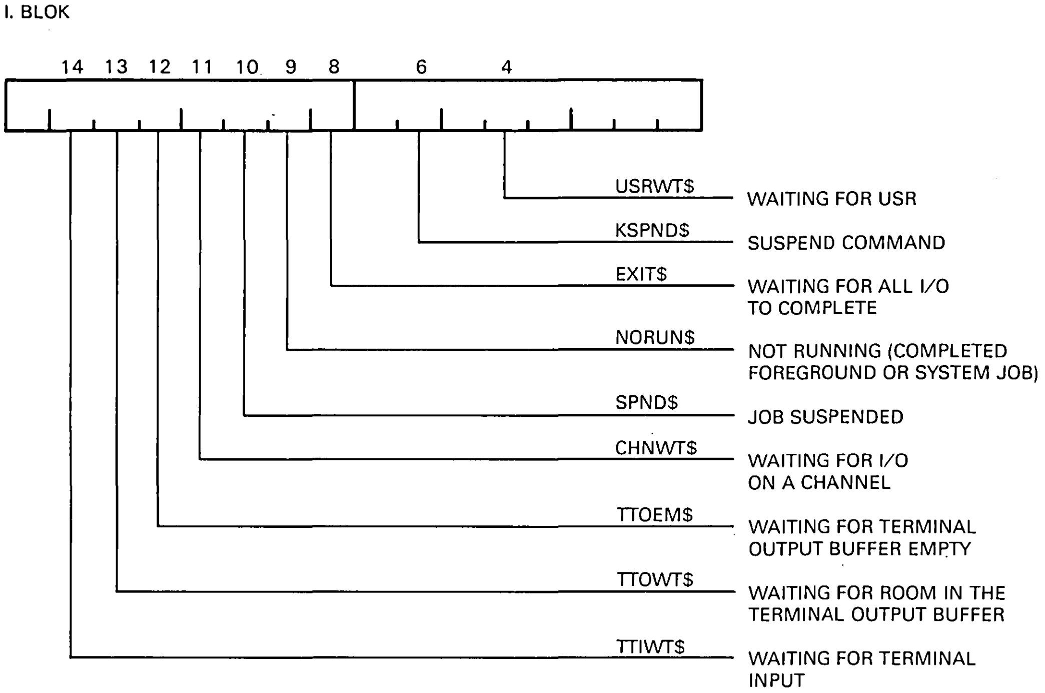
Some programmed requests can block a job, for example:

- The .WAIT request blocks a job until an I/O channel is clear.
- The .TWAIT request blocks a job for a specified period of time.
- The .SPND request blocks a job until it issues a .RSUM to resume.

The word I.BLOK in the job's impure area is a flag word used to indicate blocking conditions for the job; the bits indicate what caused the job to be blocked. The only blocking condition not flagged in I.BLOK is blocking because no queue element is available; this condition is processed by another part of RT–11. Figure 56 shows all the conditions flagged by I.BLOK. The bits not used are reserved for future use.

When RT–11 needs to block a job, it sets a bit in I.BLOK. The scheduling routine in RMON uses this word

Figure 56.
Blocking Conditions Flagged by I.BLOK


I. BLOK
14
13
12
11
10
9
8
6
4
USRWT$
WAITING FOR USR
KSPND$
SUSPEND COMMAND
EXIT$
WAITING FOR ALL I/O TO COMPLETE
NORUN$
NOT RUNNING (COMPLETED FOREGROUND OR SYSTEM JOB)
SPND$
JOB SUSPENDED
CHNWT$
WAITING FOR I/O ON A CHANNEL
TTOEM$
WAITING FOR TERMINAL OUTPUT BUFFER EMPTY
TTOWT$
WAITING FOR ROOM IN THE TERMINAL OUTPUT BUFFER
TTIWT$
WAITING FOR TERMINAL INPUT

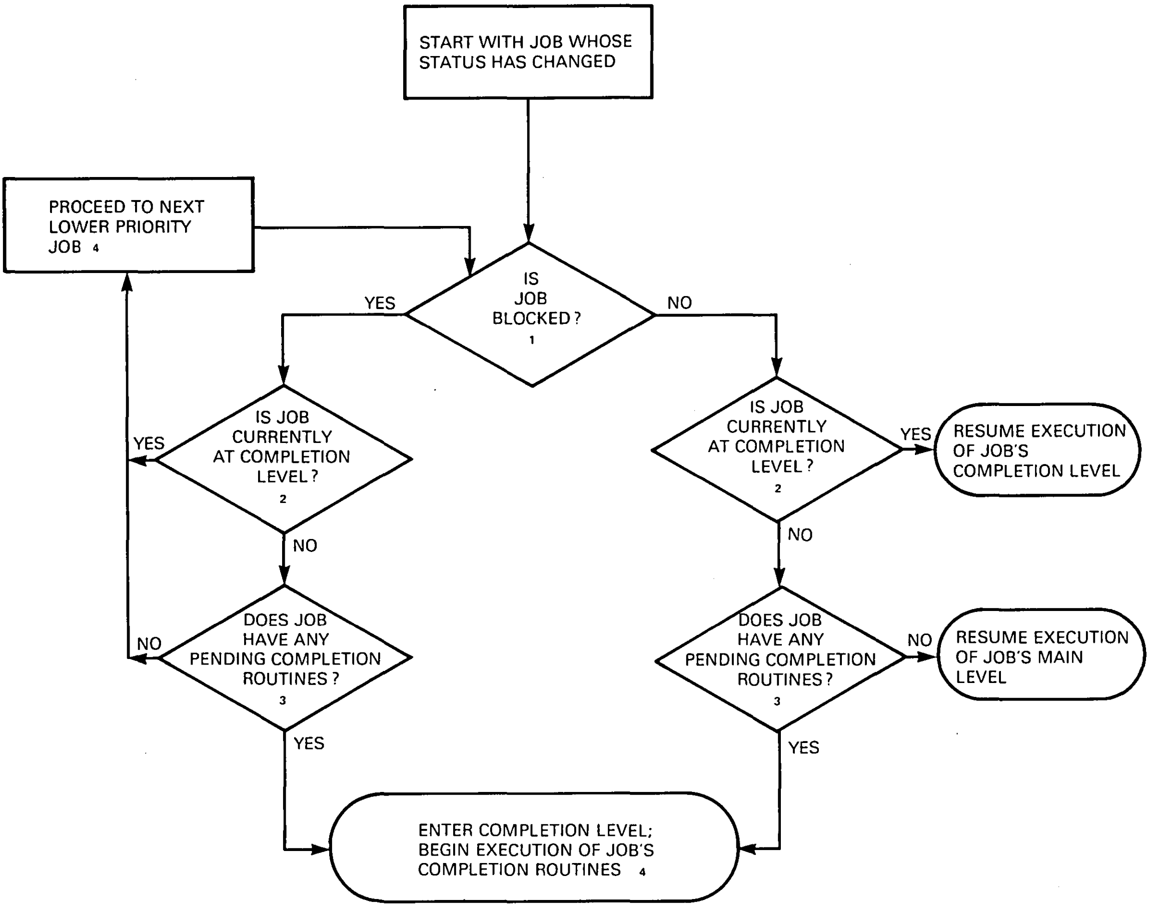
to check if the job is runnable. Later when the event being waited for occurs, the monitor clears the appropriate bit.

I.BLOK records the blocking of the job's main level, if there are no completion routines queued. When completion routines are queued, the main-level blocking bits are saved, and I.BLOK records the blocking conditions of the completion level. After the last completion routine in a job's completion queue has run, RMON restores the blocking conditions of the main level in the I.BLOK word. This means that a job's completion level can run even if the main level is blocked. However, if a completion routine becomes blocked, both levels of the job are blocked; that is, the main level does not run.

Routines within RMON recognize events that change the blocking state of a job and the level at which a job operates (main or completion). When they recognize one of these changes, they request RMON to run a pass of the scheduling routine. RMON examines jobs in order of decreasing priority, starting with the job whose status has changed. Figure 57 shows how this scheduling pass runs.

The result is that RMON runs the highest priority job runnable. It also runs a job's completion routines before it runs the main level. The numbered notes (opposite page) apply to figure 57.

Figure 57.
A Scheduling Pass


START WITH JOB WHOSE STATUS HAS CHANGED
PROCEED TO NEXT LOWER PRIORITY JOB 4
IS JOB BLOCKED? 1
YES
NO
IS JOB CURRENTLY AT COMPLETION LEVEL? 2
YES
NO
IS JOB CURRENTLY AT COMPLETION LEVEL? 2
YES
RESUME EXECUTION OF JOB'S COMPLETION LEVEL
NO
DOES JOB HAVE ANY PENDING COMPLETION ROUTINES? 3
NO
YES
DOES JOB HAVE ANY PENDING COMPLETION ROUTINES? 3
NO
RESUME EXECUTION OF JOB'S MAIN LEVEL
YES
ENTER COMPLETION LEVEL; BEGIN EXECUTION OF JOB'S COMPLETION ROUTINES 4

1 The I.BLOK word is tested.

2 When RMON switches the job to completion level (see note 4), it records this by setting the CMPLT$ bit in the job state word I.STATE in the job's impure area.

3 A pending completion routine is one that has been placed into the completion queue but has not started executing. This condition is also recorded in the I.STATE word.

4 Once a job enters completion level, it stays there until there are no more completion routines pending. If a job is blocked while at completion level, the main level is not run. After the last completion routine runs, RMON clears the CMPLT$ bit in the I.STATE word and a new scheduling pass is started.

Context Switching

Context switching occurs when the scheduler, as a result of a scheduling pass, starts to execute a job other than the one that was running before the pass. Basically, the operation saves some information from the outgoing job and restores some information about the incoming job.

Context switching can occur after any scheduling pass but not necessarily after every scheduling pass. Only scheduling passes that change jobs result in a context switch. Switching between a job's main level and its completion level does not change jobs and, therefore, does not cause a context switch. In general, a context switch occurs when a high priority job (the foreground job or a system job) becomes blocked, allowing a lower priority job to run, or when a high priority job becomes unblocked, and therefore, interrupts a lower priority job.

Certain memory areas and registers contain a job's data while it is being executed. When a different job starts executing, these areas must contain the new job's data. When a context switch occurs, the old contents of these locations are saved on the stack and in the impure area of the outgoing job, and the incoming job's data for these areas is loaded from the incoming job's stack and impure area. This information includes:

- All the general purpose registers (the stack pointer itself is always stored in the job's impure area)
- The vector for the TRAP instruction
- The system communication area (words 40 to 52)
- If floating point hardware is present, certain registers of the FPP
- In a multiterminal monitor, word 56 (fill count and character)
- In an extended memory monitor, the BPT and IOT vectors, kernel PAR1, and the memory-management fault trap vector

In addition, a MACRO–11 programmer can use the .CNTXSW request to specify other locations to be saved across a context switch operation. This is normally used for jobs that share trap or interrupt vectors.

Using FB Scheduling Efficiently

The goal of foreground/background operation is to make better use of the CPU, which under some conditions is idle for much of its time. Efficient FB scheduling depends on the coding of your application programs. If you want to use both foreground and background jobs in programming your application, the following points will help you determine which functions to put into the foreground and which to put into the background:

- Your system performance will be best if the highest priority job (foreground) is blocked most of the time, allowing lower priority jobs to run; and if the lowest priority job (background) is using the CPU most of the time, thereby taking up the CPU time left over by the foreground and any system jobs. It is common practice to do real-time I/O in the foreground, and data analysis in the background.

- In a real-time application, some operations are time critical. That is, the program must provide very fast response to changes in external conditions. Time-critical operations should be placed in the foreground so that no higher priority job can take the CPU from the program performing a time-critical operation. Operations not time-critical are best performed in the background.

- You can run a foreground job and switch from one background job to another, however, you cannot change foreground jobs without interrupting the background job. This is because you run KMON in background when you issue the command to run a foreground job. The foreground is the best place for continuous operations such as monitoring, data collecting, or process control.

- Short or infrequent operations (such as initialization, queries, changes, parameters, or report generation) can be put into programs that run as background jobs when they are needed. This saves space in the foreground job.

Examples of this type of organization include the queuing and error-logging systems in RT–11. Each has a single foreground/system job (QUEUE and ERRLOG), which runs continuously, and one or more background jobs (QUEMAN in the queuing system, and ELINIT and ERROUT in error logging), which are run only when their operations are needed. The background jobs run infrequently in comparison to foreground or system jobs.

You should also consider whether you need to use both a foreground and background job. Context switching takes time. How much time depends on the processor and monitor you are using, and whether your jobs have issued .CNTXSW requests. It is usually a few tenths of a millisecond. Normally your application runs faster if you combine all functions into one job than if you use two jobs, because there will not be any context switching. There are, however, other reasons—such as efficient use of the CPU and memory—for separating your application into two jobs.

The execution of a foreground job includes periods when the job is blocked and periods when the job is runnable. Each change between blocked and runnable may need a context switch. A job with fewer larger periods uses the system more efficiently than a job with many smaller periods because the former needs fewer context switches. Therefore, when writing a foreground job, try to organize it to minimize the number of context switches.

A common mistake in the use of completion routines is trying to increase the efficiency of a job by performing operations at completion level, and therefore, at a higher priority than at main level. This does not always help your program and can even slow it down. The FB monitor runs completion routines FIFO, so if one completion routine takes a long time to run, it delays all the routines following it. Keep completion routines as short as possible, doing any long processing at the main level. Also, make sure that very long processing is done in the background job.

Waiting for Specific Events

The synchronous I/O requests, the WAIT request for an I/O channel, and the timed wait request (TWAIT) each block a job until a specific condition has been met: completion of I/O on a channel, or expiration of a time limit. Sometimes you want your job to be blocked until some more complex condition is met. For example, you may want to:

- Issue a number of I/O requests on a channel, and then block your program until a specific request (other than the last one) completes.
- Issue a number of I/O requests on different channels, and then block your program until any one (or a specific combination) of those requests completes.
- Issue an I/O request, and then block your job until either that request completes or a watchdog mark-time routine runs. This is discussed in chapter 20, "Writing Time-dependent Programs."

An example of these techniques is the QUEUE program. It suspends the main level, waiting for a message from QUEMAN. At the same time it has a watchdog routine to check for ⟨CTRL/C⟩⟨CTRL/C⟩ being entered.

Each of these blocking situations can be implemented using completion routines, event-driven I/O requests, and mark time requests. To wait for a specified condition, your main-level routine can check a flag or some set of flags that are to be set by the completion routines, and then loop until the flags have the values needed.

This procedure is, in fact, the best way to implement a waiting condition under the SJ monitor. It is not, however, the best way under the Foreground/Background monitor, because it does not block the job. A foreground job that loops like this uses valuable CPU time, during which a background (or system) job might be able to run.

To allow a job to be blocked until some condition defined by the programmer is met, the Foreground/Background monitor provides a suspend and resume request. To suspend a job, the MACRO–11 request is:

```
.SPND
```

To suspend a job, the FORTRAN IV request is:

```
CALL SUSPND
```

To resume a job, the MACRO–11 request is:

```
.RSUM
```

To resume a job, the FORTRAN IV request is:

```
CALL RESUME
```

A suspend request is usually issued from the main level of the job. It blocks the job at the main level, but allows completion routines to run. A suspend request issued from completion level has a different effect. The resume request is almost always issued from a completion routine. It unblocks the main level which starts to execute again as soon as all processing at completion level has been done. Suspend and resume requests have the following effects:

- A suspend counter is kept for each job in its impure area. The initial value of this counter is 0.
- Each suspend request decreases the counter. Each resume request increases the counter.
- If the counter is negative, the main level of the job is blocked.

The one exception is when a suspend request issued from a completion routine causes the counter to become negative. The main level is not immediately blocked; it is blocked the next time a suspend request is issued from main level. At that time, the counter is equal to −2 and two resume requests are needed in order to unblock the main level.

The use of a counter ensures that a program is not suspended if the completion routine issues the resume request before the main level is able to issue its suspend request. The suspend request returns immediately to the main level because the blocking condition has been cleared in advance.

Use suspend and resume requests with caution if your program also uses TWAIT requests. TWAIT is implemented by RMON using an internal mark time request, a suspend request, and a resume request from within RMON's mark time completion routine. Because of this, if your completion routines issue one resume request too many, this can cause a TWAIT request in the main level to finish too soon.

References

RT–11 Programmer's Reference Manual. Chapter 2 describes the .CNTXSW request in detail.

RT–11 Software Support Manual. Chapter 3 covers RT–11 scheduling in detail.

17

17

Transferring Data Between Jobs

A foreground job and a background job running at the same time may be unrelated. For example, an application program runs in the foreground while program development takes place in the background. In this case, the Foreground/Background monitor is being used as a tool to allow the processor to do more than one thing at a time. In another instance, the foreground and background jobs may have a common application and the two jobs must be able to communicate with each other to coordinate their operations and swap information.

This chapter discusses the programmed requests that allow your foreground and background jobs to communicate, either directly through memory or through files on a mass storage device. The MACRO–11 programmed requests discussed in this chapter are: .SDAT, .SDATW, .SDATC, .MWAIT, .RCVD, .RCVDW, .RCVDC, and .CHCOPY. The FORTRAN IV requests discussed are: ISDAT, ISDATW, ISDATC, ISDATF, MWAIT, IRCVD, IRCVDW, IRCVDC, IRCVDF, and ICHCPY.

When you have completed this chapter, you will be able to use the send and receive data requests to copy information from one job to another; write a pair of foreground and background jobs that communicate through a shared buffer area; synchronize a foreground and background job using synchronous send and receive data requests; and transfer files between foreground and background jobs that are running at the same time.

Communication Through Send and Receive Requests

RT–11 supports three job communication methods: send and receive requests, shared buffers, and shared files. Foreground and background jobs can communicate directly through memory using send data and receive data requests, which are similar to write and read requests to peripheral devices under the queued I/O system. The similarities are shown in table 16. The send and receive data requests are shown in table 17.

The word count argument (wcnt) for all the requests shown in table 17 must be positive. All other arguments are identical to those for the corresponding queued I/O requests. The MWAIT request blocks the job that issues it until all send and receive requests that the job issued are completed. It is possible for one job's MWAIT request to return

Table 16.
Similarities between Queued I/O and Communication Requests

Queued Requests	Communication Requests
Requests to write to a channel	Requests to send data to other job
Requests to read from a channel	Requests to receive data from other job
Three I/O modes	Same three modes
WAIT request to wait for I/O on a channel to complete	MWAIT request to wait until message requests complete

successfully if the other job has communication requests left. MWAIT waits only for completion of the communication requests of the job that issued the MWAIT.

Table 17.
Send and Receive Data Requests

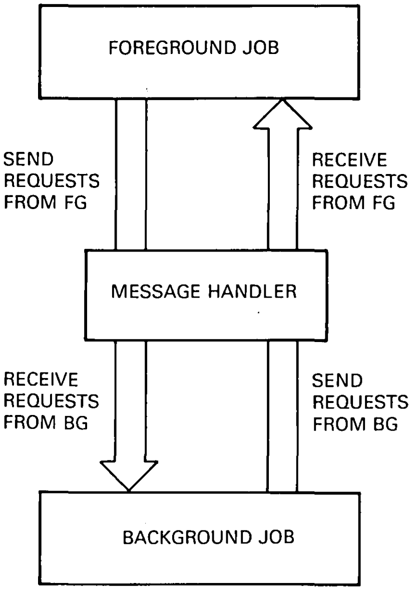
	MACRO-11 Requests	FORTRAN IV Requests
Synchronous	.SDATW area,buff,wcnt	IERR= ISDATW (buff,wcnt)
	.RCVDW area,buff,wcnt	IERR= IRCVDW (buff, wcnt)
Asynchronous	.SDAT area,buff,wcnt	IERR= ISDAT (buff,wcnt)
	.RCVD area,buff,wcnt	IERR= IRCVD (buff,wcnt)
	.MWAIT	CALL MWAIT
Event-driven	.SDATC area,buff,wcnt,crtn	IERR= ISDATC (buff,wcnt,crtn)
	.RCVDC area,buff,wcnt,crtn	IERR= IRCVDC (buff,wcnt,crtn)
		IERR= ISDATF (buff,wcnt,area,crtn)
		IERR= IRCVDF (buff,wcnt,area,crtn)

System Message Handler

Communication requests are implemented by RMON using a pseudo-device handler called the system message handler. The system message handler is responsible for executing send and receive data requests, which the handler receives as if they were queued I/O requests. On FB and XM monitors, you can reference the system message handler as the MQ: device, open channels through it, and access it using queued I/O read and write requests. Communication requests are controlled in the same way as queued I/O requests, so every communication request uses an I/O queue element from the job issuing the request. Remember this when issuing QSET requests at the start of your programs.

The system message handler matches a send queue element from one job with a receive queue element from another job. Once such a match is made, information is transferred from the sending to the receiving job, and both programmed requests (the send and the receive) are reported as having completed. This procedure sets up two

Figure 58.
Flow of Information through the System Message Handler

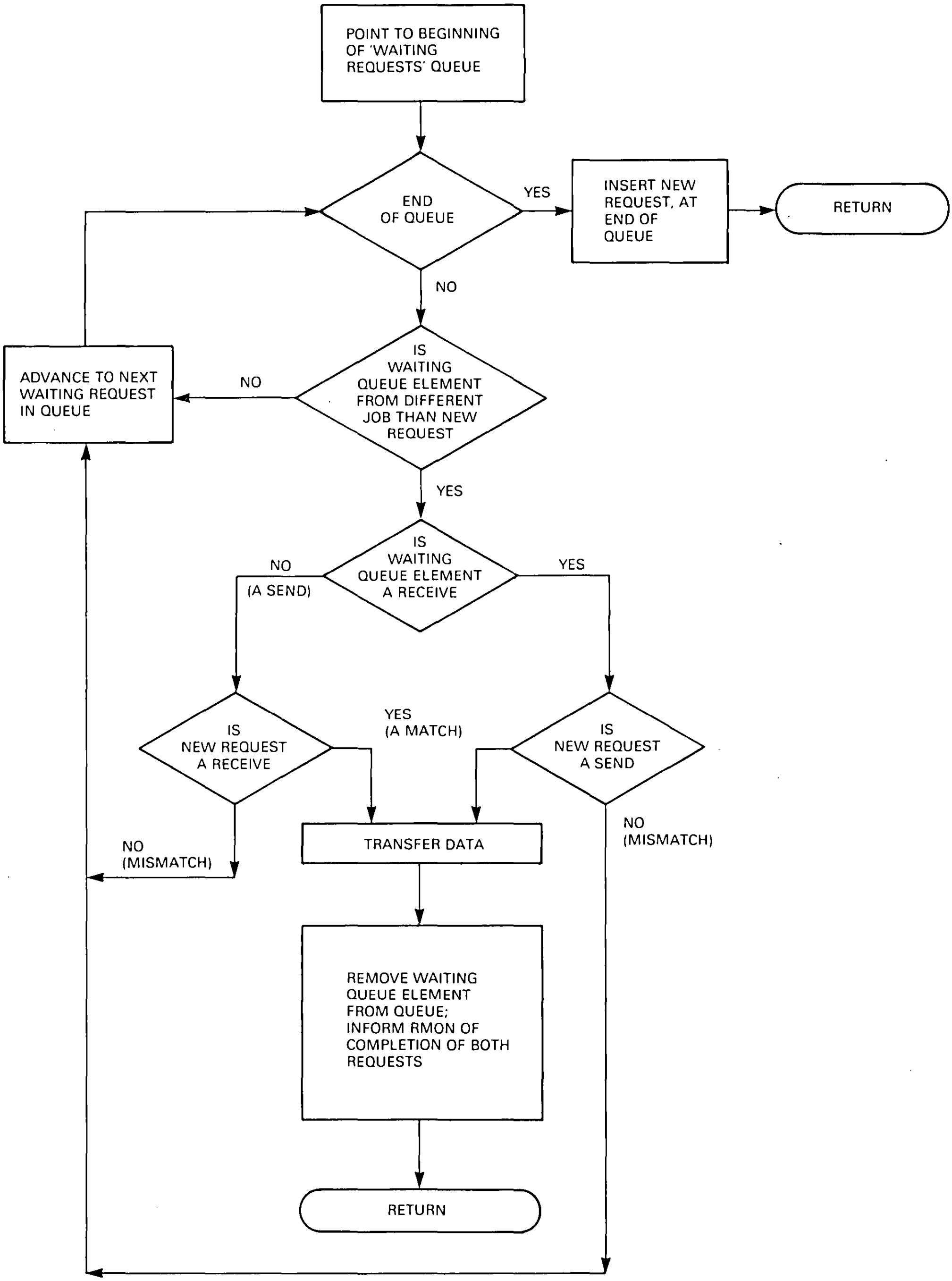


independent paths, similar to channels in the queued I/O system. One path handles information sent from foreground to background, and one goes from background to foreground. This flow of information is represented in figure 58.

The system message handler keeps track of communication requests by keeping a queue of waiting requests. This queue is made up of the I/O queue elements that RMON passes to the system message handler. The flowchart in figure 59 shows the action of the system message handler when it receives a new communication request. The following points apply to send and receive requests:

- Each new send request must be matched by a receive request from the other job (and each new receive request must be matched by a send request) before it can complete.

Figure 59.
System Message Handler Processing a New Request


POINT TO BEGINNING OF 'WAITING REQUESTS' QUEUE
END OF QUEUE
YES
INSERT NEW REQUEST, AT END OF QUEUE
RETURN
NO
IS WAITING QUEUE ELEMENT FROM DIFFERENT JOB THAN NEW REQUEST
NO
ADVANCE TO NEXT WAITING REQUEST IN QUEUE
YES
IS WAITING QUEUE ELEMENT A RECEIVE
NO (A SEND)
YES
IS NEW REQUEST A RECEIVE
YES (A MATCH)
IS NEW REQUEST A SEND
NO (MISMATCH)
TRANSFER DATA
NO (MISMATCH)
REMOVE WAITING QUEUE ELEMENT FROM QUEUE; INFORM RMON OF COMPLETION OF BOTH REQUESTS
RETURN

- If a request is received and no matching request is in the current queue, the queue element for the new request is placed in the queue to wait for a request that matches it.
- The queue is a first in—first out (FIFO) one. A search for a match for a new request always starts at the head of the queue. If it is necessary to add a new queue element, it is added at the end of the queue.
- If one job issues both a send and a receive, these requests may not be completed in the same order in which they were issued. The order of completion depends on the order in which the other job issues its requests. This is different from standard queued I/O requests. Queued I/O requests to a device handler are always completed in the order they were issued.

Because of the way that send and receive requests are matched by the system message handler, the I/O mode of the send need not match that of the receive. Thus, you can use an event-driven send from the foreground and a synchronous receive from the background.

Send and Receive Buffers

Each send request and each receive request specify a buffer within the job that issued the request. Data is copied from the sending job's buffer (send buffer) into the receiving job's buffer (receive buffer).

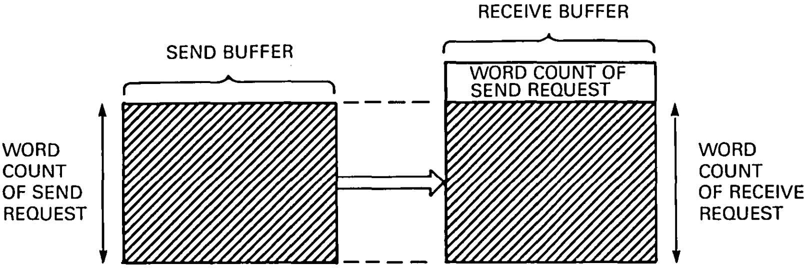
Each communication request also specifies a word count, which is the number of words it expects to be copied. In the send request, the word count should equal the number of words in the send buffer. The word count of the receive request should match that of the sending job. They may not match if there is an error. The receiving job is capable of checking for this type of error, for the message handler always passes the sending job's word count to the receiving job, in the first word of the receive buffer.

The first word of the receive buffer is used for the copy of the sending job's word count, so you should always make your receive buffer one word larger than the receive word count. This is very important. Otherwise the data in the word immediately after the receive buffer is destroyed when the information is copied from the sending job.

The following figures show how the results of a send request and a receive request differ according to the relative sizes of the send and receive word counts. Figure 60 shows the result if the send and receive word counts are equal. Figure 61 shows the result if the send count is larger than the receive count. Figure 62 shows the result if the receive word count is larger. In each figure, the diagonal shading represents the words copied from the send buffer to the receive buffer.

Remember that it is up to the receiving job to compare the requested receive word count with the send word count (found in the first word of the receive buffer) and take appropriate action if they are different. The action to be taken depends on your application.

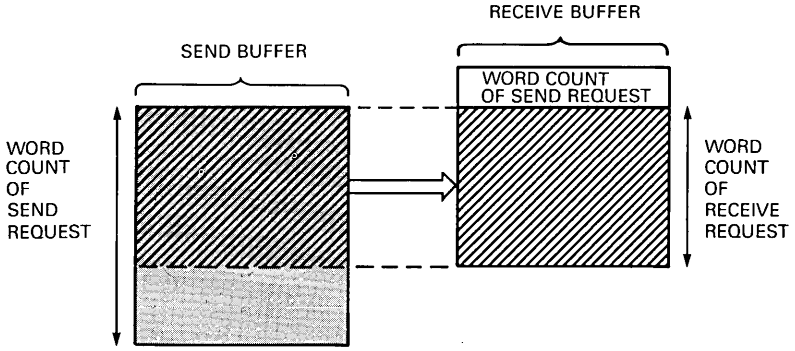
Figure 60.
Send Word Count Equal to Receive Word Count


RECEIVE BUFFER
SEND BUFFER
WORD COUNT OF SEND REQUEST
WORD COUNT OF SEND REQUEST
WORD COUNT OF RECEIVE REQUEST

RESULTS:

- THE ENTIRE SEND BUFFER IS COPIED
- THE ENTIRE RECEIVE BUFFER IS MEANINGFUL

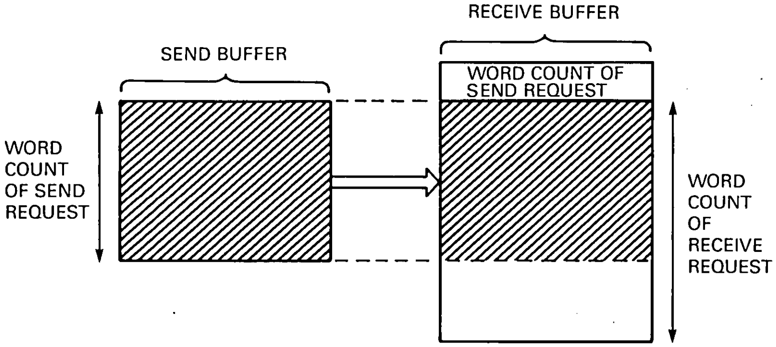
Figure 61.
Send Word Count Greater Than Receive Word Count


RECEIVE BUFFER
SEND BUFFER
WORD COUNT OF SEND REQUEST
WORD COUNT OF SEND REQUEST
WORD COUNT OF RECEIVE REQUEST

RESULTS:

- ONLY PART OF THE SEND BUFFER IS COPIED
- NOTHING OUTSIDE OF THE RECEIVE BUFFER IS DESTROYED

Figure 62.
Send Word Count Less Than Receive Word Count


RECEIVE BUFFER
SEND BUFFER
WORD COUNT OF SEND REQUEST
WORD COUNT OF SEND REQUEST
WORD COUNT OF RECEIVE REQUEST

RESULTS:

- ALL OF THE SEND BUFFER IS COPIED, BUT
- PART OF THE RECEIVE BUFFER IS MEANINGLESS

Communication Through Shared Buffers

When the amount of information to be transmitted between jobs is more than a few words, it may be better to share a common data buffer than to copy all the data from one job to the other.

Copying the buffer with send and receive requests is the more direct of the two options and, for small amounts of information, is usually the better choice. Using a shared buffer requires more careful planning and complex programming, but there can be significant benefits to using a shared buffer. To set up a shared buffer, you use a procedure like the following:

1. One job must reserve space for the buffer that is to be shared. You can reserve space at compilation or assembly time by defining a buffer or array. There are also methods to reserve more buffer space at run time (discussed in chapter 18, "Using Memory").
2. The job that reserved the shared buffer uses a send request to send a message to the other job. The message contains the address of the buffer and may also contain its length.

   A FORTRAN IV program that uses a variable or an array for the buffer can find out the address of that storage location by using the SYSLIB routine IADDR.
3. The other job issues a receive request for the message in order to find out the address (and length) of the buffer that is to be shared.

MACRO–11 programs can directly access the shared buffer. FORTRAN IV programs have to use routines like IPEEK and IPOKE in order to use the buffer because the receiving routine does not have a variable name for the buffer itself, only for a location that contains the address of the buffer.

Another way of gaining access to the buffer is to use the MACRO–11 function INDIR. INDIR acts as an intermediary between two FORTRAN IV routines. It allows the calling routine to pass arguments in an indirect manner. Instead of giving the name of the variable to be passed, the routine's CALL statement may give the name of a variable that contains the address of the value to be passed. Many levels of indirection are allowed. The form of the statement is:

Call Indir (Proc, Mode1, Arg1, Mode2, Arg2, . . .)
or
X=Indir (Proc, Mode1, Arg1, Mode2, Arg2, . . .)

"PROC" must be declared as external in the routine calling INDIR. For each argument, "MODE" is the number of levels of indirection. MODE=0 is equivalent to a direct call. The program (INDIR.MAC), which follows, shows a MACRO–11 function that passes arguments to FORTRAN IV routines.

FORTRAN IV programs can use INDIR to interpret addresses when calling subroutines. For instance, if the address of an array is in the INTEGER*2 variable IPTER, you can call a subroutine, and pass the array as an argument.

EXAMPLE

```
EXTERNAL SUBA
      .
      .
CALL INDIR(SUBA,1,IPTER,0,INUM)
      .
      .
END
SUBROUTINE SUBA(ARRAY,IVAL)
INTEGER*4 ARRAY(10)
      .
      .
RETURN
```

INDIR.MAC

```
        .TITLE  INDIRECT FORTRAN CALLS
;
; NOTE: This routine may be called by the PROC routine or
;       any of its routines.
;
INDIR:: MOV     (R5)+,R0           ;Load number of arguments
        MOV     R0,R1              ;Copy number of arguments
        DEC     R1                 ;Forget about PROC as arg
        ASR     R1                 ;/2 to remove MODEs
        ASL     R0                 ;Advance to end of
        ADD     R5,R0              ; argument list
        MOV     (R5)+,R4           ;Load PROC routine addr
        MOV     SP,SAVESP          ;Save stack pointer
1$:     MOV     -(R0),R2           ;Load ARGN address
        MOV     @-(R0),R3          ;Load MODEN value
        BEQ     3$                 ;Branch if MODE = 0
2$:     MOV     (R2),R2            ;Chase indirection
        DEC     R3                 ;Decrement mode
        BNE     2$                 ;Branch if MODE <> 0
3$:     MOV     R2,-(SP)           ;Push argument onto stack
        CMP     R0,R5              ;All arguments processed?
        BNE     1$                 ;Branch if not
        MOV     R1,-(SP)           ;Push number of argument
        MOV     SP,R5              ;Load argument block ptr
        MOV     SAVESP,-(SP)       ;Save saved stack ptr
        CALL    (R4)               ;Call the PROC routine
        MOV     (SP)+,SP           ;Restore original stack
        RETURN                     ;Return to caller
;
; NOTE: Value returned by INDIR is the value returned by
;       the PROC routine since no registers are modified
;       upon return from PROC.
;
SAVESP: .BLKW   1                  ;Saved stack pointer
        .END
```

Benefits of Using Shared Buffers

When you want to transmit information between jobs, the benefits of using a shared buffer, instead of send and receive requests include the following:

- One copy of the information takes up less memory then two copies.

- Execution can be faster because the data does not have to be copied.
- Each job has continuously updated information from the other job. The use of a shared buffer, which does not need repeated sends or receives, can increase program speed and accuracy.

Disadvantages of Using Shared Buffers

The disadvantage of using shared buffers to communicate between jobs is that programs are harder to write and debug. For example:

- The two jobs must be synchronized so that they do not access the buffer at the same time.
- The job that receives the buffer address might modify a location outside the buffer and destroy the first job.

Restrictions on the Use of Shared Buffers

There are two restrictions when using shared buffers:

- The second job may not use the shared buffer for I/O, in a CDFN or QSET request, or other operation limited to addresses within the job's own memory space.
- Under the XM monitor, a shared buffer must be in memory (below 28 Kwords) and both jobs must be privileged.

Synchronizing Buffer Access

In a coordinated foreground/background system, one job may need to know when the other job has started or completed some operation. This calls for communication between the two jobs, in the form of:

- Queries, where job A says to job B: "Send me a message when you have done operation X"
- Status reports, where job B says to job A: "I have completed operation X"

If job A needs the results of the action taken by job B, it should block itself by issuing a synchronous communication request (either send or receive). This blocks job A and also sets up the condition for removing the block. The effect is that job A says to job B: "Wake me up when you are ready." Job B then must issue the appropriate request (receive or send) when it has completed the specified task.

The content of the buffers used in these send and receive requests may not be important. The execution of the requests may be enough to indicate: "Wake me up when you are ready," or "Wake up! I am ready." In this example the send buffer can be a single word containing a zero.

The contents of the send buffer can be used to transmit additional information, such as whether job B completed its task successfully, or where the results are located in the buffer.

If the job that needs the results can do other work while waiting, use a nonsynchronous communication request. Then at a later time this job can use MWAIT to wait for the request to complete. It can also check flags set by a completion routine or suspend itself until resumed by a completion routine.

If you want a foreground job to receive a message from the background job, you must make sure that the foreground job becomes blocked, to allow the background job time to run. Do this either by using synchronous requests, or asynchronous requests and MWAIT, or by suspending a job until it is resumed by a completion routine.

Communication Through Files

Communication using a file is slower than using send and receive requests or shared buffers because of the device access time. However, two benefits of file communication are:

more information can be stored in a file than in memory and a file leaves a permanent record of the information for future use.

If two jobs each have a channel open to the same file, both can do I/O to that file. Both jobs' requests are queued to the device handler, as follows:

- A device handler services only one I/O request at a time. Once a handler starts to execute an I/O request, it completes that request before it starts on any new request. For example, if a handler is servicing an I/O request from the background job and a new request comes in from the foreground job, the handler completes the background job's request before starting to work on the foreground job's request.
- All requests for a handler, except for the one currently being executed, are queued in order of job priority. The requests for the foreground job are first, then those of the system jobs, and then the requests for the background job.
- Within each job's section the queue is FIFO.

Jobs must be synchronized if both modify a shared file. Consider an example where the background job reads a block from a file, changes a few words in that block, and then rewrites the block. After the background job reads that block and before it writes the block back, the foreground could run and make a change in the block. If the background did not know about the foreground's change, it would write its block out, writing over the changes made by the foreground. You prevent this condition in shared files in the same way you prevent it in shared buffers—by setting up a system of shared flags or send and receive requests to synchronize the jobs' execution.

Sharing a file is simple if the file was created before both jobs are run. Each job needs only to issue a LOOKUP request to the file. If two jobs create a file to share, then one job must create the file using an ENTER request and tell the second job the channel number it is using. The second job has to issue a channel copy request. This request can also be used if the first job has gained access to the file with an

ENTER request. All that is needed to copy a channel to the second job is for the first job to have that channel open. For MACRO–11 the channel copy request is:

```
.CHCOPY   area,chan,ochan
```

For FORTRAN IV the request is:

```
IERR=ICHCPY (chan,ochan)
```

In these requests, "chan" is the channel number to be used by the job copying the channel, and "ochan" is the channel number being used by the job that first opened the file.

The effect of this request is to copy one job's channel table for this file into the channel table of the other job. When this request has been executed, both the copying job and the job that first opened the file can access the file. There are only two significant differences between the privileges of the two jobs:

1. In the copying job's channel status word, the write-to-directory bit (bit 7) is always clear. So, a CLOSE request from the copying job does not make this file permanent if it was opened with an ENTER. Only the job that created the file can make it permanent.
2. If the file is opened with an ENTER request, the copying job can reference it only up to the highest block that had been written to by the entering job at the time that the channel copy was performed. The copying job cannot read from or write to any blocks past that limit.

   One way around this problem is for the entering job to write to the last block of the tentative file. This saves all blocks when the file is closed, and allows both jobs access to all blocks of the file.

Coordinated Foreground/Background Systems

The error-logging system and the queuing system are coordinated foreground/background systems. Each uses both communication requests and channel copying to communicate. Short descriptions of the operation of these systems are given here as examples of how these requests can be used.

Error Logging

The error-logging system has two background jobs (ELINIT and ERROUT), a foreground job (ERRLOG), and a data file (ERRLOG.DAT). ELINIT is responsible for finding (using LOOKUP) or creating (using ENTER) ERRLOG.DAT. ELINIT opens a channel to ERRLOG.DAT. If ELINIT creates a new file, it writes to the last block of the file, saving the complete length of the tentative file. It closes the tentative file to make sure that it is made permanent, then reopens the file with LOOKUP. It sends ERRLOG some control information about the file and the system configuration.

One of the first things that ERRLOG does is issue a .RCVDW, which blocks ERRLOG until ELINIT issues its send request containing the control information. ERRLOG copies ELINIT's channel to ERRLOG.DAT, issues a .RCVDC for a message from ERROUT, and .SPND to suspend itself. ERRLOG can become unblocked either by the completion routine from the .RCVDC, or from a special routine called from device handlers to report error-logging data. When ERRLOG becomes unblocked, it writes data to ERRLOG.DAT using the channel that it copied from ELINIT, and then suspends itself again. ERROUT executes a send data request to ERRLOG to ask it to write out the last of its data to ERRLOG.DAT. Then ERROUT opens its own channel to the file and creates a report based on the data in the file.

Queuing System

In the queuing system, a foreground job QUEUE opens a data file SY:QUFILE.WRK. This file is opened using LOOKUP or ENTER and is used to keep track of file transfer requests for QUEUE. When QUEUE completes one requested transfer, it deletes the entry for that transfer in the list in SY:QUFILE.WRK, and starts up the next operation. New requests are added to the queue as a result of messages sent from the background job QUEMAN. QUEUE's main level has three major functions. It processes:

- Requests from QUEMAN.
- The completion of one transfer and the starting of another.
- Clean-up operations if QUEUE is aborted. QUEUE inhibits normal ⟨CTRL/C⟩ action and sets up a mark time completion routine to run at regular intervals, which checks for double ⟨CTRL/C⟩.

The main level of QUEUE uses a .RCVDC request to check for messages from QUEMAN, .READCs and .WRITCs to do I/O, and .MRKT to set up the completion routine to look for ⟨CTRL/C⟩. Then the main level suspends itself, leaving the completion routines to issue the resume request. When the main level becomes unblocked, it checks to see which completion routine unblocked it, performs the necessary task, reissues any needed communication or I/O request, and blocks itself again.

QUEMAN's main job is to take requests from the user, translate them into messages, and send them to QUEUE. One request that QUEMAN handles on its own, however, is a request to look at the status of current requests in the queue. To do this, QUEMAN copies QUEUE's channel to SY:QUFILE.WRK and reads from the file directly.

Practice 17–1

This practice requires a Foreground/Background monitor. You can write the programs in MACRO–11 or FORTRAN IV. If you use FORTRAN IV, the MACRO–11 function INDIR is shown in the program (INDIR.MAC) presented earlier in this chapter.

1. Write a foreground job and a background job such that:

   a. The foreground accepts a string of characters from the terminal.

b. The foreground sends a buffer containing that string of characters, to the background.

c. The background reverses the order of the characters in the string, and then sends a buffer containing the reversed string to the foreground.

d. The foreground then prints out the reversed string, and returns to step a.

2. Write the above program using a shared buffer instead of copying the strings from one job to the other.

3. Modify the programs in step 1. so that:

   a. The foreground writes the original string to block 0 of a file that it creates using ENTER.

   b. The foreground sends to the background the channel number on which that file is open.

   c. The background copies the channel from the foreground, reads the string from block 0, reverses the string, and writes the reversed string to block 1 of the file.

   d. The foreground reads block 1 of the file and types out the reversed string.

   e. The program leaves the file permanent when the foreground program exits.

References

RT–11 Programmer's Reference Manual. Chapters 2 and 3 discuss the channel copy requests for MACRO–11 and FORTRAN IV users.

RT–11 Software Support Manual. Chapter 4 provides information on the operation of programs sharing buffers under the XM monitor.

18

18

Using Memory

The memory on your system is a limited resource. The amount of memory needed by a job cannot exceed the available memory on your system, or RMON will not be able to run the job. As your programs become complex, the need to make efficient use of memory increases.

An issue related to memory use is speed of program execution. Certain methods of increasing execution speed do so at the cost of increased program size; on the other hand, other methods decrease program size at the cost of slower execution speed. For example, overlaying a program can decrease its size but also slows execution. Once you learn how to modify and control memory use on the RT–11 system, you can select the proper balance of size versus execution speed for your particular application.

This chapter discusses two ways you can modify the way that RT–11 uses memory. The first, called dynamic allocation of memory, is to issue requests that allocate memory to your job while it is running. The second is to control the swapping of the USR during the execution of your program. You will see how each method affects both the size and execution speed of your program.

The programmed requests discussed in this chapter are: .SETTOP, IGETSP, .LIMIT, .LOCK, .UNLOCK, and .TLOCK. When you have completed this chapter, you will be able to write code to request dynamic allocation of memory to a program; control the swapping location of the USR; and increase the speed of consecutive USR operations and minimize blocking of either job.

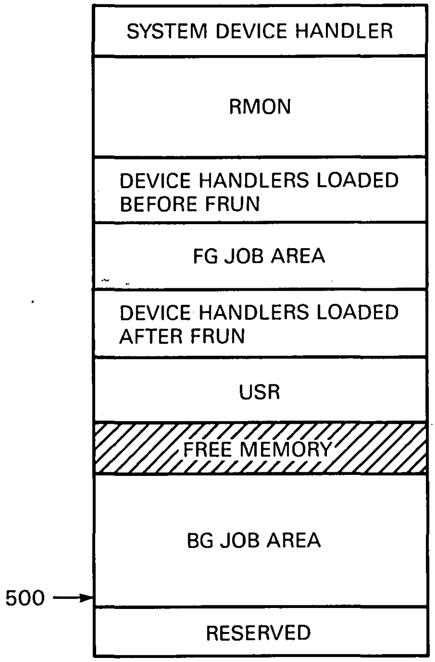
Standard Memory Use

Figure 63 shows a typical allocation of memory under the Foreground/Background monitor. Memory use under the SJ monitor is similar, but there is no foreground job; therefore, all loaded device handlers are in a contiguous area between RMON and the USR.

The background job area can be divided into the job itself—the code that you write—and the stack. The job starts at the base address at which it was linked, default 1000 (octal). By default, the stack starts at the base address and extends downward to location 500 (octal). (The stack position can be changed by the /STACK option of the LINK command.)

The foreground job area is made up of the foreground

Figure 63.
Allocation of Memory with Active Foreground and Background Jobs



job, its stack, and its impure area. As with a background job, the foreground stack is placed immediately below the job's base address by default. Its size is fixed when the job is linked. Default size is 128 (decimal), 200 (octal), bytes. The impure area is always placed at the bottom of the foreground area. (Remember that the background impure area is within RMON.) If you use the /BUFFER option of the FRUN command, space is reserved in the foreground area above the foreground job.

The absolute location of the foreground area in memory is determined by the amount of memory on your system, the size of the system device handler, RMON, and the other components (device handlers and system jobs) that are loaded into the area below RMON before the FRUN command is issued. Figure 64 shows the detail of the foreground and background areas.

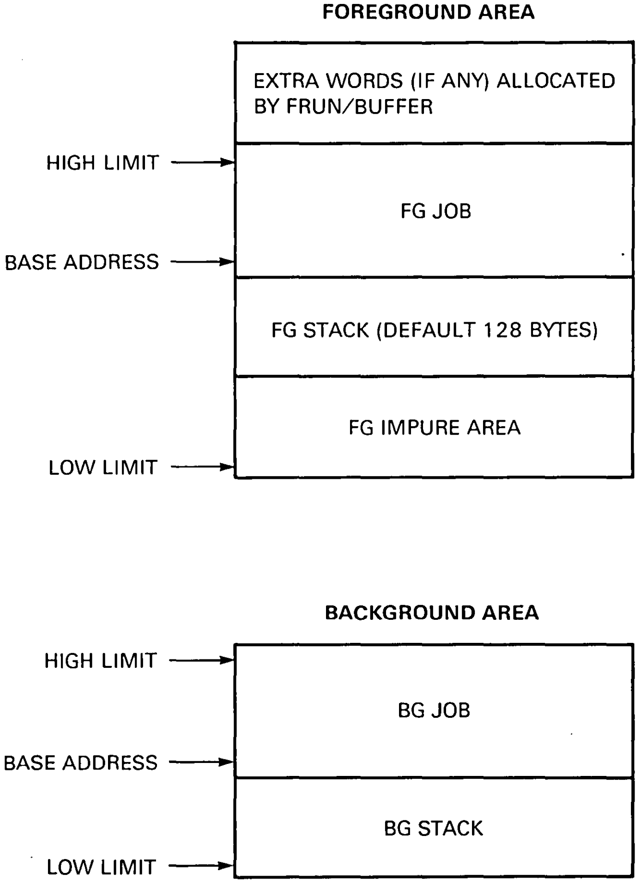
Each job has a high and a low limit. These limits are used by RMON to make sure that certain operations requested by a job do not affect any locations outside that job's area. For example, a program cannot perform I/O to a buffer outside its limits. A program can find out what its limits are by issuing a GTJB request.

The low limit of a job is the base of the area shown in figure 64. It does not change during execution. When a job is started, its high limit is the last location used in the load image (the last word used for instructions or for data storage requested within your program). For a background job, you can extend the high limit using the /TOP option of the LINK command; for a foreground job use /BUFFER of the FRUN command. The high limit can also be changed during execution. By raising its high limit, a job can request dynamic allocation of memory. This action also affects USR swapping.

Dynamic Allocation of Memory

Memory to be used for data storage can be allocated to your program in two ways. First, you can reserve storage within the load image itself; the corresponding locations are reserved in memory when the job is loaded into memory. You

Figure 64.
Detail of Foreground and Background Areas



usually reserve such storage locations using storage directives in MACRO–11 and variables in FORTRAN IV. (Some linker options also affect positioning of program components within the load image, and therefore, indirectly affect the way storage is reserved there.) Second, you may request that dynamic allocation of memory take place during your program's execution.

The performance of some programs is directly related to the amount of memory available for the programs. The more memory available, the better they will run. The improvement may extend the range of external conditions that they can handle or result in higher execution speed. For example, most programs that do I/O to a mass storage device will run faster given a larger I/O buffer, because it is almost always faster to read or write multiple blocks in one request than to do consecutive reads or writes of one block each. As another example, consider a program that accepts data from an input device in high-speed bursts. The more buffer space available to the program, the more data the program can accept before running out of room. This allows higher data rates.

The benefit of dynamic allocation of memory over reserving storage is that the amount of storage requested can be controlled by:

- The actual needs of the program, as determined during execution
- The actual amount of memory available at execution time

If you want to reserve space for these buffers as you write the program, you will have to determine how much buffer space you need. If a small buffer is reserved, the program may not perform as well as it could. If a larger buffer is reserved, the program may not fit into the amount of memory available at execution time. The problem is that, at the time you write a program, you do not know how much memory will be available when you execute it. The best solution, therefore, is not to leave space for such a buffer when you write your program, but rather, request dynamic allocation.

A MACRO–11 program can request dynamic allocation of memory by issuing a .SETTOP request. The effect of this request is to change the high limit of the program and gain the use of additional space.

The FORTRAN IV OTS (which is written in MACRO–11) automatically issues a .SETTOP request when you start up a FORTRAN IV program. The memory returned by this request is used as a workspace by the OTS for purposes such as temporary storage, I/O buffers, and space for device handlers.

The .SETTOP requested by the OTS gets as much memory as is available from RT–11, and all of the workspace is reserved for OTS use. Therefore, there is no request that you can use within your FORTRAN IV program to get more memory from RT–11. However, by using the IGETSP (get space) request, you can ask the OTS to allocate part of its workspace for your use. IGETSP is a request by which a FORTRAN IV program can request dynamic allocation of memory.

The .SETTOP Request

To determine the new high limit to request, use the .LIMIT directive. The following example shows a request for a buffer of 1000 (octal) bytes.

EXAMPLE

```
        MOV     LIMIT+2,R1 ;Get current high limit
        ADD     #1000,R1   ;Add 1000 bytes
        .SETTOP R1
        CMP     R0,R1      ;Did we get it?
        BNE     STPERR     ;No, go to err routine
        MOV     R0,LIMIT+2 ;Yes, update limit
                .
                .
                .
LIMIT:  .LIMIT
```

To allocate to your program as much memory as possible, request a high limit that you know cannot possibly be given to you.

EXAMPLE

```
.SETTOP #177776
```

RMON will change your request to the highest it can give you. Take the new high limit returned in R0.

The following example shows how to get as much memory as possible without forcing the USR to swap out because of .SETTOP. USR swapping is discussed in detail later in this chapter.

EXAMPLE

```
        .GVAL    #AREA,#266
        TST      -(R0)
        .SETTOP
          .
          .
          .
AREA:   .BLKW    2
```

.SETTOP under the Extended Memory Monitor

Using a feature known as a virtual .SETTOP, a MACRO–11 program executing under the XM monitor can request dynamic allocation of extended memory (that is, memory above 28 Kwords). This differs from a .SETTOP under the SJ or FB monitors and from a nonvirtual .SETTOP under the XM monitor, because each of those requests returns only memory under 28 Kwords regardless of how much memory is on the system. Two characteristics of the virtual .SETTOP make it very useful:

- On an XM system, you can obtain more memory by using a virtual .SETTOP to get memory above 28 Kwords than you can by using a nonvirtual .SETTOP.
- The virtual .SETTOP is an easier way to use extended memory than the extended memory programmed requests.

To request allocation of extended memory using a virtual .SETTOP, the job must be a virtual job. It is not necessary for you to understand the details of virtual and privileged mapping in order to use the virtual .SETTOP feature. To make your job virtual, you must set bit 10 of word 44 in the job's load image, the job status word. You can either patch the word or set it, using an absolute program section in your source code.

EXAMPLE

```
.ASECT
.=44
.WORD   2000   ;(BIT 10)
.PSECT
```

Then issue a .SETTOP. You can ask for a new high limit of any address between your job's next free address and 177776. The next free address is the next 4-Kword boundary above the addresses used by your program, including the root and any /O or /V overlays (if the program is overlaid). The next free address is always printed on your load map if you use the LINK/XM option or /V overlays. Your program can get its next free address at execution time if you use a .LIMIT assembler directive. This directive generates two words of storage, the second being the next free address if you use LINK/XM or /V overlays.

A virtual .SETTOP returns with two allocations:

- A new range of addresses within your program.
- A section of physical memory, taken from the memory above 28 Kwords that RMON controls. You can refer to this section of physical memory by using this new range of addresses as you would any other addresses within your program.

On return from .SETTOP, R0 contains the highest address that your job can now use. If RMON can give you all that you asked for, the new address range available to you is from the next free address to the new limit you requested in the .SETTOP.

If not enough memory is available above 28 Kwords, RMON decreases the size of your request, and the new high limit is less than you requested. If, on return from the .SETTOP, R0 contains the next free address, no new memory has been given to you.

Restrictions on the Use of .SETTOP

Certain restrictions apply when you use the .SETTOP programmed request for virtual jobs under the XM monitor.

1. You may not use addresses that lie between:

   a. Those available to you before the .SETTOP (the highest address used by the root and any /O or /V overlay segments)

   b. The next free address (which is the first 4 Kword boundary above a.)

   An attempt to use those addresses may result in an execution error that will abort your program.

2. The address you request in the .SETTOP should be above the next free address or else no new memory will be given to you.

3. The memory you get from .SETTOP cannot be used for queue elements or I/O channels.

4. A virtual job cannot access anything outside its own area. This includes RMON (other than by using .GVAL), the I/O page, and the interrupt vectors.

The IGETSP Routine

The IGETSP routine requests that the OTS allocate a part of its workspace for your program use. When the main-line routine of a job is written in FORTRAN IV, or when the OTS is initialized by a direct call to the OTS initialization routine, any request for dynamic allocation of memory should be made by a call to IGETSP. A FORTRAN IV call to IGETSP has the form:

```
ISIZE=IGETSP (min,max,iaddr)
```

In this request:

min — is the minimum acceptable size, should an area of the correct size not be available.

max — is the size (in words) of the area that you would like allocated to you.

iaddr — is an INTEGER variable in which IGETSP returns the address of the area allocated to you.

ISIZE — is an INTEGER variable that receives the actual size of the area allocated to you (min <= ISIZE <= max). If IGETSP cannot allocate an area of min words or larger, ISIZE receives a negative value.

You cannot use iaddr to refer directly to the buffer because iadder is not the actual buffer but an INTEGER variable that contains the buffer address. You can do one of the following:

- Use the SYSLIB routines IPEEK, IPEEKB, IPOKE, IPOKEB in order to reference this buffer from the routine that issued the IGETSP.
- Use an assembly language subroutine to reference it.
- Use a routine like INDIR (chapter 17, "Transferring Data Between Jobs") to allow the routine that issued the IGETSP to pass the buffer indirectly to a FORTRAN IV subroutine. The following example shows how you might request a buffer using IGETSP and then do I/O to that buffer using INDIR to pass the buffer to IREADW indirectly.

EXAMPLE

```
      ISIZE=IGETSP (256,512,IADDR)
      IF (ISIZE.LT.0) STOP 'NOT ENOUGH BUFFER SPACE'
      IERR=INDIR(IREADW,0,ISIZE,1,IADDR,0,IBLK,0,ICHAN)
C (DIRECT REFERENCES ON ALL ARGUMENTS EXCEPT IADDR.
C SEE THE LISTING OF INDIR IN CHAPTER 17)
      IF (IERR.LT.0) STOP 'READW ERROR'
```

You may call IGETSP and be given additional memory, but at a later time your program may fail because there is not enough workspace for the FORTRAN OTS. This is because the OTS cannot tell how much workspace it will need in the future. Therefore, it allocates memory based on its workspace needs at the time of the request.

If your program has this problem, you should lower the "max" argument in the IGETSP call until the program runs successfully. This is not a dependable solution as you have no assurance that the problem will not occur again if a different execution path through your program results in more use of the OTS workspace.

USR Control

The USR is designed to swap in and out, as needed, during program execution. Controlling where and when the USR swaps is another method to make efficient use of memory and achieve fast execution speed.

The Swapping Algorithm

The following conditions affect USR swapping.

- Under the XM monitor, the USR never swaps.
- If the USR is set to NOSWAP, it does not swap.

- If the USR is set to SWAP and the background high limit is above the base of the USR, the USR swaps. If the background high limit is below the base address of the USR, the USR does not swap.
- The background high limit is affected by a .SETTOP request. You can use .SETTOP to set the high limit above or below the USR base address to control whether or not the USR will swap.

The USR swaps out when a background program is run whose high limit is above the base address of the USR, or when a background program performs a .SETTOP above the base address of the USR. The USR swaps in during program execution when a job issues a request that needs code in the USR. Some of these requests are:

- Device handler operations: FETCH, DSTATUS
- I/O preparation: QSET, CDFN
- Directory operations: ENTER, LOOKUP, CLOSE, RENAME
- CSI operations (including GTLIN, which is implemented within the CSI)

Complete lists of the requests that use the USR are discussed in chapter 1 of the *RT–11 Programmer's Reference Manual.*

The address at which the USR is swapped is determined by the following conditions:

- If the contents of word 46 are 0, the USR swaps in (at its default location below RMON) any foreground or system jobs and any loaded handlers.
- If word 46 is not 0, the USR swaps at the address specified by the contents of word 46.
- The foreground job should always set an address into word 46. If the foreground job issues a USR request when word 46 is 0, and the USR is not resident, a fa-

tal error may result and the foreground job will be corrupted.

- By default, the FORTRAN IV OTS sets the USR to swap at the base of the program. Except in small FORTRAN IV programs, this is usually a good place for it to swap. If you have a small FORTRAN IV program, either set the USR to NOSWAP, or compile background jobs with the /NOSWAP option. The /NOSWAP option causes the OTS to do a .SETTOP to the base of the USR instead of up to the limit of the system.

When actual swapping occurs, the part of the job that is within the swap area of the USR is written out to SWAP.SYS. Then the USR is read into the swap area and executed. When the USR is done, the portion of the program that was written to SWAP.SYS is read back into the swap area.

Problems and Restrictions

If you allow the USR to swap, certain problems can result. Random errors can occur if, when swapping, the USR overwrites certain types of code or data that it may need while executing. See chapter 2 in the *RT–11 Software Support Manual* for a detailed list of the errors. In general, do not let the USR swap overwrite:

- The stack
- Any data that the USR itself needs—file definition blocks, for example
- Any code that may be entered asynchronously: interrupt service routines, device handlers, or completion routines

USR swapping takes time because of the disk accesses needed to write to and read from SWAP.SYS and to read the USR itself.

To solve problems caused by the USR overwriting certain code or data, you can either relink your modules so the USR swaps information that does not cause problems, or you can move the USR by setting an alternative swapping location into word 46.

Two areas which give you an opportunity to minimize delays caused by USR swapping are sequential USR operations and USR contention.

Sequential USR Operations

If one job executes several USR operations sequentially while the USR is swapping, each request needs three disk accesses for the swapping operation: write to swap file, read USR, and read swap file. Unnecessary disk accesses are made if the USR completes its operation and swaps out, then swaps right back in again.

To prevent excessive swapping, you can issue a LOCK request before the first USR operation and an UNLOCK request after the last to hold the USR in memory while sequential USR operations are performed. The USR swaps in if it is not resident when the LOCK is issued, and it stays in memory until the UNLOCK is issued. At that time normal swapping continues.

Be sure that neither the LOCK request, the UNLOCK request, nor any code or data needed between issuing the LOCK and issuing the UNLOCK, resides in the area where the USR swaps.

If your job meets the following three conditions, you can keep the USR resident at selected times:

- It is a background job.
- It has a high limit below the base address of the USR.
- It normally performs a .SETTOP to get as much buffer space as it can.

Issue .SETTOP requests below the base of the USR to allow the USR to stay resident, and above the base of the USR to force it to swap out.

Common practice is to issue requests such as .FETCH, .ENTER, .LOOKUP, and .QSET before issuing a .SETTOP (or, if you need some buffer space you can first issue a .SETTOP to the base of the USR). When these are completed, issue a .SETTUP above the base of the USR. When file use is completed and you want to close your channels, issue a .SETTOP to the base of the USR or to your original program high limit. This lowers the high limit and allows the USR to become resident once again.

USR Contention

The USR is not reentrant; it can be used by only one process at a time. Under the FB monitor, it is possible that while one job is using the USR, another job also needs to use the USR. This condition is known as USR contention. To prevent USR contention from causing execution errors, the monitor assigns ownership of the USR. As soon as the USR starts to execute a request for a job, or when a job issues a LOCK, that job is assigned USR ownership. Ownership is released when the USR completes the request or, if a LOCK has been issued, ownership is released when that job issues an UNLOCK. If, while one job owns the USR, a second job issues a request that needs the USR, RMON blocks the second job until the first job releases ownership. This blocking of the second job is called USR lockout.

With the exception of certain uses of communication requests discussed in chapter 17, "Transferring Data Between Jobs," USR lockout is the only state under the FB monitor in which a background job can force the foreground job to become blocked.

While USR lockout prevents execution errors caused by reentrance of the USR, it has the side effect of blocking the second job. If the first job has issued a LOCK request, the second job can be blocked out for a long time. This can be unacceptable if the second job is a real-time application that needs constant monitoring and control of external devices.

To prevent USR lockout from blocking a job, the job can issue a TLOCK request which:

- Tests if another job owns the USR
- Performs a LOCK if no other job owns the USR
- Returns an error if another job does own the USR

If a job that needs the USR issues a TLOCK request and receives an error in return, that job can either continue processing and try the TLOCK again or print an error message, close down the operation it is monitoring, and exit.

Practice 18–1

In this exercise you will modify one of the file-copying programs you created in practice 14–1 (either PR1801.MAC or PR1802.FOR). Create a new version of the program according to the instructions below:

1. Use a buffer that is allocated dynamically. Make the buffer as large as you can, given the available memory. If you are programming in FORTRAN IV, you can use the INDIR routine.
2. Keep the USR resident by doing a .SETTOP only to the base of the USR.
3. Maintain ownership of the USR during the operations that need the USR and during the close or purge operations.

The FORTRAN program will run only if you set the USR to NOSWAP or use the /NOSWAP option when you compile the module. If the USR is set to SWAP, the program will fail at the call to lock the USR in memory during the SETUP routine.

References

RT–11 Programmer's Reference Manual. Chapters 2 and 3 discuss the LOCK and UNLOCK request in MACRO–11 and FORTRAN IV programs. Chapters 1 and 2 contain material on the

.SETTOP request in an extended memory environment and explain how the .SETTOP issued by the OTS works. Chapters 1 and 3 provide detailed information about IGETSP.

The RT–11 Software Support Manual. Chapter 2 discusses the USR. Chapter 4 describes virtual jobs with .SETTOP in extended memory.

19

19

Using the Command String Interpreter

Some operations related to file I/O are needed by many programs. These operations include parsing command strings, fetching device handlers, opening channels, and connecting files to channels. The Command String Interpreter (CSI) performs these operations and is accessible from both MACRO–11 and FORTRAN IV programs.

This chapter focuses on the capabilities and use of the CSI. Using a programmed request to accept and parse a command string, fetch handlers, and open channels, you will write a MACRO–11 program that accesses the files you specify. You will also learn to write a program that accesses files using a programmed request to accept and parse the command string without fetching handlers or opening channels. Given a list of legal options and correct responses, you will write code that responds to options included in a command string.

Command String Interpreter

The Command String Interpreter (CSI) was implemented to provide programmers with an efficient, standard method of starting file I/O activities. The CSI processes a command string for the program that calls it. The command string can come from either the terminal, an indirect command file, or a buffer in the program. The information in the command string may include:

- Names of files to be used for input
- Names of files to be used for output
- Command options defined by the programmer

The two programmed requests that call the CSI (CSIGEN and CSISPC) are discussed later. The operations that the CSI performs for the program depend on the information in the command string and which CSI programmed request is used. The CSI can be called on to:

- Get a command string from the operator
- Parse a command string
- Convert file specifications from ASCII to RAD50 format
- Fetch device handlers
- Open channels to input and output files
- Return a summary of options to the calling program

Format of a Command String

The CSI processes a command string in the form:

```
OUT1,OUT2,OUT3=IN1,IN2,IN3,IN4,IN5,IN6
```

In this command:

- From 0 to 3 output files are allowed, with the file name format:

DEV:FILNAM.TYP[n]

Device defaults to DK: and the name, type, and number can be omitted if the device is not file structured. The program calling the CSI can specify a default file type. Here "[n]" is the requested length of the output file in blocks. Values are the same as those for the ENTER request's length argument. The default length is 0.

- From 0 to 6 input files are allowed, with the format:

  DEV:FILNAM.TYP

  The comments made above for output files apply to input files also.

- The equal sign (=) separates the list of output files from the list of input files. This symbol must be present if any output files are specified but can be omitted if only input files are specified.

If fewer than three output or six input files are specified, the following conventions apply:

- If the file at the beginning of the list is omitted, indicate the fact by including the comma that would follow it, had it been present.
- If the omitted file is at the end of the list, the comma following it, if any, may be omitted.

Table 18 shows these conventions.

Table 18.
Examples of Command Lines

Files	Command
Three input files only	A,B,C
The first output and first input files	A=B
The second output and no input files	,LP:=
Third output file and second input file	,,Q=,Z

Options or Switches

You can follow any file specification in the command line with an option (also referred to as a switch). The simplest form of an option is:

/X

where "X" is any letter. An option may be followed by either:

- A number: octal (default) or decimal (use a decimal point). For example, /X:5 and /X:12.
- A word of up to three characters. For example, /X:aaa.

Each option value must be preceded by a colon. The following example shows two command strings including options.

EXAMPLE

```
OUTFIL/K:12.=INFIL
OUTFIL,LSTFIL/L:BEX:ME=INFIL
```

The program calling the CSI defines:

- Which options are legal, and the meaning of each
- Whether an option takes a value, which values are legal, and their meaning
- Whether an option applies only to the file that directly follows it, to multiple files, or to the full command line

Program Interface with the CSI

You call the CSI using programmed requests. The CSTRING argument of the programmed request for the CSI specifies the source of the command string. If you want the CSI to take a command string from a buffer within the program, the buffer address is used as the CSTRING argument. Usually, however, you want to take the command string from the operator. If you leave out the CSTRING argument, the CSI uses the GTLIN request to get a command line. To have the CSI get the command string from a buffer within your program rather than from the terminal or a command file, follow these procedures:

- Store the command string in the buffer in ASCIZ format.
- MACRO–11: Supply the address of the buffer as the CSTRING argument.
- FORTRAN IV: The CSTRING argument must be the name of the array containing the command line string.

The DEFTYP argument to the CSI programmed request allows you to specify default file types to be used if the operator omits the file type in the command line. You specify these default file types in a four-word RAD50 argument block. The contents of this block are shown below.

Word 1	Default File Type For All Input Files
Word 2	Default File Type For First Output File
Word 3	Default File Type For Second Output File
Word 4	Default File Type For Third Output File

Use the following procedures to set up and gain access to the four-word argument block:

- If you do not want to recognize default file types for one or more files specified in the default block, make the corresponding words in the block 0 (RAD50 for three spaces).
- MACRO–11: Create the DEFTYP argument by specifying the address of the DEFTYP block.
- FORTRAN IV: Create the DEFTYP argument by specifying the name of the array or variable used as the DEFTYP block.

The address of this block (MACRO–11) or the name of the array being used as the block (FORTRAN IV) is provided as the argument to the CSI requests.

Each file that a user specifies in a command string is assigned a file number in the range 0 to 8 (decimal) or 0 to 10 (octal). These file numbers are assigned as follows:

- Files in a full command string of three output and six input files are assigned the numbers 0 to 8, working from left to right in the command string.
- If any of the nine files are omitted, the numbers that would have been assigned are not used. Figure 65 shows this by giving the file number assignments for a number of sample command strings. The matrix on the right indicates the numbers assigned by the CSI to the files in the command line on the left.

As we have said, options may be accompanied by values. The CSI passes values to your program as follows:

- Numeric values are passed as a one-word binary number (INTEGER data type in FORTRAN IV).
- Values containing alphabetic characters are passed as a single word of RAD50 code.

- An option that has multiple values is taken as multiple occurrences of the same option. For example, if the user types:

  FILENAM/X:5:0

  the option and its values are interpreted as:

  FILENAME/X:5/X:0

Figure 65.
Assignment of File Numbers by the CSI

COMMAND	FILE NUMBERS ASSIGNED								
	0	1	2	3	4	5	6	7	8
A,B,C = L,M,N,O,P,Q	A	B	C	L	M	N	O	P	Q
Z = Y	Z			Y					
,Q = R,S		Q		R	S				
H,I,J				H	I	J			
,V					V				

CSI Modes

The CSI has two different modes of operation: general mode for MACRO–11 only, and special mode for MACRO–11 or FORTRAN IV. Table 19 shows the differences between general and special mode CSI functions.

Calling the CSI in General Mode (.CSIGEN)

CSI (general mode) is accessed using the .CSIGEN programmed request in the following form:

.CSIGEN handler-address, deftyp, cstring, linbuf

In this request:

handler-address is the base address of a buffer into which the CSI is to fetch any needed handlers.

deftyp is the address of the four-word RAD50 block that is used to contain the default file types.

cstring is the address of the command string buffer, or 0 if the command must come from GTLIN.

linbuf is optional. If present in the request, it is the address of a buffer into which the CSI copies the command string in ASCIZ form. (An example is a program to produce a listing file that includes the command line for documentation purposes.) The MACRO–11 assembler carries out this request.

Table 19.
General and Special Mode CSI Functions

General Mode	Special Mode	Both Modes
Fetch handlers LOOKUP input files ENTER output files	Return RAD50 file specifications to program	Get command string from: • Memory • Terminal • Command file Parse command string: • Identify files by file number • Identify options and values • Link option with file number Return option information to calling program Convert file specifications to RAD50 format

Device handlers are fetched to the location that you specify. This is usually the high limit of your program. If multiple handlers are needed, the next handler is fetched into memory at locations above the previous one. On return from the .CSIGEN request, R0 points to the word above the last handler fetched. If none are fetched, R0 contains the handler-address argument.

Each file specification in the command string is opened on the channel given by its file number. Input files are opened using LOOKUP; output files using ENTER.

EXAMPLE

```
FOO.OBJ=FOO.MAC
```

In this command string, FOO.OBJ is opened on channel 0 using ENTER and FOO.MAC is opened on channel 3 using LOOKUP.

When executing a .CSIGEN request, the CSI always closes channels 0 to 8 before performing any ENTER or LOOKUP operations. Therefore, any files that you have opened on these channels before issuing the .CSIGEN request are closed. Any channels not used in the command string are left inactive on return from the .CSIGEN request.

Options are returned on the stack. They are popped off the stack in the order listed below.

1. One word containing the number of options specified in the command string. Remember that multiple option values are returned as repetitions of the option.
2. A one- or two-word descriptor for each option (one word if no value; two words if a value was given).

The return of option information on the stack is shown in figure 66. Figure 67 shows the format of an option word.

Before calling the CSI, save the current stack pointer (SP) if you plan to ignore options or if your program is de-

Figure 66.
Return of Option Information on the Stack

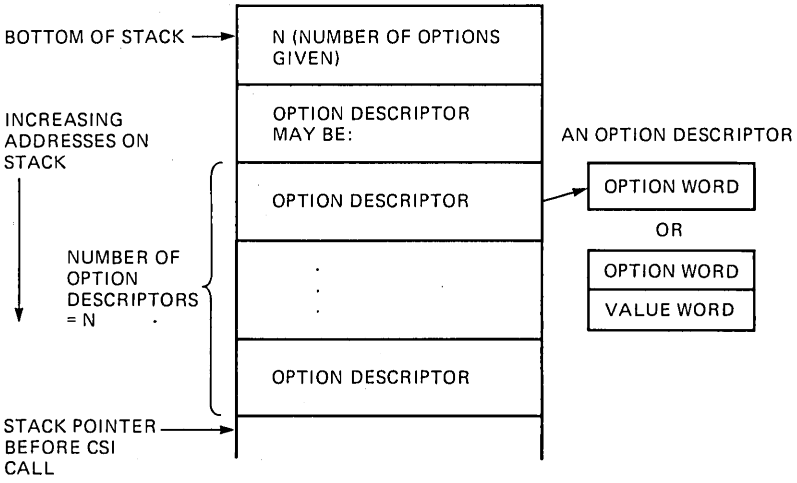
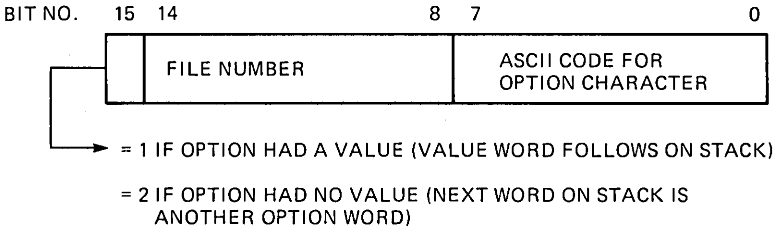

BOTTOM OF STACK
INCREASING ADDRESSES ON STACK
NUMBER OF OPTION DESCRIPTORS = N
STACK POINTER BEFORE CSI CALL
N (NUMBER OF OPTIONS GIVEN)
OPTION DESCRIPTOR MAY BE:
OPTION DESCRIPTOR
OPTION DESCRIPTOR
AN OPTION DESCRIPTOR
OPTION WORD
OR
OPTION WORD
VALUE WORD

Figure 67.
Format of an Option Word


BIT NO.
15
14
8
7
0
FILE NUMBER
ASCII CODE FOR OPTION CHARACTER
= 1 IF OPTION HAD A VALUE (VALUE WORD FOLLOWS ON STACK)
= 2 IF OPTION HAD NO VALUE (NEXT WORD ON STACK IS ANOTHER OPTION WORD)

signed to abort an illegal option. You can use the saved SP value to restore the stack to the state it was in before the CSI call.

Always restore the stack to the state preceding the call, even if no options are expected, because one word (with the value 0) is pushed on the stack even if no options are included. The general flow of control for option processing is as follows:

1. Use the contents of the first word on the stack as a counter for popping off option descriptors.
2. As you pop off each option word, pop off the value word if the option word is negative.
3. Set flags or values for use in the program.
4. Reset the stack.

As an example of CSI general mode, see the segment of code in figure 68. This code is set up to function as part of a program that will accept /C as an option.

Figure 68.
Code Showing Option Processing under General Mode

```
        CLR     R4              ;USE R4 AS FLAG WORD
        MOV     SP,SAVSP        ;SAVE STACK POINTER
        .CSIGEN _#HSPACE,_#DEFEXT,_#0
                                ;GET CSI STRING FROM KEYBOARD
        TST     (SP)            ;OPTIONS SET?
        BEQ     CLEAN           ;NO, CONTINUE
        CMP     (SP)+,_#1       ;ONE OPTION?
        BNE     CLEAN           ;NO, CONTINUE
        CMPB    (SP)+,_#'C      ;YES, WAS IT C?
        BNE     CLEAN           ;NO, IGNORE
        INC     R4              ;YES, SET FLAG FOR LATER USE
CLEAN:  MOV     SAVSP,SP        ;RESTORE STACK POINTER
        .
        .
        .
        .
DEFEXT: .WORD   0,0,0,0
SAVSP:  .WORD   0
HSPACE: .BLKW   1024.           ;HANDLER SPACE
```

Calling the CSI in Special Mode (.CSISPC or ICSI)

Whether you are programming with MACRO–11 or FORTRAN IV, you can use CSI special mode. You may want to use it to check a command string in a MACRO–11 program before proceeding with operations such as fetch, lookup, or enter. You may also use it to check for:

- Reserved file names
- The number of files in the command
- The output file size
- Old versions of output file specifications (before doing an enter)
- Default file types to be determined by options (for example, LIBR can produce either .OBJ or .MAC files)

The MACRO–11 request for calling CSI in special mode is:

```
.CSISPC   files, deftyp, string, linbuf
```

Here "files" is the address of a 39-word block to be used for file specifications and "linbuf" is the storage address for the original command string in ASCII. All other arguments for this request are identical to those needed by the .CSIGEN request.

The FORTRAN IV request for CSI access is:

```
IERR=ICSI (files,deftyp,cstring,options, N)
```

In this request:

files — is an array of 39 words to be used to hold file specifications.

deftyp — is a four-word area containing RAD50 format default file types.

cstring — if present, is an array containing a command string in ASCIZ format. If this argument is omitted, CSI expects to receive the command string using GTLIN.

options — can be omitted if N is 0; if N is not 0, this argument must be the name of an array of size 4*N. This array is used by ICSI to return option information.

If this array has the dimensions:

INTEGER OPTIONS (4,N)

then all the elements of the array have the meaning indicated in table 20 for all values of J from 1 to N.

N is the maximum number of options allowed.

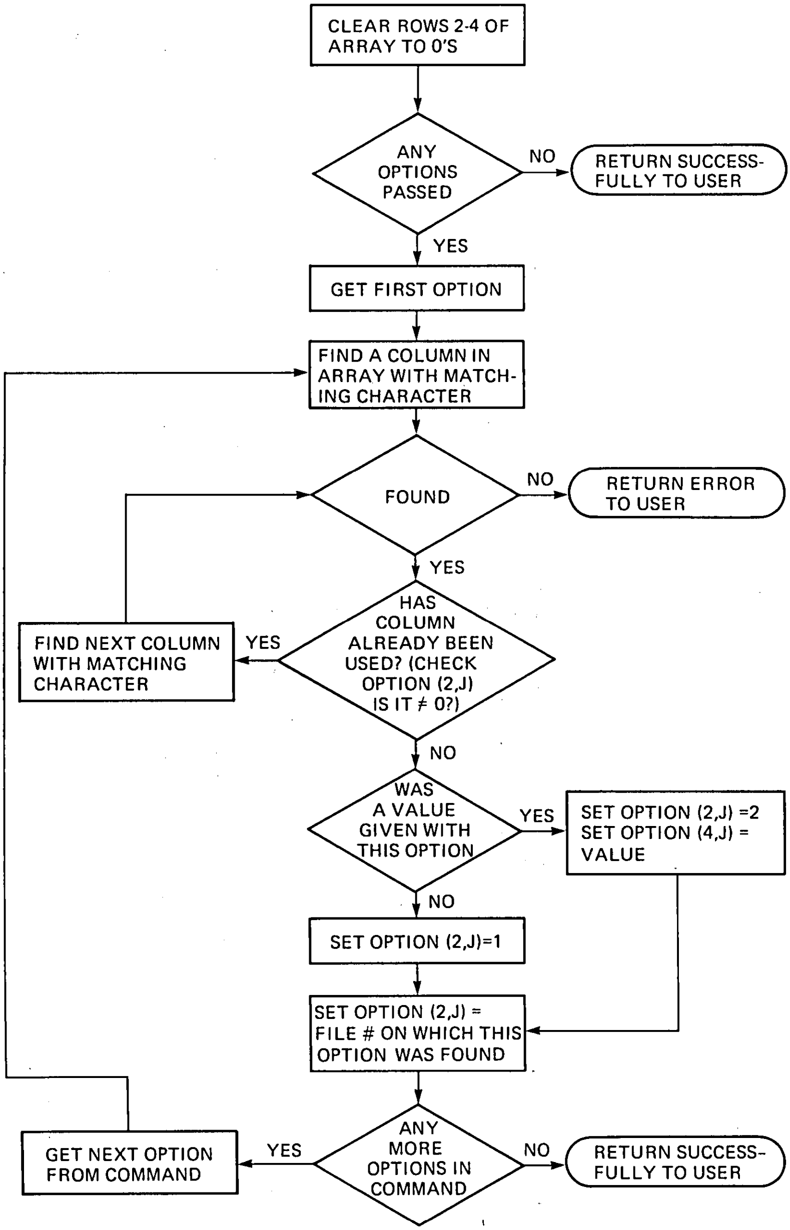
In setting up calls to ICSI, fill in the first row of the options array before calling ICSI. Follow these procedures:

- Place any legal option in at least one column of the options array.
- If a specific option can be used legally more than once in a command, it must head as many columns

Table 20.
Information Contained in Option Array

Name of Element	Description Contents	Default Value	Filled in By	Notes
OPTIONS(1,J)	ASCII code for option character	None	User job	Must be filled in before calling ICSI
OPTIONS(2,J)	"Option present"	0	ICSI	1 means option appeared without a value 2 means option appeared with a value (if 0, then option did not appear in commands)
OPTIONS(3,J)	File number	0	ICSI	Only important if OPTIONS(2,J).NE.O
OPTIONS(4,J)	Value	0	ICSI	Only important if OPTIONS(2,J) = 2

Figure 69.
ICSI Filling in the Options Array


CLEAR ROWS 2-4 OF ARRAY TO 0'S
ANY OPTIONS PASSED
NO
RETURN SUCCESSFULLY TO USER
YES
GET FIRST OPTION
FIND A COLUMN IN ARRAY WITH MATCHING CHARACTER
FOUND
NO
RETURN ERROR TO USER
YES
HAS COLUMN ALREADY BEEN USED? (CHECK OPTION (2,J) IS IT ≠ 0?)
YES
FIND NEXT COLUMN WITH MATCHING CHARACTER
NO
WAS A VALUE GIVEN WITH THIS OPTION
YES
SET OPTION (2,J) =2 SET OPTION (4,J) = VALUE
NO
SET OPTION (2,J)=1
SET OPTION (2,J) = FILE # ON WHICH THIS OPTION WAS FOUND
ANY MORE OPTIONS IN COMMAND
YES
GET NEXT OPTION FROM COMMAND
NO
RETURN SUCCESSFULLY TO USER

as the maximum number of times it can appear in the command.

- Remember that an option with multiple values (for example, /X:5:2) is taken as the same option repeated with different values (/X:5/X:2).

The flowchart in figure 69 shows the procedure used by ICSI to fill in the options array. Read this flowchart in conjunction with table 20 because it shows how the array described in the table is filled in when the ICSI request is used.

The programs (PR1901.MAC and PR1901.FOR) which follow show the use of the two CSI modes. Both programs execute a single-buffered copy routine based on the programs introduced in chapter 15, "Using Nonsynchronous Queued Input/Output." In the MACRO–11 routine (PR1901.MAC), the command string is processed using the CSI in general mode. The FORTRAN IV program (PR1901.FOR) performs the same functions. The CSI special mode is used in this program.

When called in special mode; the CSI returns information on options and file specifications to the user job. The information on file specifications is returned in RAD50 format, in a 39-word block. You allocate the full 39-word block, even if you expect only one or two files. The CSI fills any unused areas with zeros. The contents of such a block are shown in figure 70.

Practice 19–1

1. Write a program named PR1902 (in either MACRO–11 or FORTRAN IV) that performs the following operations:

   a. Accepts a CSI command string from the terminal.

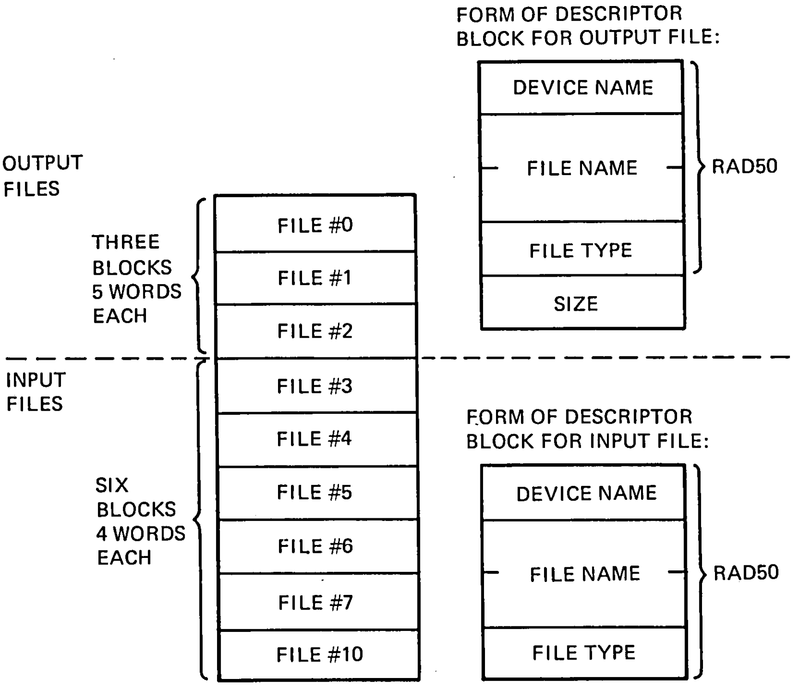
   b. Creates a file for output with the name specified in the command string (ignore input file specifications).

c. Generates 10 blocks of data containing the positive integers 1 to 2560 stored in sequence. Each block of data is to be written out to the file after it is generated, using synchronous write programmed requests.

d. Closes the file.

2. Run the program and specify an output file name. Check the output file contents using the DUMP utility.

Figure 70.
File Specifications Returned by CSI Special Mode in a 39-word Block



PR1901.MAC

```
        .TITLE  PR1901  SAMPLE I/O PROGRAM
;
;       Sample file copy program using the CSI to
;       allow user to specify input and output files.
;
        .MCALL  .EXIT   .CSIGEN .PRINT
        .MCALL  .READW  .WRITW  .CLOSE  .SRESET
EMTARG: .BLKW   6               ;EMT argument block
LIMITS: .LIMIT                  ;Program limits
SPSAVE: .BLKW   1               ;Saved stack pointer
DEFTYP: .WORD   0,0,0,0         ;Default file exts
BUFFER: .BLKW   256.            ;File I/O Buffer
ERROR:  .BYTE                   ;Error status byte
PRMPT:  .ASCIZ  "Specify OUTFILE=INFILE"
CSIERR: .ASCIZ  "Error on CSI call"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine prompts for and opens both
;       an input and an output file.  The CSI is
;       used to collect and process a command.
;
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     SP,SPSAVE       ;Save stack pointer
        .PRINT  #PRMPT          ;Prompt for files
        MOV     LIMITS+2,R1     ;Load high limit
        .CSIGEN R1,#DEFTYP,#0   ;Call CSI
        MOV     SPSAVE,SP       ;Reset stack pointer
        BCC     10$             ;Branch on success
        .PRINT  #CSIERR         ;Output error message
        SEC                     ;Indicate failure
10$:    RETURN                  ;Return to caller
        .SBTTL  CPYRTN  -- Synchronous Copy
;
;       This routine copies the input file opened on
;       channel 3 to the output file opened on
;       channel 0.
;
;       Returns with C-BIT SET on error.
;
;       Note: All registers except R0 are preserved.
;
CPYRTN: MOV     R1,-(SP)        ;Save register
        CLR     R1              ;Reset block number
        CLRB    ERROR           ;Clear error flag
10$:    .READW  #EMTARG,#3,#BUFFER,#256.,R1
```

PR1901.MAC (continued)

```
                BCC     20$             ;Branch if read OK
                TSTB    @#52            ;End-Of-File (EOF)?
                BEQ     EXIT            ;Branch if so
                BR      RDERR           ;Issue error message
        20$:    .WRITW  #EMTARG,#0,#BUFFER,#256.,R1
                BCS     WERR            ;Branch on write error
                INC     R1              ;Update block number
                BR      10$             ;And read next block
        RDERR:  .PRINT  #RERRMS         ;Issue read error
                BR      EREXIT          ; message and exit
        WERR:   .PRINT  #WERRMS         ;Issue write error
        EREXIT: DECB    ERROR           ;Set error flag
        EXIT:   MOV     (SP)+,R1        ;Restore register
                TSTB    ERROR           ;Set C-Bit?
                BEQ     10$             ;Branch if not
                SEC                     ;Set C-Bit
        10$:    RETURN                  ;Return to caller
                .SBTTL  CLSCHN  -- Close Files
        CLSCHN: .CLOSE  #3              ;Close input file
                .CLOSE  #0              ;Close output file
                BCC     RESET           ;Branch on success
                .PRINT  #PRTCT          ;Issue protected file
        PRGCHN:                         ;Purged files
        RESET:  .SRESET                 ;Reset system
                RETURN                  ;Return to caller
                .SBTTL  MAIN PROGRAM
        START:  CALL    SETUP           ;Open files
                BCS     START           ;Repeat on error
                CALL    CPYRTN          ;Copy file
                BCS     1$              ;Branch on error
                CALL    CLSCHN          ;Close the files
                BR      2$
        1$:     CALL    PRGCHN          ;Purge the channels
        2$:     .EXIT                   ;Exit
                .END    START
```

PR1901.FOR

```
        PROGRAM PR1901
C
C       Sample file copy program using the CSI to
C       allow user to specify input and output files.
C
        LOGICAL*1 SETUP,CPYRTN
        LOGICAL*1 ERROR
C
10      ERROR = SETUP()         ! Open files
        IF (ERROR) GO TO 10     ! Repeat on error
        ERROR = CPYRTN()        ! Copy file
        IF (ERROR) GO TO 20     ! Stop on error
        CALL CLSCHN             ! Close files
        GO TO 30
20      CALL PRGCHN             ! Purge channels
30      CALL EXIT
        END
        FUNCTION SETUP
C
C       This routine sets up the files for I/O.
C
C       Function returns .TRUE. on error.
C
        LOGICAL*1 SETUP
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        LOGICAL*1 FRSTTM
        DATA FRSTTM/.TRUE./
        INTEGER*2 DEFTYP(4),FILES(39),LENGTH
        DATA DEFTYP/4*0/
        IF (.NOT. FRSTTM) GO TO 10
C
C       Allocate channels only once.
C
        INCHN = IGETC()
        OUTCHN = IGETC()
        FRSTTM = .FALSE.
C
C       Output prompt and get command line,
C
10      CALL PRINT('Specify OUTFILE=INFILE')
        IF (ICSI(FILES,DEFTYP,,,0) .NE. 0) GO TO 100
C
C       Fetch device handlers
C
        IF (IFETCH(FILES(1)) .NE. 0) GO TO 101
        IF (IFETCH(FILES(16)) .NE. 0) GO TO 102
C
C       Open input and create output file.
C
```

PR1901.FOR (continued)

```
        LENGTH = LOOKUP(INCHN,FILES(16))
        IF (LENGTH .LT. 0) GO TO 103
        IF (IENTER(OUTCHN,FILES(1),LENGTH) .LT. 0)
     1     GO TO 104
        SETUP = .FALSE.         ! Success
        RETURN
C
C       ERROR ROUTINES
C
100     CALL PRINT('Error on CSI call')
        GO TO 200
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.          ! Error
        RETURN
        END
        FUNCTION CPYRTN
C
C       Single buffered, synchronous copy routine.
C
C       Function returns .TRUE. on error.
C
        LOGICAL*1 CPYRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFFER(256),BLOCK
        BLOCK = 0               ! Reset block number
C
C       Read/write loop.
C
20      IERR = IREADW(256,BUFFER,BLOCK,INCHN)
        IF (IERR .GE. 0) GO TO 30 ! Successful read
        IF (IERR .EQ. (-1)) GO TO 150 ! End of File
        GO TO 100               ! Issue error message
30      IF (IWRITW(256,BUFFER,BLOCK,OUTCHN) .LT. 0)
     1     GO TO 101
        BLOCK = BLOCK+1         ! Update block number
        GO TO 20                ! Read next block
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
101     CALL PRINT('Write error, copy aborted')
140     CPYRTN = .TRUE.
        RETURN
```

PR1901.FOR (continued)

```
C
C       Successful return.
C
150     CPYRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        RETURN
        END
```

Reference

RT–11 Programmer's Reference Manual. Chapter 2 discusses the preparation of a CSI call by using data structures for ICSI (FORTRAN IV) or .CSISPC (MACRO–11) request. .CSIGEN requests are also covered here.

20

20

Writing Time-dependent Programs

In many programming applications, you need to sample data after regular or random periods of time or perform tasks that run regularly or at set times. RT–11 runs on systems without a clock, but time-dependent routines require a KW11–L or KW11–P system clock. This chapter describes how the system stores and maintains the system date and time. It then explains how to use programmed requests and subroutines to access and use this information.

Some systems have timer service support, a feature that allows you to schedule programs to be run at a given time of day or after a given period of time has elapsed. This chapter explains how to write such programs.

The MACRO–11 programmed requests and subroutines discussed in this chapter include: .CMKT, .DATE, .GTIM, .MRKT, .SDTTM, and .TWAIT. The FORTRAN IV subroutines discussed include: CVTTIM, DATE, GTIM, ICMKT, IDATE, ISCHED, ISDTTM, ISLEEP, ITIMER, ITWAIT, IUNTIL, JJCVT, JTIME, MRKT, SECNDS, TIMASC, and TIME.

If your system has a system clock, you will be able to write programs that convert the current system date and time into numeric or ASCII format. You will also be able to write a program that sets the system date and time. If your system supports timer service, you will be able to write programs that run at a specified time of day, run after a specified period of time, or suspend program execution for a specified length of time.

System Time and Date

You can use the time and date features of RT–11 only if your system has a system clock. The system clock may be:

- A KW11–L line frequency clock which interrupts or ticks, once for each cycle of the ac line (that is, 60 ticks per second for a 60-Hz line, and 50 ticks per second for a 50-Hz line)
- A KW11–P programmable real-time clock which can tick at line frequency (60 Hz in the United States and Canada; 50 Hz in Europe, Mexico, and Australia) or can be programmed to tick at a different rate

For either device, the system clock interrupt service routine is entered each time the clock ticks.

Maintaining the System Time

RT–11 maintains the time of day in two 16-bit words which indicate the time in terms of ticks past midnight. Unless you use the TIME command to set the time, the time the system displays represents the time that has elapsed since bootstrap.

The time is automatically reset after midnight under the FB monitor, but not under the SJ monitor unless this has been selected as a system generation option.

Each pair of words used to represent the system time is considered as a 32-bit interger value. The first word holds the high-order part of the number and the second word the low-order part. This 32-bit format is referred to as the internal format. In the following examples of MACRO–11 and FORTRAN IV code, the data area TIMER contains the internal format representation for 60 seconds at 60 Hz (3600 ticks).

EXAMPLE

MACRO-11

```
TIMER:  .WORD 0,3600.   ;3600 ticks
```

FORTRAN IV

```
INTEGER*2 TIMER(2)   !2 consecutive words
DATA TIMER/0,3600/   !3600 ticks
```

Accessing the System Time

A number of requests are available to programs to get the system time, either in internal format (ticks—high order first, low order second) or in hours, minutes, and seconds.

The MACRO–11 request .GTIM returns the current system time in internal format and indicates the number of ticks since midnight (or since the last bootstrap). The format of the request is:

```
.GTIM   area,addr
```

In this request, "area" is the address of a two-word EMT argument block. Here "addr" is the address of a two-word area in which the monitor stores the system time. The .GTIM request is described in chapter 2 of the *RT–11 Programmer's Reference Manual.*

The FORTRAN IV subroutine GTIM is equivalent to the MACRO–11 .GTIM request. The format of the call is:

```
CALL GTIM(itime)
```

Here "itime" is a two-word area in which the system time is stored in internal format.

You can also use the SECNDS function to get the time. The function displays time as the number of seconds since midnight or a given time of day. The format of the function is:

time=SECNDS(sttim)

In this function, "time" is a REAL*4 variable to store the returned value (in seconds); "sttim" is a REAL*4 expression containing the start time, in seconds since midnight. If "sttim" is 0, the value returned is the current time of day (in seconds since midnight).

The following example shows how you can find the execution time, in seconds, for a sequence of FORTRAN IV statements.

EXAMPLE

```
T1=SECNDS(0.) !Get current time in secs.
   .
Instructions to be timed
   .
   .
T2=SECNDS(T1) !Number of secs. elapsed
              !since start time (T1)
```

The TIME subroutine returns the system time in hours, minutes, and seconds as an ASCII character string.

EXAMPLE

```
09:56:14
```

You should note that the 24-hour clock is used; for example 1:00 p.m. is returned as 13:00:00. The format of the subroutine call is:

CALL TIME(string)

Here "string" is a variable or array eight bytes in length. In the following example, the FORTRAN IV code requests the current system time and prints it out at the terminal.

EXAMPLE

```
      REAL*8 STIME        !To hold system time
      CALL TIME(STIME)    !Get time in ASCII
      TYPE 100,STIME
100   FORMAT (' The time is ',A8)
```

Three conversion routines are available to change current time or elapsed time from one format to another.

- CVTTIM. Converts from internal time format to the integer number of hours, minutes, seconds, and ticks.
- JTIME. Converts from the integer number of hours, minutes, seconds, and ticks to internal time format.
- TIMASC. Converts from internal time format to ASCII character string in the format hh:mm:ss.

Converting the System Time to 32-bit Integers

The standard format for 32-bit integers under RT–11 is as follows:

- The low-order data is stored in the first word.
- The high-order data is stored in the second word.

The RT–11 internal time format is the reverse of this arrangement. You can use the JJCVT request to convert between the two formats (in either direction), as follows:

CALL JJCVT(ivar)

Here "ivar" is the INTEGER*4 variable whose high-order and low-order words are to be reversed.

In order for the words to be printed in the correct order, you should call JJCVT to convert internal format system times to standard 32-bit integer format before you use formatted I/O routines to print them out. In the following example the code gets the time of day (in internal format), converts it to standard RT–11 32-bit integer format, then prints it.

EXAMPLE

```
        INTEGER*4 TIME    !Internal format time
        CALL GTIM(TIME)   !Convert to INTEGER*4
        CALL JJCVT(TIME)
        TYPE 10,TIME
10      FORMAT (' The clock has ticked ',19,
     1          ' times since midnight.')
```

A number of other routines for using 32-bit integers are available to FORTRAN IV programmers. See the *PDP–11 FORTRAN Language Reference Manual* for RT–11 for additional information.

Maintaining the System Date

The RT–11 system date is stored in a single word. The month, day, and year are held in numeric format. To arrive at the number that represents the year, the system subtracts 1972 from the current year. For example, the date May 30th, 1983 would be stored as the following numbers:

- Month 5 (must be in the range 1 to 12)
- Day 30 (must be in the range 1 to the length of the month)
- Year 11 (1983 minus 1972).

The format of the RT–11 system date word is shown in figure 71.

The internal system date may be set or changed by any of the following events:

- The user issues the DATE monitor command to set the current date.
- The date changes at midnight. However, RT–11 does not change the date until a job requests time of day with a .GTIM request.
- The month and year values roll over. This occurs when month and year roll over has been selected at system generation time.
- The DATE monitor command is issued to print the date.
- A job issues a .SDTTM or ISDTTM request to change the system date (described later).

Figure 71.
Format of the System Date Word

15	14	13 10	9 5	4 0
0	0	MONTH (1–12)	DAY (1–31)	YEAR (MINUS 1972)

Accessing the System Date

A number of requests can be used in MACRO–11 and FORTRAN IV programs to access the system date and convert it to different formats.

Accessing the system date from MACRO–11 programs

To get the current system date in the internal format described above, issue the .DATE request, as follows:

```
.DATE
```

This returns the date in R0. Notice that the system date does not change at midnight unless one of the events listed above occurs. To make sure that you get the correct date, it is recommended that you issue a .GTIM request before the .DATE request.

EXAMPLE

```
DATE:    .WORD    0              ;Save date here
           .
           .
           .
         .GTIM    #AREA,#TIME    ;Get time
         .DATE                   ;and date
         MOV      R0,DATE        ;Save current date
```

Accessing the system date from FORTRAN IV programs

The DATE subroutine gets the system date in ASCII format, dd-mmm-yy. For example, 28-APR-84. The format of the call is:

CALL DATE(date)

In this call, "date" is a variable or array at least nine bytes in length. If no system date has been set, "date" contains spaces on return from the subroutine.

EXAMPLE

```
LOGICAL*1 TODAY(9)
CALL DATE(TODAY)
```

You can also get the month number, the day, and the last two digits of the year, with the IDATE subroutine. The format of the call is:

CALL IDATE(month,day,year)

where all the arguments are INTEGER*2 variables.

EXAMPLE

```
INTEGER*2 MONTH,DAY,YEAR
CALL IDATE(MONTH,DAY,YEAR)
```

If today's date is April 28, 1984, then after executing these statements, "MONTH" contains 4, "DAY" contains 28, and "YEAR" contains 84. If no system date has been set, these variables contain 0 on return from the subroutine.

The DATE and IDATE subroutines can only be used in FORTRAN IV programs.

Writing Programs Independent of Line Frequency

Programs that use the system time and date normally depend on the frequency of the ac line supplying the hardware. You may want to write programs that perform computations based on time, but independent of the line frequency, for example, programs which will run on systems in Europe as well as in the United States.

At run time, your programs need to find out whether they are being run on a 50–Hz or 60–Hz system. This information is held in the RMON configuration word (offset 300 in the fixed offset area).

Setting the System Time and Date

You have seen that you can set the system time and date using the TIME and DATE monitor commands. You can also set them from your program using the following methods.

Setting the time and date from MACRO–11 programs

To change the system time or date, issue the .SDTTM programmed request as follows:

```
.SDTTM    area, addr
```

In this request, "area" is the address of a two-word EMT argument block and "addr" is the address of a three-word block with the following contents:

Word 1:	The new date (internal format), or any negative number if you want to leave the date unchanged.
Words 2 and 3:	The new system time (internal format, with the high order in word 2, low order in word 3). Place any negative number in word 2 if you want to leave the time unchanged.

Setting the time and date from FORTRAN IV programs

The FORTRAN IV equivalent of the MACRO–11 .SDTTM request is ISDTTM. Another way of setting the system time and date from a FORTRAN IV program is to pass the TIME and DATE monitor commands to KMON when the program exits. To do this, use the SETCMD subroutine as described in chapter 10, "Controlling Program Execution." If you want to set both the date and the time or pass any two or more monitor commands, then you should place these commands in an indirect file and pass the command to execute the file to KMON.

A second method of setting the system date and time from a FORTRAN IV program is by calling a MACRO–11 subroutine that issues the .SDTTM programmed request. Table 21 lists the requests and subroutines that access the system time and date.

Mark-time Routines

A mark-time routine is a subroutine that you request to be executed at a specific time of day, or after a specific period

Table 21.
RT-11 Time and Date Requests

Request	Function	FORTRAN IV	MACRO-11
.GTIM	Get current time in ticks since midnight		*
GTIM	Get current time in ticks since midnight	*	*
SECNDS	Get current or elapsed time in seconds	*	*
TIME	Get time in ASCII format	*	*
CVTTIM	Convert internal time to hrs,mins,secs,ticks	*	*
JTIME	Convert hrs,mins,secs, ticks to internal time	*	*
TIMASC	Convert internal time to ASCII hh:mm:ss	*	*
.DATE	Get date in internal format		*
DATE	Get date as ASCII string in format dd-mmm-yy	*	
IDATE	Get date as integer month,day,year	*	
.SDTTM	Set system date and time		*
ISDTIM	Set system date and time	*	*

of time has elapsed. Your program can issue programmed requests to schedule a mark-time routine to be run, and if you need, to cancel such requests.

These requests are available only on systems that have timer service support. This feature is provided in all systems under the FB or XM monitors, but it is a system generation option under the SJ monitor.

Mark-time routines run as completion routines. When you issue a mark-time request from a program, RMON builds a timer queue element, which is placed in the timer queue. This queue is managed by the same routines that serve clock interrupts. The elements in the queue are sorted by the time at which they expire, with the element that expires first at the head of the queue.

Each time a clock interrupt occurs, the clock service routine checks to see if there is a timer queue element whose time limits have expired. If there is, it is removed from the timer queue and placed in the completion queue for the job that issued the mark-time request.

Timer queue elements are taken from a pool of I/O queue elements. Your program must allocate at least as many queue elements as the number of mark-time and I/O requests that expect to be pending simultaneously. If you need to change the number, use the .QSET request (MACRO–11) or the IQSET subroutine (FORTRAN IV) as described in chapter 15, "Using Nonsynchronous Queued Input/Output." When there are insufficient queue elements in the pool, the mark time request returns an error.

Scheduling Mark-time Routines

When you issue a mark time request, an entry is placed in the timer queue. You have to specify the address of the completion routine and the time that must elapse before the completion routine is to be run. Note that this is not the time of day at which the routine should be run. The time must be given as a number of ticks, in internal format (high-order word first, low-order word second).

You also specify a nonzero identification number, or id number, for each mark-time request. It must not be in the range 177000 (octal) to 177777 (octal), as these numbers are reserved for system use. The id number allows you to cancel specific mark time requests, as you will see later. Id numbers need not be unique; you may specify the same id number for more than one mark-time request.

Issuing MACRO–11 mark-time requests

You use the .MRKT programmed request to issue a mark time, as follows:

```
.MRKT    area,time,crtn,id
```

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

time is the address of a 2-word area containing the time that must elapse (in ticks, internal format).

crtn is the entry point (start address) of the completion routine (written in MACRO–11).

id is the identification number. On entry to the completion routine, the id number is in R0.

Since the expiration time is given in terms of ticks until the event, it is independent of the system time set by the monitor command TIME or the .SDTTM programmed request. If you change the time of day, the expiration time of mark-time requests does not change.

In the following example the MACRO–11 code schedules the completion routine COMPL to be run after 3600 ticks.

EXAMPLE

```
ELAPS:   .WORD    0,3600.   ;Elapsed time
AREA:    .BLKW    4         ;EMT argument block
            .
            .
;ISSUE MARK TIME FOR 3600 TICKS, ID=4
         .MRKT    #AREA,#ELAPS,#COMPL,#4
            .
            .
COMPL:                      ;Completion routine
            .
            .
         RTS      PC
```

Issuing FORTRAN IV mark-time requests

FORTRAN IV programmers have a choice of three mark-time requests. Each of the functions returns a zero value to indicate a normal return, or nonzero if there are insufficient queue elements in the pool. The three functions are:

1. The MRKT function is equivalent to the MACRO–11 .MRKT request. The format of the call is:

   IERR = MRKT(id,crtn,time)

   The completion routine must be written in MACRO–11 and specified in an EXTERNAL statement in the FORTRAN IV routine that issues the MRKT call. "time" is the period of time to pass until the routine is run.

2. The ITIMER function is similar to MRKT, except that the completion routine is written in FORTRAN IV, and the period of time to pass is specified in hours, minutes, seconds, and ticks. The format of the call is:

   IERR = ITIMER(hrs,mins,secs,ticks,area,id,crtn)

3. The ISCHED function is similar to ITIMER, except that the time specified is the time of day at which the routine is to be run, not the period of time which is to elapse. The time is specified in hours, minutes, seconds, and ticks, and the completion routine is written in FORTRAN IV. The format of the call is:

   IERR = ISCHED(hrs,mins,secs,ticks,area,id,crtn)

The arguments for these requests are as follows:

id	is a nonzero identification (id) number.
crtn	is the address of the completion routine. This routine may be written in MACRO–11 or FORTRAN IV, depending on the request you use. The name of the routine must be specified in an EXTERNAL statement in the FORTRAN IV routine that issues the mark time request.
area	is the address of a four-word block in your program, similar to the one used in event-driven I/O.
time	is a time specified in internal RT–11 time format.
hrs, mins, secs, ticks	are integer numbers of hours, minutes, seconds, and ticks.

Cancelling Mark-time Requests

You can write your programs so that you can cancel one or more scheduled mark-time requests if you detect certain conditions. To do so, use a programmed request that removes the timer queue elements from the timer queue. If a timer queue element has already expired and been transferred to the completion queue, then you cannot cancel it.

To cancel a specific mark-time request, you must specify the same id number you gave in the original mark-time request. If there is more than one element in the queue with the same id number, the one which is to expire first is cancelled.

To cancel all the mark-time requests in the timer queue, you should specify an id number of 0. Notice that system mark-time requests, those with an id number in the range 177000 (octal) to 177777 (octal), are not affected. If you specify a nonzero id number, you can supply another parameter to the cancel mark-time request in which the monitor returns the amount of time remaining for the cancelled request.

Cancelling MACRO–11 mark-time requests

To cancel a mark time request, use the .CMKT request, as follows:

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

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

id — is the identification number specified in the original mark time request, or 0 to cancel all remaining mark time requests.

time — (optional) is the address of a two-word area in which the monitor returns the time remaining for the cancelled request (internal format: high-order word first, low-order word second).

This request sets the C bit if there are no elements in the queue with the specified id. The following example shows the code to cancel all mark-time requests with the id number 5.

EXAMPLE

```
1$:     .CMKT   #AREA,#5,#0  ;Cancel mark time
        BCC     1$
```

Cancelling FORTRAN IV mark-time requests

Use the ICMKT function to cancel one or more mark-time requests, as follows:

IERR = ICMKT(id,time)

id — is the identification number specified in the original mark time request, or 0 to cancel all remaining mark time requests.

time — is the name of a two-word area in which the monitor returns the time remaining for the cancelled request (internal format—high-order word first, low-order word second).

The function returns a nonzero value if there is no element with the specified id in the timer queue. The following example shows the code to cancel all mark-time requests with the id number 5.

EXAMPLE

```
        INTEGER*4  TLEFT
10      IF (ICMKT(5,TLEFT).EQ.0)GO TO 10
```

The value returned is 0, as long as there is an active request with the specified id (5 in this example).

Periodic Scheduling

Scheduling a routine to run periodically is a common application of the RT–11 mark-time requests. For example, you can schedule a routine to sample data regularly. The smallest time period you can use for sampling data is one clock tick. This is 16.67 millesecond for 60-Hz systems or 20 millesecond for 50-Hz.

In order to do periodic scheduling, you issue the mark-time request in the completion routine. This request places another entry in the timer queue each time you enter the routine.

Designing Watchdog Routines

In data acquisition applications it is often important to determine whether or not an I/O event has occurred within a critical period of time. For example, if an instrument in a nuclear plant is designed to generate an interrupt every two minutes, the computer monitoring system may have to take action if the expected interrupt fails to occur (device failure). Routines that check for such events are called watchdog routines.

The sequence of steps involved in the general design scheme for a watchdog routine is:

1. Issue an event-driven I/O request and a mark-time request for an alarm routine. The alarm routine is entered if the I/O request is not executed.
2. In the completion routine for the event-driven I/O request, cancel the mark-time request, issue a new event-driven I/O request, then issue a new mark-time request.
3. If the event-driven I/O fails to occur, the alarm routine scheduled by the mark-time request is entered. This routine should alert the operator, or take other measures such as turning off devices.

Timed Waits

You may want to suspend execution of your program for a specified period of time if certain conditions occur. RT–11 provides programmed requests to do this, but you can use them only if your system has timer service support. All of these requests need a timer queue element, so you have to count them when you determine the number of queue elements to allocate to your program with .QSET or IQSET. These requests suspend execution of your main program only. Any completion routines from previous mark-time or nonsynchronous queued I/O requests continue to execute.

Suspending execution of a MACRO–11 program

To suspend execution of your program for a specified period of time, use the .TWAIT request as follows:

```
.TWAIT   area,time
```

Here "area" is the address of a two-word EMT argument block and "time" is a pointer to two words containing the time (in ticks) in internal format (high-order word first, low-order word second) for which the job is to be suspended. The following example shows program execution suspended for 5000 ticks.

EXAMPLE

```
TIME:    .WORD    0,5000.         ;Time to wait
AREA:    .BLKW    2               ;EMT argument block
              .
              .
         .TWAIT   #AREA,#TIME  ;Suspend for 5000
                               ;ticks
```

Suspending execution of a FORTRAN IV program

FORTRAN IV programmers can use any of three different requests to suspend program execution. Each function returns the value zero to indicate a normal return, or a nonzero value if no more queue elements are available:

1. The ITWAIT function is equivalent to the MACRO–11 .TWAIT request. The format of the request is:

   IERR = ITWAIT(time)

   In this request, "time" is the two-word internal format (high-order word first, low-order word second) time period for which the program is to be suspended. For example, the following code suspends program execution for 3600 ticks:

   ```
   INTEGER*2 TIME
   DATA    TIME/0,3600/
   IERR=ITWAIT(TIME)  !Suspend program
   ```

2. The ISLEEP function suspends job execution for a period of time specified in hours, minutes, seconds, and ticks. The format of the request is:

   IERR = ISLEEP(hrs,mins,secs,ticks)

   Here "hrs,mins,secs,ticks" are the integer numbers of hours, minutes, seconds, and ticks for which job execution is to be suspended. For example, the following statement suspends program execution for three hours and five minutes:

   ```
   IERR=ISLEEP(3,5,0,0)
   ```

3. The IUNTIL function suspends program execution until the specified time of day. The format of the request is:

   IERR = IUNTIL(hrs,mins,secs,ticks)

   Here "hrs,mins,secs,ticks" are integer numbers representing the time of day at which the job is to start execution again. For example, the following statement suspends program execution, and starts execution again at 15:45:00:

   ```
   IERR=IUNTIL(15,45,0,0)
   ```

Table 22 lists the requests and subroutines that issue mark time requests and timed waits.

Practice 20–1

You can do this exercise only if your computer has a system clock and timer service support.

1. Write a program, in MACRO–11 or FORTRAN IV, to perform the following:

   a. Get the system time and date. If the date has not been set, the program exits, printing a message asking the operator to set the date and time. If the date has been set, the program displays the current time and date, and continues to step b.

   b. Allocate enough queue elements for the remainder of the program, and set up a completion routine to run for 20 seconds. The completion routine should set a flag to indicate that it has run. If the request to set up the completion routine fails, print an error message and exit; otherwise go to step c.

   c. Test to see whether the completion routine has run. If it has, go to step e. Otherwise display the message:

   ```
   Type as much as you can in 2 seconds, starting
   now:
   ```

   Enable terminal special mode and error returns for terminal I/O (bits 12 and 6 of the JSW). Then wait for 2 seconds.

   d. When 2 seconds have passed, read all the characters in the I/O buffer, and print out the message:

   ```
   You managed to type:
   ```

   followed by the input text. Then go to step e.

   e. Get and display the current time and date, then exit.

2. Test your program. See how many characters you can type in during each cycle. Check that the program times out after 20 seconds.

Table 22.
RT-11 Mark-time and Time-wait Requests

Request	Function	FORTRAN IV	MACRO-11
.MRKT	Mark time for period (internal format)		*
MRKT	Mark time for period (internal format)	*	*
ITIMER	Mark time for period (hrs,mins,secs,ticks)	*	*
ISCHED	Mark time until time of day (hrs,mins,..)	*	*
.CMKT	Cancel one or all mark-time requests		*
ICMKT	Cancel one or all mark-time requests	*	*
.TWAIT	Wait for timeout period (internal format)		*
ITWAIT	Wait for timed period (internal format)	*	*
ISLEEP	Wait for timed period (hrs,mins,secs,ticks)	*	*
IUNTIL	Wait until given time of day (hrs,mins,..)	*	*

References

RT–11 Programmer's Reference Manual. Chapter 2 discusses the .CMKT, .DATE, .GTIM, .MRKT, .QSET, .SDTTM, and .TWAIT programmed requests. Chapter 3 describes the CVTTIM, GTIM, ICMKT, IQSET, ISCHED, ISDTTM, ISLEEP, ITIMER, ITWAIT, IUNTIL, JJCVT, JTIME, MRKT, SECNDS, TIMASC, and TIME subroutines in detail.

RT–11/RSTS/E FORTRAN IV User's Guide. Appendix B covers the DATE and IDATE subroutines.

RT–11 Software Support Manual. Chapter 3 discusses the timer queue element.

PDP–11 FORTRAN Language Reference Manual.

Solutions to Practices

CHAPTER 10

10-1. MACRO-11

```
.EDIT/CREATE PR1001.MAC

.EDIT/CREATE PR1002.MAC

.MACRO PR1001

.LINK PR1001

.RUN PR1001
Enter command: DATE 01-JUN-84

.REENTER
?KMON-F-Invalid command

.EDIT/OUTPUT:PR1003.MAC PR1001.MAC
        .TITLE  PR1003
;
; PR1003  Prompt the user to enter a command, read
;         it, and exit allowing reentry. On REENTER
;         chain to PR1004 passing the input command
;         as data.
;
        .MCALL  .GTLIN  .PRINT  .PEEK   .POKE
        .MCALL  .CHAIN  .EXIT
        .ENABL  LC
AREA:   .BLKW   3               ;EMT argument block
;
```

```
; The data in between PRG2 and the end of MSGBFR
; will be copied into the communication region
; (locations 500 onwards).
;
PRG2:   .RAD50  /DK /           ;File specification
        .RAD50  /PR1004/        ;for the CHAIN request
        .RAD50  /SAV/
MSGBFR: .REPT   41              ;Message buffer
        .WORD   0               ;  (zeroed)
        .ENDR
PROMPT: .ASCII  "Enter command: "<200>
REWIND: .ASCIZ  "REENTER the program"
        .EVEN
        JSW     =       44      ;JOB STATUS WORD
        REENTR  =       20000   ;REENTER BIT (13)
        BR      ENTRY2          ;REENTRY POINT
START:  .GTLIN  #MSGBFR,#PROMPT ;Prompt and place
                                ; input into MSGBFR
        .PRINT  #REWIND         ;Remind user to REENTER
        .PEEK   #AREA,#JSW      ;Get the JSW
        MOV     R0,R1           ;Set REENTER allowed
        BIS     #REENTR,R1      ;  bit in the JSW
        .POKE   #AREA,#JSW,R1
        MOV     #1,R0           ;Exit with REENTER
        .EXIT                   ; enabled
;
; PART 2  When program exits, REENTER it.  Chain
;         to the next program (PR1004).
;
ENTRY2: MOV     #500,R1         ;Load info area addr
        MOV     #PRG2,R2        ;Load data buffer addr
20$:    .POKE   #AREA,R1,(R2)   ;Move command string
        TST     (R1)+           ; into CHAIN area
        TST     (R2)+           ; and advance pointers
        BNE     20$             ;Branch until done
        .CHAIN                  ;CHAIN to next program
        .END    START
```

```
.MACRO PR1003

.LINK PR1003

.MACRO PR1002

.LINK PR1002

.RUN PR1003
Enter command: DATE 01-JUN-83
REENTER the program

.REENTER
HI THERE!  THIS IS PROGRAM 2.
```

```
.EDIT/OUTPUT:PR1004.MAC PR1002.MAC

        .TITLE PR1004
;
; PR1004  Announce that program 2 has started.  See
;          if program was chained to.  If not print
;          an error message and exit.  Otherwise,
;          pass the command from the chain buffer to
;          KMON.
;
        .MCALL  .PRINT  .PEEK   .POKE   .EXIT
        .ENABL  LC
AREA:   .BLKW   3                       ;EMT Argument block
MSGLEN: .WORD   0                       ;Command string length
CMDSTR: .BLKW   41.                     ;Command string buffer
HELLO:  .ASCIZ  "HI THERE!  THIS IS PROGRAM 2."
NOTCH:  .ASCIZ  "** NOT CHAINED TO **"
        .EVEN
        JSW     =       44              ; JOB STATUS WORD
        CHAIN   =       400             ; CHAIN BIT (8)
        KMON    =       4000            ; KMON CMMD BIT (11)
START:  .PRINT  #HELLO                  ;Display message
        .PEEK   #AREA,#JSW              ;Get JSW
        MOV     R0,R3                   ;Save JSW
        BIT     #CHAIN,R3               ;Were we chained to?
        BEQ     STOP                    ;Branch if not (error)
;+
;       We were chained to and the command for KMON is
;       in the CHAIN AREA, but in the wrong format.
;       Retrieve the command from the chain area and
;       restore it inserting the string length as the
;       first word.  Then set JSW bit 11 and R0 to 0
;       to give the command line to KMON on EXIT.
;-
        MOV     #510,R1                 ;Load CHAIN AREA addr
        MOV     #CMDSTR,R2              ;Buffer to receive data
10$:    .PEEK   #AREA,R1                ;Copy data from CHAIN
        TST     (R1)+                   ;  area into receive
        MOV     R0,(R2)+                ;  data buffer
        BNE     10$                     ;Repeat until done
        TST     -(R2)                   ;Step back to string
20$:    TSTB    -(R2)                   ;Was previous byte zero?
        BEQ     20$                     ;Branch if so
        SUB     #CMDSTR-2,R2            ;Calculate & save string
        MOV     R2,MSGLEN               ; length (with null)
        MOV     #510,R1                 ;Load CHAIN AREA address
        MOV     #MSGLEN,R2              ;Load command string addr
30$:    .POKE   #AREA,R1,(R2)           ;Copy data into CHAIN
        TST     (R1)+                   ;  area from buffer
        TST     (R2)+
        BNE     30$                     ;Repeat until done
        BIS     #KMON,R3                ;Set KMON command bit
        .POKE   #AREA,#JSW,R3           ; in JSW
        CLR     R0              ;Clear R0 for exit
        .EXIT                   ;Exit
STOP:   .PRINT  #NOTCH          ;Issue not chained msg
        .EXIT                   ;And exit
        .END    START


.MACRO PR1004

.LINK PR1004

.RUN PR1004
HI THERE!  THIS IS PROGRAM 2.
** NOT CHAINED TO **

.EDIT PR1003.MAC

.MACRO PR1003

.LINK PR1003

.RUN PR1003
Enter command: DATE 11-JUN-84
REENTER the program

.REENTER
HI THERE!  THIS IS PROGRAM 2.

.DATE
11-Jun-83

.RUN PR1003
Enter command: DATE 10-JUN-83
REENTER the program

.REENTER
HI THERE!  THIS IS PROGRAM 2.

.DATE
10-Jun-83
```

10-2. FORTRAN IV

```
.EDIT/CREATE PR1001.FOR

.EDIT/CREATE PR1002.FOR

.FORTRAN PR1001

.LINK PR1001,SY:FORLIB

.RUN PR1001
Enter command: DATE 01-JUN-84

STOP -- END OF PROGRAM

.EDIT PR1001.FOR

.FORTRAN PR1001

.LINK PR1001,SY:FORLIB

.RUN PR1001
Enter command: DATE 01-JUN-84

.EDIT/OUTPUT:PR1003.FOR PR1001.FOR

        PROGRAM PR1003
C
C       Prompt the user to enter a monitor command.
C       Chain to program 2 and pass the command to it.
C       A zero byte is added to the command string read
C       (note that the 80th character may be lost).
C
        BYTE MSGBFR(80)
        REAL*8 FIDBLK            ! FILE DESCRIPTOR BLOCK
        DATA FIDBLK /12RDK PR1004SAV/
        TYPE 100                 ! PROMPT FOR A COMMAND
100     FORMAT (1H$,'Enter command: ')
        ACCEPT 101,ICHARS,MSGBFR ! READ A COMMAND
101     FORMAT  (Q,80A1)
        IF (ICHARS .EQ. 80) ICHARS=79
        MSGBFR(ICHARS+1)=0       ! ADD STRING TERMINATOR
        CALL CHAIN(FIDBLK,MSGBFR,40)
        CALL EX!T                ! EXIT AND CHAIN
        END


.FORTRAN PR1003

.LINK PR1003,SY:FORLIB

.FORTRAN PR1002

.LINK PR1002,SY:FORLIB
```

```
.RUN PR1003
Enter command: DATE 01-JUN-84
HI THERE!  THIS IS PROGRAM 2.

.EDIT/OUTPUT:PR1004.FOR PR1002.FOR

        PROGRAM PR1004
C
C       Announce that program 2 has started.  Request
C       chain information.  If program was chained to
C       by another program setup to have KMON execute
C       the passed command and exit to execute command.
C       Otherwise, issue an error message.
C
        BYTE MSGBFR(80)
        TYPE 100                ! ANNOUNCE PROGRAM
100     FORMAT (1H ,'HI THERE!  THIS IS PROGRAM 2.')
        CALL RCHAIN(ICHAIN,MSGBFR,40)
        IF (ICHAIN .EQ. 0) STOP '** NOT CHAINED TO **'
        CALL SETCMD(MSGBFR)     ! PASS COMMAND TO KMON
        CALL EXIT               ! EXIT
        END


.FORTRAN PR1004

.LINK PR1004,SY:FORLIB

.RUN PR1004
HI THERE!  THIS IS PROGRAM 2.
STOP -- ** NOT CHAINED TO **

.EDIT PR1003.FOR

.FORTRAN PR1003

.LINK PR1003,SY:FORLIB

.RUN PR1003
Enter command: DATE 01-JUN-84
HI THERE!  THIS IS PROGRAM 2.

.DATE
1-Jun-84

.RUN PR1003
Enter command: DATE 10-JUN-84
HI THERE!  THIS IS PROGRAM 2.

.DATE
10-Jun-84
```

CHAPTER 12

12-1. (Step 1) MACRO-11

```
.EDIT/CREATE PR1202.MAC
```

```
        .TITLE  PR1202 -- TERMINAL I/O EXERCISE
        .MCALL  .TTYIN  .TTYOUT .TTINR  .PEEK   .POKE
        .MCALL  .EXIT
;       Data Defintions
AREA:   .BLKW   3               ;EMT argument block
PROMPT: .ASCIZ  "Please type in your name: "
OUT:    .ASCII  "Welcome to RT-11, "
MSGBFR: .BLKB   80.             ;Input buffer
        .EVEN
;       Program Code
START:  MOV     #PROMPT,R1      ;Point to prompt buffer
10$:    .TTYOUT (R1)+           ;Print (w/wait) 1 char
        TSTB    (R1)            ;End of message?
        BNE     10$             ;Loop if not
        MOV     #MSGBFR,R1      ;Point to input buffer
        MOV     #80.,R2         ;Load maximum char count
GET:    .TTYIN  (R1)+           ;Read (w/wait) 1 char
        DEC     R2              ;Decrement char count
        BEQ     OFLO            ;Branch if buffer full
        CMPB    #15,R0          ;Was character a <CR>?
        BNE     GET             ;Branch if not
LINE:   CLRB    -1(R1)          ;Yes, store null byte
        .TTYIN                  ;Get <LF> char
        BR      PRINT           ;Otherwise, print
;       Buffer full.  Flush terminal input buffer
OFLO:   CLRB    -(R1)           ;Append null byte
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R1           ;Move to R1
        BIS     #100,R1         ;Inhibit TT wait
        .POKE   #AREA,#44,R1    ;Update JSW
10$:    .TTINR                  ;Read (wo/wait) 1 char
        BCC     10$             ;Branch if char read
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R1           ;Move to R1
        BIC     #100,R1         ;Enable TT wait
        .POKE   #AREA,#44,R1    ;Update JSW
PRINT:  MOV     #OUT,R1         ;Load buffer address
10$:    .TTYOUT (R1)+           ;Print (w/wait) 1 char
        TSTB    (R1)            ;Is next byte null?
        BNE     10$             ;Branch if not
        .TTYOUT #15             ;Otherwise print <CR>
        .TTYOUT #12             ; and <LF>
        .EXIT                   ;Exit
        .END    START
```

```
.MACRO PR1202

.LINK PR1202

.RUN PR1202
Please type in your name: USER'S NAME
Welcome to RT-11, User's Name
```

12-1. (Step 2) MACRO-11

```
.EDIT/CREATE PR1203.MAC

        .TITLE  PR1203 -- TERMINAL I/O EXERCISE
        .MCALL  .TTYIN  .TTYOUT .TTINR  .PEEK   .POKE
        .MCALL  .EXIT
;       Data Defintions
AREA:   .BLKW   3               ;EMT argument block
PROMPT: .ASCIZ  <15><12>"Enter message: "
MSGBFR: .BLKB   80.             ;Input buffer
        .EVEN
;       Program Code
START:  .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R3           ;Move to R3
        BIC     #10000,R3       ;Enter normal mode input
        .POKE   #AREA,#44,R3    ;Update JSW
        MOV     #PROMPT,R1      ;Point to prompt buffer
10$:    .TTYOUT (R1)+           ;Print (w/wait) 1 char
        TSTB    (R1)            ;End of message?
        BNE     10$             ;Loop if not
        MOV     #MSGBFR,R1      ;Point to input buffer
        MOV     #80.,R2         ;Load maximum char count
GET:    .TTYIN  (R1)+           ;Read (w/wait) 1 char
        DEC     R2              ;Decrement char count
        BEQ     OFLO            ;Branch if buffer full
        CMPB    #15,R0          ;Was char a <CR>?
        BNE     GET             ;Branch if not
        CLRB    -(R1)           ;Yes, store null byte
        .TTYIN                  ;Get <LF> char
        CMP     #MSGBFR,R1      ;Empty line?
        BNE     10$             ;Branch if not
        .EXIT                   ;Otherwise, exit
10$:    MOV     #80.,R2         ;No, set line length = 80.
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R3           ;Move to R3
        BIS     #10000,R3       ;Enter special input mode
        .POKE   #AREA,#44,R3    ;Update JSW
        BR      OUTSET          ;Branch to print cycle
;       Buffer full.  Flush terminal input buffer
OFLO:   .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R3           ;Move to R1
        BIS     #100,R3         ;Inhibit TT wait
        .POKE   #AREA,#44,R3    ;Update JSW
10$:    .TTINR                  ;Read (wo/wait) 1 char
        BCC     10$             ;Branch if char read
        .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R3           ;Move to R1
        BIC     #100,R3         ;Enable TT wait
        .POKE   #AREA,#44,R3    ;Update JSW
        MOV     #80.,R2         ;Set line length = 80.
OUTSET: MOV     #MSGBFR,R1      ;Point to buffer
LOOP:   .PEEK   #AREA,#44       ;Get JSW
        MOV     R0,R3           ;Move to R3
        BIS     #100,R3         ;Inhibit TT wait
        .POKE   #AREA,#44,R3    ;Update JSW
        CLR     R4              ;R4 = 0 for NO INPUT
        .TTINR                  ;Try to read a char
        BCS     NIL             ;Branch if no input
        MOV     R0,R4           ;Save input char
NIL:    .PEEK   #AREA,#44       ;Get the JSW
        MOV     R0,R3           ;Move to R3
        BIC     #100,R3         ;Enable TT wait
        .POKE   #AREA,#44,R3    ;Update JSW
        TST     R4              ;Any input?
        BEQ     NOIN            ;Branch if not
        CMPB    #32,R4          ;Was it ^Z?
        BNE     10$             ;Branch if not
        JMP     START           ;Yes, restart
10$:    CMPB    #15,R4          ;Was it <CR>?
        BEQ     LOOP            ;Branch if so, get <LF>
        CMPB    #12,R4          ;Was it <LF>?
        BNE     NOIN            ;Branch if no, NO INPUT
        JMP     START           ;Yes, restart
NOIN:   .TTYOUT (R1)+           ;Print (w/wait) 1 char
        DEC     R2              ;Decrement counter
        BNE     10$             ;Branch if not at end
        .TTYOUT #15             ;Otherwise print <CR>
        .TTYOUT #12             ; and <LF>
        MOV     #80.,R2         ;Reset counter to 80.
10$:    TSTB    (R1)            ;End of text buffer?
        BEQ     OUTSET          ;Branch if so
        BR      LOOP            ;No, continue
        .END    START
```

```
.EXECUTE PR1203.MAC

Enter message: THIS IS A TEST.
THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS
IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A
TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.
THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS
IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A
TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.
THIS IS A TEST.THIS IS A TEST.THIS IS
Enter message: SECOND LINE.
SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND L
INE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECO
ND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SE
Enter message:
```

12-1. (Step 1) FORTRAN IV

```
.EDIT/CREATE PR1202.FOR

        PROGRAM PR1202
        BYTE PROMPT(80)         ! PROMPT OUTPUT BUFFER
        BYTE MSGBFR(100)        ! INPUT MESSAGE BUFFER
        BYTE MSGOUT(118)        ! OUTPUT MESSAGE BUFFER
        CALL SCOPY('Please type in your name: ',PROMPT)
C
C       Output prompt.
C
        DO 10,I=1,80
        IF (PROMPT(I) .EQ. 0) GO TO 20
5       IF (ITTOUR(PROMPT(I)) .NE. 0) GO TO 5
10      CONTINUE
C
C       Now input user's name.
C
20      DO 40 I=1,100
25      IERR=ITTINR()           ! ACCEPT CHAR
        IF (IERR .LT. 0) GO TO 25 ! LOOP UNTIL READ
        MSGBFR(I)=IERR          ! STORE CHAR IN BUFFER
        IF (MSGBFR(I) .NE. "15) GO TO 40
        MSGBFR(I)=0             ! CHANGE <CR> TO NULL
30      IF (ITTINR() .LT. 0) GO TO 30 ! ACCEPT <LF>
        GO TO 100
40      CONTINUE
C
C       Buffer overflowed (more than 100 chars typed)
C       Read & lose remaining chars in input buffer.
C
60      CALL IPOKE("44,IPEEK("44).OR."100)
65      IF (ITTINR() .GE. 0) GO TO 65
        CALL IPOKE("44,IPEEK("44) .AND. .NOT. "100)
        MSGBFR(100)=0
C
C       Concatenate message with user's name.
C
100     CALL CONCAT('Welcome to RT-11, ',MSGBFR,MSGOUT)
        DO 130 I=1,118
        IF (MSGOUT(I) .EQ. 0) GO TO 150
110     IERR=ITTOUR(MSGOUT(I))  ! OUTPUT A CHAR
        IF (IERR .NE. 0) GO TO 110 ! LOOP UNTIL OUTPUT
130     CONTINUE
150     IF (ITTOUR("15) .NE. 0) GO TO 150 ! OUTPUT <CR>
151     IF (ITTOUR("12) .NE. 0) GO TO 151 !  AND <LF>
        CALL EXIT               ! EXIT
        END


.FORTRAN PR1202

.LINK PR1202,SY:FORLIB
```

```
.RUN PR1202
Please type in your name: <USER'S NAME>
Welcome to RT-11, <User's Name>

.RUN PR1202
Please type in your name: JOHN Q. PUBLIC
Welcome to RT-11, JOHN Q. PUBLIC
```

12-1. (Step 2) FORTRAN IV

```
.EDIT/CREATE PR1203.FOR

        PROGRAM PR1203
C
C       Ask the user to type a message terminated by <CR>.
C       Output the message until the user types Control/Z
C       or <CR>.  Then ask for another message.  Stop if
C       a blank line is entered.  Display 80 characters
C       per line wrapping the rest of the message onto the
C       following line as needed.
C
        BYTE PROMPT(80)         ! OUTPUT BUFFER
        BYTE MSGBFR(80)         ! INPUT BUFFER
        BYTE CRLF(3)
        DATA CRLF /"15,"12,0/
C
C       Start of program, output the prompt.
C
1       CALL IPOKE("44,IPEEK("44) .AND. .NOT. "10000)
        CALL CONCAT(CRLF,'Enter message: ',MSGBFR)
        CALL SCOPY(MSGBFR,PROMPT) ! BUILD PROMPT MESSAGE
        DO 10 I=1,80
        IF (PROMPT(I) .EQ. 0) GO TO 20
5       IF (ITTOUR(PROMPT(I)) .NE. 0) GO TO 5
10      CONTINUE
C
C       Now input a message terminated by a <CR>.
C
20      DO 40 I=1,80
25      IERR=ITTINR()           ! ACCEPT A CHAR
        IF (IERR .LT. 0) GO TO 25 ! LOOP UNTIL READ
        MSGBFR(I)=IERR          ! STORE CHAR IN BUFFER
        IF (MSGBFR(I) .NE. "15) GO TO 40! TEST FOR <CR>
        MSGBFR(I)=0             ! CHANGE <CR> TO NULL
30      IERR=ITTINR()           ! ACCEPT AND LOSE <LF>
        IF (IERR .LT. 0) GO TO 30
        GO TO 100
40      CONTINUE
C
C       Buffer overflow (more than 80 characters).
C       Read & lose remaining chars in input buffer.
C
```

```
60      CALL IPOKE("44,IPEEK("44) .OR. "100)
65      IF (ITTINR() .GE. 0) GO TO 65
        CALL IPOKE("44,IPEEK("44) .AND. .NOT. "100)
        MSGBFR(100)=0           ! MARK END OF BUFFER
C
C       Now ready to output message from MSGBFR.
C
100     IF (MSGBFR(1) .EQ. 0) GO TO 150 ! EXIT IF BLANK
        CALL IPOKE("44,IPEEK("44) .OR. "10000)
        ICHAR=1                 ! POINT TO MESSAGE START
110     DO 130 I=1,80
115     CALL IPOKE("44,IPEEK("44) .OR. "100)
        IERR=ITTINR()
        CALL IPOKE("44,IPEEK("44) .AND. .NOT. "100)
        IF (IERR .LT. 0) GO TO 120
C
C       Deal with character just read.
C
        IF (IERR .EQ. "32) GO TO 1 ! CNTRL/Z, RESTART
        IF (IERR .EQ. "15) GO TO 115 ! <CR>, GET <LF>
        IF (IERR .EQ. "12) GO TO 1 ! <LF>, RESTART
120     IERR=ITTOUR(MSGBFR(ICHAR)) ! OUTPUT NEXT CHAR
        IF (IERR .NE. 0) GO TO 120
        ICHAR=ICHAR+1
        IF (MSGBFR(ICHAR) .EQ. 0) ICHAR=1
130     CONTINUE
131     IF (ITTOUR("15) .NE. 0) GO TO 131
132     IF (ITTOUR("12) .NE. 0) GO TO 132
        GO TO 110
C
C       Exit.
C
150     CALL EXIT
        END
```

```
.EXECUTE PR1203.FOR/LINKLIBRARY:SY:FORLIB

 Enter message: THIS IS A TEST.
THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS
IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A
TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.
THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS
IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A
TEST.THIS IS A TEST.THIS IS A TEST.THIS IS A TEST.THI
Enter message: SECOND LINE.
SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND L
INE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECO
ND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.
SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND L
INE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECOND LINE.SECO
ND LINE.SECOND LINE.SECOND
Enter message:
```

12-2. MACRO-11

```
.EDIT/CREATE PR1204.MAC

        .TITLE  PR1204  -- TERMINAL I/O EXERCISE
        .MCALL  .GTLIN  .PRINT  .TTINR  .PEEK   .POKE
        .MCALL  .RCTRLO .EXIT
;        Data Definitions
AREA:   .BLKW   3                ;EMT argument block
PROMPT: .ASCII  "Enter message: "<200>
MSGBFR: .BLKB   80.              ;Input buffer
        .EVEN
;       Program Code
START:  .PEEK   #AREA,#44        ;Get JSW
        MOV     R0,R1            ;Save value
        BIC     #10000,R1        ;Enter normal input mode
        .POKE   #AREA,#44,R1     ;Update JSW
        .RCTRLC                  ;Reset Control/O
        .GTLIN  #MSGBFR,#PROMPT  ;Prompt & get input line
        TSTB    MSGBFR           ;Null input line?
        BEQ     STOP             ;Branch if so
        .PEEK   #AREA,#44        ;Get JSW
        MOV     R0,R1            ;Save value
        BIS     #10000,R1        ;Enter special input mode
        .POKE   #AREA,#44,R1     ;Update JSW
LOOP:   .PEEK   #AREA,#44        ;Get JSW
        MOV     R0,R1            ;Save value
        BIS     #100,R1          ;Inhibit TT wait
        .POKE   #AREA,#44,R1     ;Update JSW
        CLR     R4               ;Assume no char read
        .TTINR                   ;Read a char
        BCS     10$              ;Branch if none available
        MOV     R0,R4            ;Save char
10$:    .PEEK   #AREA,#44        ;Get JSW
        MOV     R0,R1            ;Save value
        BIC     #100,R1          ;Enable TT wait
        .POKE   #AREA,#44,R1     ;Update JSW
        TST     R4               ;Any input?
        BEQ     NOIN             ;Branch if not
        CMPB    #32,R4           ;Control/Z?
        BEQ     START            ;Branch if so, restart
        CMPB    #15,R4           ;<CR>?
        BEQ     LOOP             ;Branch if so, read <LF>
        CMPB    #12,R4           ;<LF>?
        BEQ     START            ;Branch if so, restart
NOIN:   .PRINT  #MSGBFR          ;Print message
        BR      LOOP             ;Continue
STOP:   .EXIT                    ;Exit
        .END    START
```

```
.EXECUTE PR1204.MAC
Enter message: THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
Enter message: SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
Enter message:
```

12-2. FORTRAN IV

```
.EDIT/CREATE PR1204.FOR

        PROGRAM PR1204
C
C       Ask the user to type a message terminated by <CR>.
C       Output the message until the user types Control/Z
C       or <CR>.  Then ask for another message.  Stop if
C       a blank line is entered.  Display each message on
C       a new line.
C
        BYTE MSGBFR(80)           ! INPUT MESSAGE BUFFER
C
C       Start of program, output the prompt.
C
1       CALL IPOKE("44,IPEEK("44) .AND. .NOT. "10000)
        CALL RCTRLO
        CALL GTLIN(MSGBFR,'Enter message: ')
        IF (MSGBFR(1) .EQ. 0) GO TO 150 ! EXIT IF BLANK
        CALL IPOKE("44,IPEEK("44) .OR. "10000)
C
C       Output message until <CR> or Control/Z typed.
C
115     CALL IPOKE("44,IPEEK("44) .OR. "100)
        IERR=ITTINR()
        CALL IPOKE("44,IPEEK("44) .AND. .NOT. "100)
        IF (IERR .LT. 0) GO TO 120
C
```

```
C       Deal with character just read.
C
        IF (IERR .EQ. "32) GO TO 1 ! CNTRL/Z, RESTART
        IF (IERR .EQ. "15) GO TO 115 ! <CR>, GET <LF>
        IF (IERR .EQ. "12) GO TO 1 ! <LF>, RESTART
120     CALL PRINT(MSGBFR)
        GO TO 115
C
C       Exit.
C
150     CALL EXIT
        END
```

```
.EXECUTE PR1204.FOR/LINKLIBRARY:SY:FORLIB
Enter message:
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
THIS IS A TEST.
Enter message:
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
SECOND LINE.
Enter message:
```

CHAPTER 13

13-1. MACRO-11

```
.EDIT/CREATE PR1301.MAC

        .TITLE  PR1301  -- MULTITERMINAL I/O EXERCISE
        .MCALL  .GTLIN  .MTSTAT .MTATCH .MTGET  .MTSET
        .MCALL  .MTRCTO .MTPRNT .MTIN   .MTDTCH .PRINT
        .MCALL  .EXIT
;       Data Definitions
        .ENABL  LC              ;Enable lower case
WHICH:  .ASCII  "Which terminal do you want to use"
        .ASCII  " (1-7) ? "<200>
ERR1:   .ASCIZ  "** Attach Failure **"
ERR2:   .ASCIZ  "** No Multiterminal Support **"
WHO:    .ASCIZ  "Who are you? " ;Prompt text
REPLY:  .ASCII  "Welcome to Multiterminal RT-11, "
MSGBFR: .BLKB   81.             ;Input buffer
        .EVEN
AREA:   .BLKW   4               ;EMT argument block
STAT:   .BLKW   8.              ;MT status block
TSB:    .BLKW   4               ;Terminal status block
        M.NLUN  =       4       ;Offset to # of terms
;       Program Code
START:  MOV     #STAT,R3        ;Status buffer
        .MTSTAT #AREA,R3        ;Get MT status
        TST     M.NLUN(R3)      ;How many terminals?
        BEQ     NOMTY           ;Branch if none (no MT)
10$:    .GTLIN  #MSGBFR,#WHICH  ;Ask for terminal lun
        TSTB    MSGBFR+1        ;More than one char?
        BNE     10$             ;Branch if so, repeat
        MOVB    MSGBFR,R1       ;Move char into R1
        CMPB    #'0,R1          ;Check for numeric char
        BGT     10$             ;Branch if out of range
        CMPB    #'7,R1          ;Check for numeric char
        BLT     10$             ;Branch if out of range
        BICB    #'0,R1          ;ASCII --> binary
        BEQ     STOP            ;Exit if lun = 0
        .MTATCH #AREA,#0,R1     ;Attach terminal
        BCC     ATT             ;Branch if success
        .PRINT  #ERR1           ;Otherwise, error
        BR      10$             ;And try another
ATT:    .MTGET  #AREA,#TSB,R1   ;Get terminal status
        BIS     #40000,TSB      ;Enable lower case I/O
        .MTSET  #AREA,#TSB,R1   ;Set terminal status
        .MTRCTO #AREA,R1        ;Reset CTRL/O to update
        .MTPRNT #AREA,#WHO,R1   ;Display prompt
        MOV     #MSGBFR,R2      ;Point to input buffer
GET:    .MTIN   #AREA,R2,R1     ;Get a char
        CMPB    #15,(R2)+       ;Is it <CR>?
        BNE     GET             ;Branch if not
        CLRB    -(R2)           ;Yes, store null byte
        .MTPRNT #AREA,#REPLY,R1 ;Print response
        .MTDTCH #AREA,R1        ;Detach terminal
STOP:   .EXIT                   ;Exit
NOMTY:  .PRINT  #ERR2           ;No MT support error
        .EXIT                   ;Exit
        .END    START
```

```
.MACRO PR1301

.LINK PR1301

.SHOW TERMINALS

Unit Owner     Type      WIDTH TAB CRLF FORM SCOPE SPEED
---------------------------------------------------------
  0          Local     DL    80  No  Yes  No    Yes   N/A
  1          Local     DL    80  No  Yes  No    No    N/A
  2          S-Console DL    80  Yes Yes  No    Yes   N/A

.RUN PR1301
Which terminal do you want to use (1-7) ? 2
Who are you?
The user.
Welcome to Multiterminal RT-11, The user.


.RUN PR1301
Which terminal do you want to use (1-7) ? 3
** Attach Failure **
Which terminal do you want to use (1-7) ? 2
Who are you?
Ann
Welcome to Multiterminal RT-11, Ann


.RUN PR1301
Which terminal do you want to use (1-7) ? 7
** Attach Failure **
Which terminal do you want to use (1-7) ? 0
```

13-1. FORTRAN IV

```
.EDIT/CREATE PR1301.FOR

        PROGRAM PR1301
        BYTE PROMPT(120)        ! OUTPUT BUFFER
        BYTE MSGBFR(80)         ! INPUT BUFFER
        BYTE ASCIZ
        INTEGER*2 IMSB(8)       ! MT STATUS BLOCK
        INTEGER*2 ITSB(4)       ! TERMINAL STATUS BLOCK
        DATA ASCIZ/'0'/         ! ASCII ZERO
        CALL IPOKE("44,IPEEK("44) .OR. "40000) ! ENABLE LC
        IERR=MTSTAT(IMSB)       ! GET MT STATUS
        IF (IMSB(3) .EQ. 0) GO TO 110 ! EXIT IF NO MT SUPPORT
C
C       Ask for which terminal the users wishs to attach.
C
        CALL CONCAT('Which terminal do you want to use (1-7) ? ',
     1              "200,PROMPT)
10      CALL GTLIN(MSGBFR,PROMPT)
C
C       Validate input -- user must have typed one character
C       in the range 0 to 7.
C
        IF (MSGBFR(1) .EQ. 0) GO TO 10  ! REPEAT IF BLANK
        IF (MSGBFR(2) .NE. 0) GO TO 10  ! REPEAT IF TOO LONG
        IF (MSGBFR(1) .LT. '0') GO TO 10 ! REPEAT IF INVALID
        IF (MSGBFR(1) .GT. '7') GO TO 10 ! REPEAT IF INVALID
        ILUN=MSGBFR(1)-ASCIZ    ! CONVERT TO BINARY LUN
        IF (ILUN .EQ. 0) GO TO 100 ! EXIT IF LUN 0 SELECTED
C
C       Attach the terminal.
C
        IERR=MTATCH(ILUN,,IJOB)
        IF (IERR .EQ. 0) GO TO 30 ! IF ATTACH SUCCEEDED
        CALL PRINT('** Attach Failure **')
        GO TO 10                ! TRY AGAIN
C
C       Get terminal status block for attached terminal
C       and enable lower case input for terminal.
C
30      IERR=MTGET(ILUN,ITSB)   ! GET TERMINAL STATUS
        ITSB(1)=ITSB(1) .OR. "40000 ! ENABLE LOWER CASE
        IERR=MTSET(ILUN,ITSB)   ! WRITE NEW STATUS
        IERR=MTRCTO(ILUN)       ! FORCE UPDATE OF STATUS
C
C       Ask user to type his name.
C
        CALL MTPRNT(ILUN,'Who are you? ')
C
C       And read reply terminated by <CR>.
C
        DO 60 I=1,80
        IERR=MTIN(ILUN,MSGBFR(I)) ! GET A CHARACTER
        IF (MSGBFR(I) .EQ. "15) GO TO 70 ! DONE IF <CR>
60      CONTINUE
C
C       If buffer overflowed, clear last byte.
C
```

```
70      MSGBFR(I)=0              ! REPLACE <CR> WITH NULL
C
C       Output the message.
C
        CALL CONCAT('Welcome to Multiterminal RT-11, ',
     1               MSGBFR,PROMPT)
        CALL MTPRNT(ILUN,PROMPT)
C
C       Now detach terminal and exit.
C
        IERR=MTDTCH(ILUN)
100     CALL EXIT
C
C       Handle error if no MT support available.
C
110     CALL PRINT('** No Multiterminal Support **')
        CALL EXIT
        END
```

```
.SHOW TERMINALS

Unit Owner     Type      WIDTH TAB CRLF FORM SCOPE SPEED
--------------------------------------------------------
  0          Local     DL    80  No  Yes  No    Yes   N/A
  1          Local     DL    80  No  Yes  No    No    N/A
  2          S-Console DL    80  Yes Yes  No    Yes   N/A
```

```
.EXECUTE/LINKLIBRARY:SY:FORLIB PR1301.FOR
Which terminal do you want to use (1-7) ? 7
** Attach Failure **
Which terminal do you want to use (1-7) ? 0

.RUN PR1301
Which terminal do you want to use (1-7) ? 2
Who are you?
Ann
Welcome to Multiterminal RT-11, Ann
```

CHAPTER 14

14-1. MACRO-11

```
.EDIT/CREATE PR1401.MAC

.MACRO PR1401

.LINK PR1401

.COPY TT: TRAN1.XYZ
 Files copied:
^TEST DATA FILE WHICH WILL BE COPIED TO TRAN2.XYZ

*** EOF ***
^ZTT:            to DK:TRAN1.XYZ

.TYPE TRAN2.XYZ
?PIP-F-File not found DK:TRAN2.XYZ

.RUN PR1401
Program copies TRAN1.XYZ to TRAN2.XYZ

.TYPE TRAN2.XYZ
TEST DATA FILE WHICH WILL BE COPIED TO TRAN2.XYZ

*** EOF ***

.DEL TRAN1.XYZ

.RUN PR1401
Program copies TRAN1.XYZ to TRAN2.XYZ
Error on LOOKUP of input file

.COPY PR1401.MAC PR1801.MAC

.EDIT PR1401.MAC
ANNCE:  .ASCIZ  "Program copies PR1401.MAC to 123456.TMP"

INFILE: .RAD50  /DK PR1401MAC/  ;Copy from DK:PR1401.MAC
OUTFIL: .RAD50  /DK 123456TMP/  ; to DK:123456.TMP

.EXECUTE PR1401.MAC
Program copies PR1401.MAC to 123456.TMP

.DIFF PR1401.MAC 123456.TMP
?SRCCOM-I-No differences found

.EDIT PR1401.MAC
BUFFER: .BLKW   512.            ;File I/O Buffer

1$:     .READW  #EMTARG,#3,#BUFFER,#512.,R1

2$:     .WRITW  #EMTARG,#0,#BUFFER,#512.,R1
```

```
        BCS     WERR                ;Branch on write failure
        ADD     #2,R1               ;Update block number
        BR      1$                  ;And read next block

.DELETE 123456.TMP

.EXECUTE PR1401.MAC
Program copies PR1401.MAC to 123456.TMP

.DIFF PR1401.MAC 123456.TMP
?SRCCOM-I-No differences found
```

14-1. FORTRAN IV

```
.EDIT/CREATE PR1402.FOR

.COPY PR1402.FOR TRAN1.XYZ

.DEL TRAN2.XYZ

.EXECUTE/LINKLIBRARY:SY:FORLIB PR1402.FOR
Program copies TRAN1.XYZ to TRAN2.XYZ

.DIFF TRAN1.XYZ TRAN2.XYZ
?SRCCOM-I-No differences found

.DELETE TRAN1.XYZ

.RUN PR1402
Program copies TRAN1.XYZ to TRAN2.XYZ
Error on LOOKUP of input file

.COPY PR1402.FOR PR1802.FOR

.EDIT PR1402.FOR
        DATA INFILE/2RDK,3RPR1,3R402,3RFOR/
        DATA OUTFIL/2RDK,3R123,3R456,3RTMP/

        CALL PRINT('Program copies PR1402.FOR to 123456.TMP')

.EXECUTE/LINKLIBRARY:SY:FORLIB PR1402.FOR
Program copies PR1402.FOR to 123456.TMP

.DIFF PR1402.FOR 123456.TMP
?SRCCOM-I-No differences found

.EDIT PR1402.FOR
        INTEGER*2 BUFFER(512),BLOCK

20      IERR = IREADW(512,BUFFER,BLOCK,INCHN)
```

```
30      IF (IWRITW(512,BUFFER,BLOCK,OUTCHN) .LT. 0)
     1    GO TO 101
        BLOCK = BLOCK+2             ! Update to block
        GO TO 20                    ! Read next block

.DELETE 123456.TMP

.EXECUTE/LINKLIBRARY:SY:FORLIB PR1402.FOR
Program copies PR1402.FOR to 123456.TMP

.DIFF PR1402.FOR 123456.TMP
?SRCCOM-I-No differences found
```

CHAPTER 15

15-1. MACRO-11

```
.EDIT/CREATE PR1505.MAC

        .TITLE  PR1505
;
;       This program uses asynchronous double buffered
;       computation.  The program reads data from an
;       input file, performs computation on that data,
;       and writes the transformed data to the output
;       file.
;
        .MCALL  .EXIT   .FETCH  .LOOKUP .ENTER  .PRINT
        .MCALL  .READ   .WRITE  .CLOSE  .SRESET .WAIT
        .MCALL  .QSET
EMTARG: .BLKW   6               ;EMT argument block
INFILE: .RAD50  /DK TRAN1 XYZ/  ;Copy from DK:TRAN1.XYZ
OUTFIL: .RAD50  /DK TRAN2 XYZ/  ;  to DK:TRAN2.XYZ
LIMITS: .LIMIT                  ;High/low program limits
BUFF1:  .BLKW   256.            ;File I/O Buffer 1
BUFF2:  .BLKW   256.            ;File I/O Buffer 2
QELMT:  .BLKW   10.             ;Queue element
BIGEST: .BLKW   1               ;Stores largest value
ERROR:  .BYTE                   ;Error status byte
EOF:    .BYTE                   ;End of File flag
ANNCE:  .ASCIZ  "Program copies TRAN1.XYZ to TRAN2.XYZ"
FCH1MS: .ASCIZ  "Error on FETCH of output handler"
FCH2MS: .ASCIZ  "Error on FETCH of input handler"
LKPMES: .ASCIZ  "Error on LOOKUP of input file"
ENTMES: .ASCIZ  "Error on creation of output file"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
```

```
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine sets up files for I/O.
;       The file specifications are fixed.
;
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     R1,-(SP)        ;Save register
        .PRINT  #ANNCE          ;Announce program
;       Fetch device handlers
        MOV     LIMITS+2,R1     ;Load high limit
        .FETCH  R1,#OUTFIL      ;Get output handler
        BCS     FCH1ER          ;Branch on FETCH error
        MOV     R0,R1           ;Load high limit
        .FETCH  R0,#INFILE      ;Get input handler
        BCS     FCH2ER          ;Branch on FETCH error
;       Open files
        .LOOKUP #EMTARG,#3,#INFILE
        BCS     LKPERR          ;Branch if open failed
        MOV     R0,R1           ;Load input file length
        .ENTER  #EMTARG,#0,#OUTFIL
        BCC     DONE            ;Branch if successful
;       Error Routines
        .PRINT  #ENTMES         ;Issue create failure msg
        BR      ERDONE          ;And finish up
LKPERR: .PRINT  #LKPMES         ;Issue open failure msg
        BR      ERDONE          ;And finish up
FCH2ER: .PRINT  #FCH2MS         ;Issue FETCH error
        BR      ERDONE          ;And finish up
FCH1ER: .PRINT  #FCH1MS         ;Issue FETCH error
ERDONE: SEC                     ;Indicate error occurred
DONE:   MOV     (SP)+,R1        ;Restore R1 (save C-bit)
        RETURN                  ;Return to caller
        .SBTTL  CMPRTN  -- Computation Routine
;
;       Routine assumes the input file is opened on
;       channel 3 and the output on channel 0.
;       Returns with C-BIT SET on error.
;
;       Note: All registers except R0 are preserved.
;
CMPRTN: MOV     R1,-(SP)        ;Save registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        MOV     R4,-(SP)
        MOV     R5,-(SP)
        .QSET   #QELMT,#1       ;Allocate a queue element
BEG:    .READ   #EMTARG,#3,#BUFF1,#256.,#0
        BCC     INIT            ;Branch if read ok
        TSTB    @#52            ;End-of-File?
        BEQ     EXIT            ;Branch if so, all done
        BR      RDERR           ;Issue read error
```

```
INIT:   MOV     #1,R1               ;Load read block number
        CLR     R5                  ;Load write block number
        MOV     #BUFF2,R2           ;R2 ==> input buffer
        MOV     #BUFF1,R3           ;R3 ==> output buffer
        CLRB    ERROR               ;Clear error flag
        CLRB    EOF                 ;Clear EOF flag
SLOOP:  .WAIT   #3                  ;Wait for input to finish
        BCS     RDERR               ;Branch on error
        .READ   #EMTARG,#3,R2,#256.,R1
        BCC     COMP                ;Branch if read succeeded
        TSTB    @#52                ;End-Of-File (EOF)?
        BNE     RDERR               ;Branch if fatal error
        INCB    EOF                 ;Set EOF encountered flag
;
;       Perform computation on previously read block.
;       This routine finds the largest value within the
;       block and subtracts each word in the block from
;       that value.
;
COMP:   MOV     #255.,R0            ;Initialize counter
        MOV     R3,R4               ;Load buffer starting address
        MOV     (R4)+,BIGEST        ;Assume 1st word is biggest
10$:    CMP     (R4)+,BIGEST        ;Next word bigger?
        BLT     20$                 ;Branch if not
        MOV     -2(R4),BIGEST       ;Otherwise, save new biggest
20$:    DEC     R0                  ;Done?
        BNE     10$                 ;Branch if not
        MOV     #256.,R0            ;Initialize counter
        MOV     R3,R4               ;Load buffer starting address
30$:    MOV     BIGEST,-(SP)        ;Put biggest value onto stack
        SUB     (R4),(SP)           ;Stack = biggest - current
        MOV     (SP)+,(R4)+         ;Save result
        DEC     R0                  ;Done?
        BNE     30$                 ;Branch if not
;
;       Write out buffer on which computation has just
;       been performed.
;
        .WRITE  #EMTARG,#0,R3,#256.,R5
        BCS     WERR                ;Branch on write error
        TSTB    EOF                 ;EOF on last read?
        BGT     EXIT                ;Branch if so
        MOV     R2,R4               ;Otherwise, switch input
        MOV     R3,R2               ;  and output buffer
        MOV     R4,R3
        INC     R1                  ;Update input block #
        INC     R5                  ;Update output block #
        BR      SLOOP               ;And continue
;
;       Error messages and cleanup.
;
RDERR:  .PRINT  #RERRMS             ;Issue read error msg
        BR      EREXIT              ;And finish up
WERR:   .PRINT  #WERRMS             ;Issue write error msg
```

```
EREXIT: DECB    ERROR           ;Set error flag
EXIT:   MOV     (SP)+,R5        ;Restore saved registers
        MOV     (SP)+,R4
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        .WAIT   #0              ;Wait for last output
        BCC     1$              ;Branch if successful
        .PRINT  #WERRMS         ;Issue write error msg
        BR      2$
1$:     TSTB    ERROR           ;Set C-Bit? (clear it)
        BEQ     3$              ;Branch if not
2$:     SEC                     ;Otherwise, set it
3$:     RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Cleanup For Copy Program
CLSCHN: .CLOSE  #3              ;Close input file
        .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch on success
        .PRINT  #PRTCT          ;Issue protected file msg
PRGCHN:                         ;Purge files
RESET:  .SRESET                 ;Reset (purge files)
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Open files
        BCS     1$              ;Branch on error
        CALL    CMPRTN          ;Transfer the file
        BCS     1$              ;Branch on error
        CALL    CLSCHN          ;Close the files
        BR      2$
1$:     CALL    PRGCHN          ;Purge the files
2$:     .EXIT
        .END    START
```

```
.BAS
BASIC-11/RT-11 V02-03
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A

READY
NEW BUILD
100 OPEN "DK:TRAN1.XYZ" FOR OUTPUT AS FILE #1%
110 DIM #1%,A%(511%)
120 FOR Y%=0% TO 511%
130 A%(Y%)=Y%
140 NEXT Y%
150 CLOSE
160 END
RUN

BUILD     20-MAR-84  10:59:10

READY
BYE
```

```
.DUMP/TERMINAL TRAN1.XYZ/NOASCII
DK:TRAN1.XYZ
BLOCK NUMBER  000000
000/ 000000 000001 000002 000003 000004 000005 000006 000007
020/ 000010 000011 000012 000013 000014 000015 000016 000017
040/ 000020 000021 000022 000023 000024 000025 000026 000027
060/ 000030 000031 000032 000033 000034 000035 000036 000037
100/ 000040 000041 000042 000043 000044 000045 000046 000047
120/ 000050 000051 000052 000053 000054 000055 000056 000057
140/ 000060 000061 000062 000063 000064 000065 000066 000067

                    .
                    .
                    .

620/ 000710 000711 000712 000713 000714 000715 000716 000717
640/ 000720 000721 000722 000723 000724 000725 000726 000727
660/ 000730 000731 000732 000733 000734 000735 000736 000737
700/ 000740 000741 000742 000743 000744 000745 000746 000747
720/ 000750 000751 000752 000753 000754 000755 000756 000757
740/ 000760 000761 000762 000763 000764 000765 000766 000767
760/ 000770 000771 000772 000773 000774 000775 000776 000777

.EXECUTE PR1505.MAC
Program copies TRAN1.XYZ to TRAN2.XYZ

.DUMP/TERMINAL TRAN2.XYZ/NOASCII
DK:TRAN2.XYZ
BLOCK NUMBER  000000
000/ 000377 000376 000375 000374 000373 000372 000371 000370
020/ 000367 000366 000365 000364 000363 000362 000361 000360
040/ 000357 000356 000355 000354 000353 000352 000351 000350
060/ 000347 000346 000345 000344 000343 000342 000341 000340
100/ 000337 000336 000335 000334 000333 000332 000331 000330
120/ 000327 000326 000325 000324 000323 000322 000321 000320
140/ 000317 000316 000315 000314 000313 000312 000311 000310

                    .
                    .
                    .

620/ 000067 000066 000065 000064 000063 000062 000061 000060
640/ 000057 000056 000055 000054 000053 000052 000051 000050
660/ 000047 000046 000045 000044 000043 000042 000041 000040
700/ 000037 000036 000035 000034 000033 000032 000031 000030
720/ 000027 000026 000025 000024 000023 000022 000021 000020
740/ 000017 000016 000015 000014 000013 000012 000011 000010
760/ 000007 000006 000005 000004 000003 000002 000001 000000
```

15-1. FORTRAN IV

```
.EDIT/CREATE PR1506.FOR

        PROGRAM PR1506
C
C       This program reads data from the input file,
C       performs computation on the data read, and
C       writes the transformed data to the output file.
C       This program uses asynchronous I/O to allow the
C       computation to occur during I/O operations.
C
        LOGICAL*1 SETUP,CMPRTN
        LOGICAL*1 ERROR
C
        ERROR = SETUP()         ! Open files
        IF (ERROR) GO TO 20     ! Stop on setup error
        ERROR = CMPRTN()        ! Copy file
        IF (ERROR) GO TO 20     ! Stop on error
        CALL CLSCHN             ! Close channels
        GO TO 30                ! Exit
20      CALL PRGCHN             ! Purge channels
30      CALL EXIT
        END
        FUNCTION SETUP
C
C       This routine sets up the files for I/O.
C       The file specifications are fixed in the version.
C
C       Function returns .TRUE. if an error occurred.
C
        LOGICAL*1 SETUP
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
C
C       Channel numbers in common because they are used
C       by CMPRTN, CLSCHN, and PRGCHN.
C
        INTEGER*2 INFILE(4),OUTFIL(4)
        DATA INFILE/2RDK,3RTRA,2RN1,3RXYZ/ ! Input & output
        DATA OUTFIL/2RDK,3RTRA,2RN2,3RXYZ/ !  file specs
C
C       Output introductory message and allocate channels.
C
        CALL PRINT('Program copies TRAN1.XYZ to TRAN2.XYZ')
        INCHN = IGETC()
        OUTCHN = IGETC()
C
C       Fetch needed device handlers.
C
        IF (IFETCH(OUTFIL(1)) .NE. 0) GO TO 101
        IF (IFETCH(INFILE(1)) .NE. 0) GO TO 102
C
C       Open input file.
C
```

```
        LENGTH = LOOKUP(INCHN,INFILE)
        IF (LENGTH .LT. 0) GO TO 103
C
C       Create output file.
C
        IF (IENTER(OUTCHN,OUTFIL,LENGTH) .LT. 0) GO TO 104
        SETUP = .FALSE.         ! Return success
        RETURN
C
C       ERROR ROUTINES
C
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.          ! Return error
        RETURN
        END
        FUNCTION CMPRTN
C
C       Double-buffered computation routine.
C
C       Function returns .TRUE. on error.
C       NOTE: Some severe errors will abort the program.
C
        LOGICAL*1 CMPRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFFER(256,2),BLOCK,INPTR,OUTPTR
        LOGICAL*1 FRSTTM        ! Once only flag
        DATA FRSTTM/.TRUE./
        IF (.NOT. FRSTTM) GO TO 10 ! Do QSET only once
        IF (IQSET(1) .NE. 0) STOP 'No room for queue element'
        FRSTTM = .FALSE.
C
C       Begin by reading into buffer 1.
C
10      IERR = IREAD(256,BUFFER(1,1),0,INCHN)
        IF (IERR .GE. 0) GO TO 20 ! Successful read
        IF (IERR .EQ. (-1)) GO TO 150 ! EOF means all done
        GO TO 100               ! Read error
C
C       Initialize block numbers and flags.
C
20      BLOCK = 1               ! Initialize block number &
        INPTR = 2               !  input buffer number &
        OUTPTR = 1              !  output buffer number
C
C       Loop: Wait for input to complete, compute,
C             output.
C
```

```
30      IF (IWAIT(INCHN) .NE. 0) GO TO 100 ! Error on read
C
C       Read next block into input buffer.
C
        IERR = IREAD(256,BUFFER(1,INPTR),BLOCK,INCHN)
        IF (IERR .LT. (-1)) GO TO 100! Error on read
C
C       Perform computation on output buffer while read is
C       in progress. Computation consists of subtracting
C       each word in the block from the largest value in
C       that block.
C
50      IBIG = BUFFER(1,OUTPTR)
        DO 60 I=2,256
        IF (BUFFER(I,OUTPTR) .GT. IBIG)
     1     IBIG=BUFFER(I,OUTPTR)
60      CONTINUE
        DO 70 I=1,256
        BUFFER(I,OUTPTR) = IBIG-BUFFER(I,OUTPTR)
70      CONTINUE
C
C       Write out buffer on which computations have just
C       completed.
C
        IF (IWRITE(256,BUFFER(1,OUTPTR),BLOCK-1,OUTCHN) .LT. 0)
     1     GO TO 101            ! Error on write
C
C       Check if last read resulted in EOF.
C
        IF (IERR .EQ. (-1)) GO TO 150   ! Copy completed.
C
C       Otherwise, switch buffers and advance block number.
C
        ITMP = INPTR
        INPTR = OUTPTR
        OUTPTR = ITMP
        BLOCK = BLOCK+1
        GO TO 30                        ! Repeat
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
101     CALL PRINT('Write error, copy aborted')
140     CMPRTN = .TRUE.
        RETURN
C
C       Wait for last output to complete and return.
C
150     IF (IWAIT(OUTCHN) .NE. 0) GO TO 101
        CMPRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        RETURN
        END
```

```
.BAS
BASIC-11/RT-11 V02-03
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A

READY
NEW BUILD
100 OPEN "DK:TRAN1.XYZ" FOR OUTPUT AS FILE #1%
110 DIM #1%,A%(511%)
120 FOR Y%=0% TO 511%
130 A%(Y%)=Y%
140 NEXT Y%
150 CLOSE
160 END
RUN

BUILD     20-MAR-84  11:34:15

READY
BYE

.DUMP/TERMINAL TRAN1.XYZ/NOASCII
DK:TRAN1.XYZ
BLOCK NUMBER  000000
000/ 000000 000001 000002 000003 000004 000005 000006 000007
020/ 000010 000011 000012 000013 000014 000015 000016 000017
040/ 000020 000021 000022 000023 000024 000025 000026 000027
060/ 000030 000031 000032 000033 000034 000035 000036 000037
100/ 000040 000041 000042 000043 000044 000045 000046 000047
120/ 000050 000051 000052 000053 000054 000055 000056 000057
140/ 000060 000061 000062 000063 000064 000065 000066 000067
```

```
                   .
                   .
                   .

620/ 000710 000711 000712 000713 000714 000715 000716 000717
640/ 000720 000721 000722 000723 000724 000725 000726 000727
660/ 000730 000731 000732 000733 000734 000735 000736 000737
700/ 000740 000741 000742 000743 000744 000745 000746 000747
720/ 000750 000751 000752 000753 000754 000755 000756 000757
740/ 000760 000761 000762 000763 000764 000765 000766 000767
760/ 000770 000771 000772 000773 000774 000775 000776 000777

.EXECUTE PR1506/LINKLIBRARY:SY:FORLIB
Program copies TRAN1.XYZ to TRAN2.XYZ

.DUMP/TERMINAL TRAN2.XYZ/NOASCII
DK:TRAN2.XYZ
BLOCK NUMBER  000000
000/ 000377 000376 000375 000374 000373 000372 000371 000370
020/ 000367 000366 000365 000364 000363 000362 000361 000360
040/ 000357 000356 000355 000354 000353 000352 000351 000350
060/ 000347 000346 000345 000344 000343 000342 000341 000340
100/ 000337 000336 000335 000334 000333 000332 000331 000330
120/ 000327 000326 000325 000324 000323 000322 000321 000320
140/ 000317 000316 000315 000314 000313 000312 000311 000310

                   .
                   .
                   .

620/ 000067 000066 000065 000064 000063 000062 000061 000060
640/ 000057 000056 000055 000054 000053 000052 000051 000050
660/ 000047 000046 000045 000044 000043 000042 000041 000040
700/ 000037 000036 000035 000034 000033 000032 000031 000030
720/ 000027 000026 000025 000024 000023 000022 000021 000020
740/ 000017 000016 000015 000014 000013 000012 000011 000010
760/ 000007 000006 000005 000004 000003 000002 000001 000000

.
```

15–2.

(1) Event-driven
(2) Asynchronous
(3) Synchronous

CHAPTER 17

17-1. (Step 1) MACRO-11

```
.EDIT/CREATE PR1701.MAC

        .TITLE  PR1701  Solution 17-1 Background
        .MCALL  .QSET   .SDATW  .RCVDW  .PRINT  .EXIT
        .NLIST  BEX
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
INBUF:  .BLKW   42.             ;Input buffer
SNDBF:  .BLKW   42.
SNDERR: .ASCIZ  "?PR1701 Send error, no other job"
RCVER1: .ASCIZ  "?PR1701 Receive error, no other job"
        .EVEN
START:  .QSET   #QLIST,#2       ;Allocate queue elements
        MOV     #RCVER1,R2      ;Assume no other job
        .RCVDW  #AREA,#INBUF,#41. ;Wait for message
        BCS     ERR             ;Report failure
        MOV     #INBUF+2,R2     ;Point to received data
        MOV     #SNDBF,R3       ;Point to send buffer
10$:    TSTB    (R2)+           ;Search for null byte
        BNE     10              ;Loop until byte found
        DEC     R2              ;Backup over null byte
20$:    MOVB    -(R2),(R3)+     ;Move string to output
        CMP     R2,#INBUF+2     ;All done?
        BHI     20              ;Loop until done if not
        CLRB    (R3)            ;Insert null byte
        MOV     INBUF,R1        ;Load word count
        MOV     #SNDERR,R2      ;Assume send error
        .SDATW  #AREA,#SNDBF,R1 ;Send message and wait
        BCS     ERR             ;Report failure
        .EXIT                   ;Otherwise, exit
ERR:    .PRINT  R2              ;Print error message
        .EXIT                   ;And exit
        .END    START
```

```
.EDIT/CREATE PR1702.MAC

        .TITLE  PR1702  Solution 17-1 Foreground
        .MCALL  .PEEK   .POKE   .QSET   .GTLIN
        .MCALL  .SDATW  .RCVDW  .PRINT  .EXIT
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
INBUF:  .BLKW   42.             ;Input buffer
PROMPT: .ASCIZ  "Input your message: "
REPLY:  .ASCIZ  "In reverse, that becomes:"
SNDERR: .ASCIZ  "?PR1702 Send error, no other job"
RCVER1: .ASCIZ  "?PR1702 Receive error, no other job"
RCVER2: .ASCIZ  "?PR1702 Receive error, length error"
```

```
        .EVEN
        JSW     =       44
        LOWER   =       40000
START:  .QSET   #QLIST,#2         ;Allocate queue elements
        .PEEK   #AREA,#JSW        ;Get JSW
        MOV     R0,R1             ;Save the value
        BIS     #LOWER,R1         ;Enable lower case input
        .POKE   #AREA,#JSW,R1     ;Set JSW
        .GTLIN  #INBUF,#PROMPT    ;Get input from console
        MOV     #INBUF,R1         ;Load string buffer addr
10$:    TSTB    (R1)+             ;Search for null byte
        BNE     10$               ;Loop until byte found
        SUB     #INBUF,R1         ;Calculate string length
        INC     R1                ;Round up to next word
        ASR     R1                ;Calculate word count
        MOV     #SNDERR,R2        ;Assume no other job
        .SDATW  #AREA,#INBUF,R1   ;Send message to backgrd
        BCS     ERR               ;Report error
        MOV     #RCVER1,R2        ;Assume receive error
        .RCVDW  #AREA,#INBUF,R1   ;Wait for reply
        BCS     ERR               ;Report error
        MOV     #RCVER2,R2        ;Assume invalid length
        CMP     R1,INBUF          ;Check received length
        BNE     ERR               ;Report error
        .PRINT  #REPLY            ;Print heading
        .PRINT  #INBUF+2          ;Print reply
        .EXIT                     ;Exit
ERR:    .PRINT  R2                ;Print error message
        .EXIT                     ;And exit
        .END    START
```

```
.MACRO PR1701,PR1702

.LINK PR1701

.LINK/FOREGROUND PR1702

.RUN PR1701
?PR1701 Receive error, no other job

.FRUN PR1702

F>
Input your message:

B>

.RUN PR1701

F>
This is the message for the foreground job.
In reverse, that becomes:
.boj dnuorgerof eht rof egassem eht si sihT

B>
```

17-1. (Step 1) FORTRAN IV

```
        PROGRAM PR1701
C
C       Solution to 17-1 Background Job
C
C       Receive string from foreground, reverse it,
C       and return it.
C
        BYTE STR1(84), STR2(84), STRING(82)
        INTEGER*2 ICOUNT
        EQUIVALENCE (ICOUNT,STR1),(STRING,STR1(3))
C
C       Allocate queue elements.
C
        IF (IQSET(2) .NE. 0)
     1     STOP '?PR1701 Insufficient queue elements'
C
C       Receive string from foreground job.
C
        IF (IRCVDW(STR1,41) .NE. 0)
     1     STOP '?PR1701 Receive error'
        L=LEN(STRING)           ! Load string length
        DO 10 I=1,L
10      STR2(I)=STRING(L-I+1)   ! Reverse string
        STR2(L+1)=0             ! Append null byte
        IL=ICOUNT               ! Save word count
C
C       Send reversed string to foreground job.
C
        IF (ISDATW(STR2,IL) .NE. 0)
     1     STOP '?PR1701 Send error'
        CALL EXIT
        END
```

```
        PROGRAM PR1702
C
C       Solution to 17-1 Foreground Job
C
C       Read string, send it to background job, receive
C       string from background job, and print it.
C
        BYTE STR1(84), STR2(84), PROMPT(80)
        INTEGER*2 ICOUNT
        EQUIVALENCE (ICOUNT,STR2)
C
C       Allocate queue elements.
C
        IF (IQSET(2) .NE. 0)
      1    STOP '?PR1702 Insufficient queue elements'
C
C       Prompt for string and get it.
C
        CALL IPOKE("44,"40000 .OR. IPEEK("44))
      CALL SCOPY('Input your message:',PROMPT)
      CALL GTLIN(STR1,PROMPT)
      L=(LEN(STR1)+2)/2 ! Length in words
C
C     Send string to background job.
C
      IF (ISDATW(STR1,L) .NE. 0)
     1   STOP '?PR1702 Send error'
C
C     Receive reversed string and print it.
C
      IF (IRCVDW(STR2,L) .NE. 0)
     1   STOP '?PR1702 Receive error'
      IF (ICOUNT .NE. L)
     1   STOP '?PR1702 Receive length error'
      CALL PRINT('In reverse, that becomes:')
      CALL PRINT(STR2(3))
      CALL EXIT
      END
```

17-1. (Step 2) FORTRAN IV

```
.EDIT/CREATE PR1703.FOR

      PROGRAM PR1703
C
C     Solution to 17-1 (Step 2) Background Job
C
C     Receive shared buffer from foreground,
C     reverse string, and return it.
C
      EXTERNAL REV
      BYTE STR2(84)
      INTEGER*2 MSG(3)
C
C     Allocate queue elements.
C
      IF (IQSET(2) .NE. 0)
     1   STOP '?PR1703 Insufficient queue elements'
C
C     Receive string from foreground job.
C
      IF (IRCVDW(MSG,2) .NE. 0)
     1   STOP '?PR1703 Receive error'
      CALL INDIR(REV,1,MSG(2),0,STR2) ! Reverse string
C
C     Send reversed string.
C
      IF (ISDATW(MSG(2),1) .NE. 0)
     1   STOP '?PR1703 Send error'
      CALL EXIT
      END
      SUBROUTINE REV (ISTR,OSTR)
C
C     Subroutine to reverse string.
C
      BYTE ISTR(84),OSTR(84)
      INTEGER*2 L
      L=LEN(ISTR)             ! Get length of string
      DO 10 I=1,L
10    OSTR(I)=ISTR(L-I+1)     ! Reverse string
      DO 20 I=1,L
20    ISTR(I)=OSTR(I)         ! Move back to buffer
      RETURN
      END


.EDIT/CREATE PR1704.FOR

      PROGRAM PR1704
C
C     Solution to 17-1 (Step 2) Foreground Job
C
C     Read string and share buffer with background,
C     wait for returned string, and print results.
C
      BYTE STR1(84), PROMPT(80)
      INTEGER*2 ICOUNT,MSG(2)
      EQUIVALENCE (ICOUNT,STR1)
C
C     Allocate queue elements.
C
      IF (IQSET(2) .NE. 0)
     1   STOP '?PR1704 Insufficient queue elements'
C
C     Prompt for string and get it.
C
      CALL IPOKE("44,"40000 .OR. IPEEK("44))
      CALL SCOPY('Input your message:',PROMPT)
      CALL GTLIN(STR1,PROMPT)
      L=LEN(STR1)             ! Get string length
      MSG(1)=IADDR(STR1)      ! Build shared
      MSG(2)=IADDR(STR1(L))   !  buffer descriptor
C
C     Send data with shared buffer descriptor to
C     the background job.
C
      IF (ISDATW(MSG,2) .NE. 0)
     1   STOP '?PR1704 Send error'
C
C     Receive reversal completed from background.
C
      IF (IRCVDW(MSG,1) .NE. 0)
     1   STOP '?PR1704 Receive error'
      IF (MSG(1) .NE. 1)
     1   STOP '?PR1704 Receive length error'
```

```
        CALL PRINT('In reverse, that becomes:')
        CALL PRINT(STR1)
        CALL EXIT
        END

.EDIT/CREATE INDIR.MAC

.FORTRAN PR1703,PR1704

.MACRO INDIR

.LINK PR1703,INDIR,SY:FORLIB

.LINK/FOREGROUND PR1704,SY:FORLIB

.FRUN PR1704

F>
?Err 62 FORTRAN start fail

B>

.FRUN PR1704/BUFFER:2000

.
F>
Input your message:

B>


.RUN PR1703

F>
This is the message for PR1703.
In reverse, that becomes:
.3071RP rof egassem eht si sihT

B>
```

17-1. (Step 2) MACRO-11

```
        .TITLE  PR1703  Solution 17-1 (Step 2) Background
        .MCALL  .QSET   .SDATW  .RCVDW  .PRINT  .EXIT
        .NLIST  BEX
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
MSG:    .BLKW   3               ;Shared buffer data
TMPBF:  .BLKW   42.             ;Work area
SNDERR: .ASCIZ  "?PR1703 Send error, no other job"
RCVER1: .ASCIZ  "?PR1703 Receive error, no other job"
RCVER2: .ASCIZ  "?PR1703 Receive error, data length"
        .EVEN
START:  .QSET   #QLIST,#2       ;Allocate queue elements
        MOV     #RCVER1,R2      ;Assume no other job
        .RCVDW  #AREA,#MSG,#41. ;Wait for data
        BCS     ERR             ;Report failure
        MOV     #RCVER2,R2      ;Assume length error
        CMP     MSG,#2          ;Two words received?
        BNE     ERR             ;Branch if not
        MOV     MSG+4,R2        ;Load shared buffer addr
        MOV     #TMPBF,R3       ;Load temp buffer addr
20$:    MOVB    -(R2),(R3)+     ;Move data into temp
        CMP     R2,MSG+2        ;All done?
        BHI     20$             ;Branch if not
        CLRB    (R3)            ;Insert null byte
        MOV     #TMPBF,R3       ;Load temp buffer addr
30$:    MOVB    (R3)+,(R2)+     ;Move data into shared
        BNE     30$             ;Loop until end of text
        MOV     #SNDERR,R2      ;Assume send error
        .SDATW  #AREA,#MSG,#1   ;Send data and wait
        BCS     ERR             ;Report failure
        .EXIT                   ;Otherwise, exit
ERR:    .PRINT  R2              ;Print error message
        .EXIT                   ;And exit
        .END    START
```

```
        .TITLE  PR1704  Solution 17-1 (Step 2) Foreground
        .MCALL  .PEEK   .POKE   .QSET   .GTLIN
        .MCALL  .SDATW  .RCVDW  .PRINT  .EXIT
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
INBUF:  .BLKW   42.             ;Input buffer
MSG:    .BLKW   2               ;Send buffer
PROMPT: .ASCIZ  "Input your message: "
REPLY:  .ASCIZ  "In reverse, that becomes:"
SNDERR: .ASCIZ  "?PR1704 Send error, no other job"
RCVER1: .ASCIZ  "?PR1704 Receive error, no other job"
        .EVEN
        JSW     =       44
        LOWER   =       40000
```

```
START:  .QSET   #QLIST,#2       ;Allocate queue elements
        .PEEK   #AREA,#JSW      ;Get JSW
        MOV     R0,R1           ;Save the value
        BIS     #LOWER,R1       ;Enable lower case input
        .POKE   #AREA,#JSW,R1   ;Set JSW
        .GTLIN  #INBUF,#PROMPT  ;Get input from console
        MOV     #INBUF,R1       ;Load string buffer addr
10$:    TSTB    (R1)+           ;Search for null byte
        BNE     10$             ;Loop until byte found
        DEC     R1              ;Backup over null byte
        MOV     #INBUF,MSG      ;Load buffer starting
        MOV     R1,MSG+2        ;  and ending address
        MOV     #SNDERR,R2      ;Assume send error
        .SDATW  #AREA,#MSG,#2   ;Send data to background
        BCS     ERR             ;Report error
        MOV     #RCVER1,R2      ;Assume receive error
        .RCVDW  #AREA,#MSG,#1   ;Wait for reply
        BCS     ERR             ;Report error
        .PRINT  #REPLY          ;Print heading
        .PRINT  #INBUF          ;Print reply
        .EXIT                   ;Exit
ERR:    .PRINT  R2              ;Print error message
        .EXIT                   ;And exit
        .END    START
```

17-1. (Step 3) MACRO-11

```
.EDIT/CREATE PR1705.MAC

        .TITLE  PR1705  Solution 17-1 Background
        .MCALL  .QSET   .SDATW  .RCVDW  .PRINT  .EXIT
        .MCALL  .CHCOPY .READW  .WRITW  .CLOSE
        .NLIST  BEX
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
MSG:    .BLKW   3               ;Shared buffer data
TMPBF:  .BLKW   42.             ;Work area
INBUF:  .BLKW   42.             ;File input area
SNDERR: .ASCIZ  "?PR1705 Send error, no other job"
RCVER1: .ASCIZ  "?PR1705 Receive error, no other job"
RCVER2: .ASCIZ  "?PR1705 Receive error, data length"
CHERR:  .ASCIZ  "?PR1705 Channel copy error"
RDERR:  .ASCIZ  "?PR1705 Read error"
WRTERR: .ASCIZ  "?PR1705 Write error"
CLSERR: .ASCIZ  "?PR1705 Close error"
        .EVEN
START:  .QSET   #QLIST,#2       ;Allocate queue elements
        MOV     #RCVER1,R2      ;Assume no other job
        .RCVDW  #AREA,#MSG,#2   ;Receive data from frgrnd
        BCS     ERR             ;Report failure
        MOV     #RCVER2,R2      ;Assume data length error
        CMP     MSG,#2          ;Two words received?
        BNE     ERR             ;Branch if not
        MOV     #CHERR,R2       ;Assume chan copy error
        MOV     MSG+2,R3        ;Load foregound channel
        .CHCOPY #AREA,#1,R3     ;  number and copy it
        BCS     ERR             ;Report error
        MOV     #RDERR,R2       ;Assume read error
        .READW  #AREA,#1,#INBUF,MSG+4,#0 ;Read data
        BCS     ERR             ;Report error
        MOV     #INBUF,R2       ;Load data buffer addr
10$:    TSTB    (R2)+           ;Search for end of
        BNE     10$             ;  buffer
        DEC     R2              ;Backup over null byte
        MOV     #TMPBF,R3       ;Point to temp buffer
20$:    MOVB    -(R2),(R3)+     ;Move data into output
        CMP     R2,#INBUF       ;  buffer
        BHI     20$             ;Branch if not done
        CLRB    (R3)            ;Insert null byte
        MOV     #WRTERR,R2      ;Assume write error
        .WRITW  #AREA,#1,#TMPBF,MSG+4,#1 ;Write data
        BCS     ERR             ;Report error
        MOV     #SNDERR,R2      ;Assume send error
        .SDATW  #AREA,#MSG+4,#1 ;Send data and wait
        BCS     ERR             ;Report failure
        MOV     #CLSERR,R2      ;Assume close error
        .CLOSE  #1              ;Close file
        BCS     ERR             ;Report error
        .EXIT                   ;Otherwise, exit
ERR:    .PRINT  R2              ;Print error message
        .EXIT                   ;And exit
        .END    START
```

```
.EDIT/CREATE PR1706.MAC

        .TITLE  PR1706  Solution 17-1 Foreground
        .MCALL  .PEEK   .POKE   .QSET   .GTLIN
        .MCALL  .SDATW  .RCVDW  .PRINT  .EXIT
        .MCALL  .ENTER  .WRITW  .READW  .CLOSE
        .ENABL  LC
AREA:   .BLKW   5               ;EMT argument block
QLIST:  .BLKW   2*7             ;Queue element list
INBUF:  .BLKW   42.             ;Input buffer
MSG:    .BLKW   2               ;Send data buffer
ZERO:   .WORD   0
FILE:   .RAD50  /DK PR1706TXT/
PROMPT: .ASCIZ  "Input your message: "
REPLY:  .ASCII  "In reverse, that becomes:"<15><12>
MSGBUF: .BLKB   82.
SNDERR: .ASCIZ  "?PR1706 Send error, no other job"
RCVER1: .ASCIZ  "?PR1706 Receive error, no other job"
OPNERR: .ASCIZ  "?PR1706 Enter error"
```

```
WRTERR: .ASCIZ  "?PR1706 Write error"
RDERR:  .ASCIZ  "?PR1706 Read error"
CLSERR: .ASCIZ  "?PR1706 Close error"
        .EVEN
        JSW     =       44
        LOWER   =       40000
START:  .QSET   #QLIST,#2       ;Allocate queue elements
        .PEEK   #AREA,#JSW      ;Get JSW
        MOV     R0,R1           ;Copy job status word
        BIS     #LOWER,R1       ;Enable lower case input
        .POKE   #AREA,#JSW,R1   ;Set JSW
        .GTLIN  #INBUF,#PROMPT  ;Get input from console
        MOV     #INBUF,R1       ;Load input buffer addr
10$:    TSTB    (R1)+           ;Search for null byte
        BNE     10$             ;Loop until byte found
        SUB     #INBUF,R1       ;Calculate string length
        ASR     R1              ;  and then word count
        MOV     R1,MSG+2        ;Load word count
        MOV     #OPNERR,R2      ;Assume open error
        .ENTER  #AREA,#0,#FILE,#2 ;Create 2 block file
        BCS     ERR             ;Report error
        MOV     #WRTERR,R2      ;Assume write error
        .WRITW  #AREA,#0,#INBUF,R1,#0 ;Write text
        BCS     ERR             ;Report error
        .WRITW  #AREA,#0,#ZERO,#1,#1 ;Zero block 1
        CLR     MSG             ;Load channel number
        MOV     #SNDERR,R2      ;Assume send error
        .SDATW  #AREA,#MSG,#2   ;Send data
        BCS     ERR             ;Report error
        MOV     #RCVER1,R2      ;Assume no other job
        .RCVDW  #AREA,#MSG,#1   ;Wait for reply
        BCS     ERR             ;Report error
        MOV     #RDERR,R2       ;Assume read error
        .READW  #AREA,#0,#INBUF,R1,#1 ;Read data
        BCS     ERR             ;Report error
        MOV     #CLSERR,R2      ;Assume close error
        .CLOSE  #0              ;Close the new file
        BCS     ERR             ;Report error
        MOV     #INBUF,R1       ;Prepare to move text
        MOV     #MSGBUF,R2      ;  to output buffer
20$:    MOVB    (R1)+,(R2)+     ;Move text
        BNE     20$             ;Loop until done
        .PRINT  #REPLY          ;Print modifed message
        .EXIT
ERR:    .PRINT  R2              ;Print error message
        .EXIT                   ;And exit
        .END    START
```

```
.MACRO PR1705,PR1706

.LINK PR1705

.RUN PR1705
?PR1705 Receive error, no other job

.LINK/FOREGROUND PR1706

.FRUN PR1706

F>
Input your message:

B>

.RUN PR1705

F>
ABCDEFGHIJKLMNOPQRSTUVWXYZ...1234567890!

B>

.
F>
In reverse, that becomes:
!0987654321...ZYXWVUTSRQPONMLKJIHGFEDCBA

B>


.TYPE PR1706.TXT
ABCDEFGHIJKLMNOPQRSTUVWXYZ...1234567890!
!0987654321...ZYXWVUTSRQPONMLKJIHGFEDCBA
```

17-1. (Step 3) FORTRAN IV

```
        PROGRAM PR1705
C
C       Solution to 17-1 (Step 3) Background
C
C       Receive shared file from foreground, read block
C       0, reverse string, and write it to block 1.
C
        BYTE STR1(84), STR2(84)
        INTEGER*2 MSG(3)
C
C       Allocate queue elements.
C
```

```
      IF (IQSET(2) .NE. 0)
     1   STOP '?PR1705 Insufficient queue elements'
C
C     Receive string from foreground job.
C
      IF (IRCVDW(MSG,2) .NE. 0)
     1   STOP '?PR1705 Receive error'
      IF (MSG(1) .NE. 2)
     1   STOP '?PR1705 Receive length error'
      IF (ICHCPY(1,MSG(2)) .NE. 0)
     1   STOP '?PR1705 Channel copy error'
      IERR=IREADW(MSG(3),STR1,0,1) ! Read block 0
      IF (IERR .LT. 0) STOP '?PR1705 Read error'
      L=LEN(STR1)
      DO 10 I=1,L
10    STR2(I)=STR1(L-I+1)     ! Reverse string
      STR2(L+1)=0             ! Append null byte
      IERR=IWRITW(MSG(3),STR2,1,1) ! Write block 1
      IF (IERR .EQ. -1)
     1   STOP '?PR1705 Write error -- PAST EOF'
      IF (IERR .EQ. -2)
     1   STOP '?PR1705 Write error -- HARDWARE ERROR'
      IF (IERR .EQ. -3)
     1   STOP '?PR1705 Write error -- FILE NOT OPEN'
C
C     Send reversal completed to foreground.
C
      IF (ISDATW(MSG(3),1) .NE. 0)
     1   STOP '?PR1705 Send error'
      IF (ICLOSE(1) .NE. 0)
     1   STOP '?PR1705 Close error'
      CALL EXIT
      END


      PROGRAM PR1706
C
C     Solution to 17-1 (Step 3) Foreground
C
C     Read string and write it to a file, send
C     channel to background, wait for return, and
C     print results.
C
      BYTE STR1(84), PROMPT(80)
      INTEGER*2 FILE(4),MSG(2)
      DATA FILE /3RDK ,3RPR1,3R807,3RTXT/
C
C     Allocate queue elements.
C
      IF (IQSET(2) .NE. 0)
     1   STOP '?PR1706 Insufficient queue elements'
C
C     Prompt for string and get it.
C
```

```
        CALL IPOKE("44,"40000 .OR. IPEEK("44))
        CALL SCOPY('Input your message:',PROMPT)
        CALL GTLIN(STR1,PROMPT)
        IERR=IENTER(0,FILE,2)   ! Create 2-Block file
        IF (IERR .LT. 0)
     1     STOP '?PR1706 File create error'
        MSG(1)=0                ! Send channel number
        MSG(2)=(LEN(STR1)+2)/2  ! Send length in words
        IERR=IWRITW(MSG(2),STR1,0,0) ! Write data
        IF (IERR .LT. 0) STOP '?PR1706 Write error'
        IERR=IWRITW(1,MSG,1,0)  ! Zero block 1
        IF (IERR .LT. 0) STOP '?PR1706 Write error'
C
C       Send shared file data.
C
        IF (ISDATW(MSG,2) .NE. 0)
     1     STOP '?PR1706 Send error'
C
C       Receive data from background.
C
        IF (IRCVDW(MSG,1) .NE. 0)
     1     STOP '?PR1706 Receive error'
        IF (MSG(1) .NE. 1)
     1     STOP '?PR1706 Receive length error'
        IERR=IREADW(MSG(2),STR1,1,0) ! Read data
        IF (IERR .LT. 0) STOP '?PR1706 Read error'
        IF (ICLOSE(0) .NE. 0)
     1     STOP '?PR1706 Close error'
        CALL PRINT('In reverse, that becomes:')
        CALL PRINT(STR1)
        CALL EXIT
        END
```

CHAPTER 18

18-1. MACRO-11

```
.EDIT/OUTPUT:PR1803.MAC PR1801.MAC

        .TITLE  PR1803  SOLUTION TO 18-1
;
;       This program performs a file-to-file copy by
;       using a dynamically allocated buffer.  The
;       program performs only one copy operation.
;
        .MCALL  .EXIT   .FETCH  .LOOKUP .ENTER  .PRINT
        .MCALL  .READW  .WRITW  .CLOSE  .SRESET
        .MCALL  .LOCK   .UNLOCK .GVAL   .SETTOP
```

```
EMTARG: .BLKW   6               ;EMT argument block
INFILE: .RAD50  /DK TRAN1 XYZ/  ;Copy from DK:TRAN1.XYZ
OUTFIL: .RAD50  /DK TRAN2 XYZ/  ;   to DK:TRAN2.XYZ
LIMITS: .LIMIT                  ;Program limits
BUFPTR: .BLKW   1               ;I/O buffer address
ERROR:  .BYTE                   ;Error status byte
ANNCE:  .ASCIZ  "Program copies TRAN1.XYZ to TRAN2.XYZ"
FCH1MS: .ASCIZ  "Error on FETCH of output handler"
FCH2MS: .ASCIZ  "Error on FETCH of input handler"
LKPMES: .ASCIZ  "Error on LOOKUP of input file"
ENTMES: .ASCIZ  "Error on creation of output file"
RERRMS: .ASCIZ  "Read error, copy aborted"
WERRMS: .ASCIZ  "Write error, copy aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Setup Files For Copy
;
;       This routine sets up files for I/O.
;       The file specifications are fixed.
;
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     R1,-(SP)        ;Save register
        .PRINT  #ANNCE          ;Announce program
;
;       Allocate buffer from free memory
;
        .GVAL   #266            ;R0 = base of USR
        .SETTOP                 ;SETTOP to base of USR
        .LOCK                   ;Lock USR in memory
;
;       Fetch device handlers
;
        MOV     LIMITS+2,R1     ;Load high limit
        .FETCH  R1,#OUTFIL      ;Fetch output handler
        BCS     FCH1ER          ;Branch on error
        MOV     R0,R1           ;Load new high limit
        .FETCH  R1,#INFILE      ;Fetch input handler
        BCS     FCH2ER          ;Branch on error
        MOV     R0,BUFPTR       ;Save buffer address
;
;       Open files
;
        .LOOKUP #EMTARG,#3,#INFILE
        BCS     LKPERR          ;Branch on open error
        MOV     R0,R1           ;Save file length
        .ENTER  #EMTARG,#0,#OUTFIL
        BCC     DONE            ;Branch on success
;
;       Error Routines:
;
        .PRINT  #ENTMES         ;Issue create failure
        BR      ERDONE          ; message and return
LKPERR: .PRINT  #LKPMES         ;Issue open failure
```

```
        BR      ERDONE          ; message and return
FCH2ER: .PRINT  #FCH2MS         ;Issue FETCH error
        BR      ERDONE          ; message and return
FCH1ER: .PRINT  #FCH1MS         ;Issue FETCH error
ERDONE: .UNLOCK                 ;Unlock USR
        SEC                     ;Set error flag
        BR      DONE1
DONE:   .UNLOCK                 ;Unlock the USR
DONE1:  MOV     (SP)+,R1        ;Restore R1
        RETURN                  ;Return to caller
        .SBTTL  CPYRTN  -- Synchronous Copy
;
;       This routine copies the file opened on channel
;       3 to the file opened on channel 0.
;       Returns with C-BIT SET on error.
;
;       Note: All registers except R0 are preserved.
;
CPYRTN: MOV     R1,-(SP)        ;Save register
        CLR     R1              ;Reset block number
        CLRB    ERROR           ;Clear error flag
1$:     .READW  #EMTARG,#3,BUFPTR,#256.,R1
        BCC     2$              ;Branch if read OK
        TSTB    #52            ;End-Of-File (EOF)?
        BEQ     EXIT            ;Branch if so
        BR      RDERR           ;Issue error message
2$:     .WRITW  #EMTARG,#0,BUFPTR,#256.,R1
        BCS     WERR            ;Branch on write error
        INC     R1              ;Update block number
        BR      1$              ;And read next block
RDERR:  .PRINT  #RERRMS         ;Issue read error
        BR      EREXIT          ; message and return
WERR:   .PRINT  #WERRMS         ;Issue write error
EREXIT: DECB    ERROR           ;Set error flag
EXIT:   MOV     (SP)+,R1        ;Restore register
        TSTB    ERROR           ;Error? (clear C-BIT)
        BEQ     1$              ;Branch if not
        SEC                     ;Set C-BIT (error)
1$:     RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Close Files
CLSCHN: .LOCK                   ;Lock USR
        .CLOSE  #3              ;Close input file
        .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch on success
        .PRINT  #PRTCT          ;Issue protected file
PRGCHN:                         ;Purge files
RESET:  .UNLOCK                 ;Unlock USR
        .SRESET                 ;Reset system
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Open files
        BCS     1$              ;Branch on error
        CALL    CPYRTN          ;Copy the file
        BCS     1$              ;Branch on error
        CALL    CLSCHN                  ;Close the files
        BR      2$                      ; and exit
1$:     CALL    PRGCHN                  ;Purge the files
2$:     .EXIT                           ;Exit
        .END    START
```

```
.MACRO PR1803

.LINK PR1803

.COPY TT: TRAN1.XYZ
 Files copied:
^TEST DATA FILE WHICH WILL BE COPIED TO TRAN2.XYZ

*** EOF ***
^ZTT:            to DK:TRAN1.XYZ

.RUN PR1803
Program copies TRAN1.XYZ to TRAN2.XYZ

.TYPE TRAN2.XYZ
TEST DATA FILE WHICH WILL BE COPIED TO TRAN2.XYZ

*** EOF ***

.
```

18-1. FORTRAN IV

```
.EDIT/OUTPUT:PR1804.FOR PR1802.FOR

        PROGRAM PR1804
C
C       Solution to 18-1
C
C       This program performs a file copy using a
C       dynamically allocated buffer.  Only one file
C       is copied.
C
C       ********        CAUTION         ********
C       Compile with /NOSWAP option OR use SET USR
C       NOSWAP before executing!
C
        LOGICAL*1 SETUP,CPYRTN
        LOGICAL*1 ERROR
C
10      ERROR = SETUP()         ! Open files
        IF (ERROR) GO TO 20     ! Stop on error
        ERROR = CPYRTN()        ! Copy file
        IF (ERROR) GO TO 20     ! Stop on error
        CALL CLSCHN             ! Close files
        GO TO 30
20      CALL PRGCHN             ! Purge channels
30      CALL EXIT
        END
        FUNCTION SETUP
C
C       This routine sets up the files for I/O.
C       The file specifications are fixed.
C
C       Function returns .TRUE. on error.
C
        LOGICAL*1 SETUP
        INTEGER*2 INCHN,OUTCHN
        COMMON  /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 INFILE(4),OUTFIL(4)
        DATA INFILE/2RDK,3RTRA,2RN1,3RXYZ/
        DATA OUTFIL/2RDK,3RTRA,2RN2,3RXYZ/
C
C       Annouce the program and allocate channels.
C
        CALL PRINT('Program copies TRAN1.XYZ to TRAN2.XYZ')
        CALL LOCK               ! Lock the USR
        INCHN = IGETC()         ! Allocate input &
        OUTCHN = IGETC()        !  output channels
C
C       Fetch device handlers.
C
        IF (IFETCH(OUTFIL(1)) .NE. 0) GO TO 101
        IF (IFETCH(INFILE(1)) .NE. 0) GO TO 102
C
C       Open input file and create output file.
C
        LENGTH = LOOKUP(INCHN,INFILE)
        IF (LENGTH .LT. 0) GO TO 103
        IF (IENTER(OUTCHN,OUTFIL,LENGTH) .LT. 0)
     1     GO TO 104
        CALL UNLOCK             ! Unlock USR
        SETUP = .FALSE.         ! No error
        RETURN
C
C       ERROR ROUTINES
C
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
102     CALL PRINT('Error on FETCH of input handler')
        GO TO 200
103     CALL PRINT('Error on LOOKUP of input file')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     CALL UNLOCK             ! Unlock USR
        SETUP = .TRUE.          ! Error
        RETURN
        END
        FUNCTION CPYRTN
C
C       Single buffered, synchronous copy routine.
C
C       Function returns .TRUE. on error.
C
        EXTERNAL IREADW, IWRITW
        LOGICAL*1 CPYRTN
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        INTEGER*2 BUFPTR,BLOCK
        BLOCK = 0               ! Reset block number
C
C       Allocate buffer dynamically.
C
        IF (IGETSP(256,256*20,BUFPTR) .LT. 0)
     1     GO TO 102
C
C       Read/write loop.
C
20      IERR=INDIR(IREADW,0,256,1,BUFPTR,0,BLOCK,0,INCHN)
        IF (IERR .GE. 0) GO TO 30 ! Successful read
        IF (IERR .EQ. (-1)) GO TO 150   ! End of File
        GO TO 100               ! Error
30      IF (INDIR(IWRITW,0,256,1,BUFPTR,0,BLOCK,0,OUTCHN)
     1     .LT. 0) GO TO 101
        BLOCK = BLOCK+1         ! Update block number
        GO TO 20
C
C       ERROR ROUTINES
C
100     CALL PRINT('Read error, copy aborted')
        GO TO 140
101     CALL PRINT('Write error, copy aborted')
        GO TO 140
102     CALL PRINT('Cannot get buffer space')
140     CPYRTN = .TRUE.
        RETURN
C
C       Successful return.
C
150     CPYRTN = .FALSE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close files.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL LOCK               ! Lock USR
        CALL CLOSEC(INCHN)
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
```

```
        CALL UNLOCK             ! Unlock USR
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channels.
C
        INTEGER*2 INCHN,OUTCHN
        COMMON /CHNNLS/ INCHN,OUTCHN
        CALL LOCK               ! Lock USR
        CALL PURGE(INCHN)
        CALL PURGE(OUTCHN)
        CALL UNLOCK             ! Unlock USR
        RETURN
        END


.FORTRAN PR1804.FOR/NOSWAP

.LINK PR1804,SY:FORLIB

.RUN PR1804
Program copies TRAN1.XYZ to TRAN2.XYZ

.DIFF TRAN1.XYZ TRAN2.XYZ
?SRCCOM-I-No differences found

.
```

CHAPTER 19

19-1. MACRO-11

```
.EDIT/CREATE PR1902.MAC

        .TITLE  PR1902  Solution to 19-1
;
;       Program creates the output file specified by
;       the user.
;
        .MCALL  .EXIT   .CSIGEN .PRINT  .WRITW
        .MCALL  .CLOSE  .SRESET
EMTARG: .BLKW   6               ;EMT argument block
LIMITS: .LIMIT                  ;Program limits
SPSAVE: .BLKW   1               ;Saved stack pointer
DEFTYP: .WORD   0,0,0,0         ;Default file types
BUFFER: .BLKW   256.            ;File I/O buffer
        BUFFE   =       .       ;End of I/O buffer
PRMPT:  .ASCIZ  "Specify output filename"
```

```
CSIERR: .ASCIZ  "Error on CSI call"
WERRMS: .ASCIZ  "Write error, output aborted"
PRTCT:  .ASCIZ  "Protected output file already exists"
        .EVEN
        .SBTTL  SETUP   -- Create Output File
;
;       This routine gets a command string using the
;       CSI and opens the specified output file.
;
;       Returns with C-Bit SET on error.
;
SETUP:  MOV     SP,SPSAVE       ;Save stack pointer
        .PRINT  #PRMPT          ;Display prompt
        .CSIGEN LIMITS+2,#DEFTYP,#0
        MOV     SPSAVE,SP       ;Restore stack
        BCC     10$             ;Branch on success
        .PRINT  #CSIERR         ;Issue error message
        SEC                     ;Set error flag
10$:    RETURN                  ;Return to caller
        .SBTTL  WRTFIL  -- Synchronous Output
;
;       This routine writes to the output file opened
;       on channel 0.
;
;       Returns with C-Bit SET on error.
;
;       Note: All registers except R0 are preserved.
;
WRTFIL: MOV     R1,-(SP)        ;Save registers
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        CLR     R3              ;Clear counter
        CLR     R1              ;Clear block number
10$:    MOV     #BUFFER,R2      ;Load buffer address
20$:    MOV     R3,(R2)+        ;Store counter
        INC     R3              ;Increment counter
        CMP     R2,#BUFFE       ;At end of buffer?
        BNE     20$             ;Loop if not
        .WRITW  #EMTARG,#0,#BUFFER,#256.,R1
        BCS     WERR            ;Branch on error
        INC     R1              ;Update block number
        CMP     R1,#10.         ;10. blocks written?
        BLT     10$             ;Branch if not
        CLC                     ;Clear error flag
        BR      EXIT            ;Otherwise, done
WERR:   .PRINT  #WERRMS         ;Issue write error
        SEC                     ;Set error flag
EXIT:   MOV     (SP)+,R3        ;Restore registers
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        RETURN                  ;Return to caller
        .SBTTL  CLSCHN  -- Close File
CLSCHN: .CLOSE  #0              ;Close output file
        BCC     RESET           ;Branch on success
        .PRINT  #PRTCT          ;Issue file protected
PRGCHN:                         ;Purge output file
RESET:  .SRESET                 ;Reset system
        RETURN                  ;Return to caller
        .SBTTL  MAIN PROGRAM
START:  CALL    SETUP           ;Open output file
        BCS     START           ;Loop on error
        CALL    WRTFIL          ;Write file
        BCS     10$             ;Branch on error
        CALL    CLSCHN          ;Close the output
        BR      20$             ;  file and exit
10$:    CALL    PRGCHN          ;Purge output file
20$:    .EXIT                   ;Exit
        .END    START
```

```
.EXECUTE PR1902.MAC
Specify output filename
*DK:TEST1.DAT=

.DUMP/TERMINAL TEST1.DAT/NOASCII
DK:TEST1.DAT
BLOCK NUMBER  000000
000/ 000000 000001 000002 000003 000004 000005 000006 000007
020/ 000010 000011 000012 000013 000014 000015 000016 000017
040/ 000020 000021 000022 000023 000024 000025 000026 000027
060/ 000030 000031 000032 000033 000034 000035 000036 000037
100/ 000040 000041 000042 000043 000044 000045 000046 000047
120/ 000050 000051 000052 000053 000054 000055 000056 000057
140/ 000060 000061 000062 000063 000064 000065 000066 000067

                .
                .
                .

760/ 000370 000371 000372 000373 000374 000375 000376 000377

BLOCK NUMBER  000001
000/ 000400 000401 000402 000403 000404 000405 000406 000407
020/ 000410 000411 000412 000413 000414 000415 000416 000417
040/ 000420 000421 000422 000423 000424 000425 000426 000427

                .
                .
                .

.DIR TEST1.DAT
 21-Mar-84
TEST1 .DAT    10  21-Mar-84
 1 Files, 10 Blocks
 552 Free blocks
```

19-1. FORTRAN IV

```
.EDIT/CREATE PR1902.FOR

         PROGRAM PR1902
C
C        Solution to 19-1.
C
C        Program creates a 10 block long output file.
C
        LOGICAL*1 SETUP,WRTFIL
        LOGICAL*1 ERROR
C
10      ERROR = SETUP()         ! Open output file
        IF (ERROR) GO TO 10     ! Try again on error
        ERROR = WRTFILN()       ! Output the data
        IF (ERROR) GO TO 20     ! Stop on error
        CALL CLSCHN             ! Close output file
        GO TO 30
20      CALL PRGCHN             ! Purge output file
30      CALL EXIT
        END
        FUNCTION SETUP
C
C       This routine gets a command string and creates
C       the specified output file.  The CSI is used.
C
C       Returns .TRUE. on error.
C
        LOGICAL*1 SETUP
        INTEGER*2 OUTCHN
        COMMON /CHNNLS/ OUTCHN
        LOGICAL*1 FRSTTM
        DATA FRSTTM/.TRUE./
        INTEGER*2 DEFTYP(4),FILES(39),LENGTH
        DATA DEFTYP/4*0/
C
        IF (.NOT. FRSTTM) GO TO 10
        OUTCHN = IGETC()        ! Allocate output
        FRSTTM = .FALSE.        !  channel only once
C
C       Output prompt and parse command string.
C
10      CALL PRINT('Specify output file:')
        IF (ICSI(FILES,DEFTYP,,,0) .NE. 0) GO TO 100
C
C       Fetch device handler and create output file.
C
        IF (IFETCH(FILES(1)) .NE. 0) GO TO 101
        IF (IENTER(OUTCHN,FILES(1),10) .LT. 0)
      1    GO TO 104
        SETUP = .FALSE.         ! No error
        RETURN
C
```

```
C       ERROR ROUTINES
C
100     CALL PRINT('Error on CSI call')
        GO TO 200
101     CALL PRINT('Error on FETCH of output handler')
        GO TO 200
104     CALL PRINT('Error on creation of output file')
200     SETUP = .TRUE.          ! Error
        RETURN
        END
        FUNCTION WRTFIL
C
C       Single buffered, synchronous output routine.
C
C       Returns .TRUE. on error.
C
        LOGICAL*1 WRTFIL
        INTEGER*2 OUTCHN
        COMMON /CHNNLS/ OUTCHN
        INTEGER*2 BUFFER(256),BLOCK
C
C       Build a buffer and write it to the file.
C
        ICT = 0                 ! Reset counter
        DO 50 BLOCK=0,9
        DO 10 I=1,256
        BUFFER(I) = ICT         ! Store counter
        ICT = ICT+1             ! Increment counter
10      CONTINUE
        IF (IWRITW(256,BUFFER,BLOCK,OUTCHN) .LT. 0)
     1     GO TO 101
50      CONTINUE
        WRTFIL  = .FALSE.       ! Indicate success
        RETURN
101     CALL PRINT('Write error, output aborted')
140     WRTFIL = .TRUE.
        RETURN
        END
        SUBROUTINE CLSCHN
C
C       Close output file.
C
        INTEGER*2 OUTCHN
        COMMON /CHNNLS/ OUTCHN
        IF (ICLOSE(OUTCHN) .EQ. 4) CALL PRINT
     1     ('Protected output file already exists')
        RETURN
        END
        SUBROUTINE PRGCHN
C
C       Purge channel.
C
        INTEGER*2 OUTCHN
        COMMON /CHNNLS/ OUTCHN
        CALL PURGE(OUTCHN)
        RETURN
        END


.EXECUTE/LINKLIB:SY:FORLIB PR1902.FOR
Specify output filename
*DK:TEST2.DAT=

.DUMP/TERMINAL TEST2.DAT/NOASCII
DK:TEST2.DAT
BLOCK NUMBER  000000
000/ 000000 000001 000002 000003 000004 000005 000006 000007
020/ 000010 000011 000012 000013 000014 000015 000016 000017
040/ 000020 000021 000022 000023 000024 000025 000026 000027
060/ 000030 000031 000032 000033 000034 000035 000036 000037
100/ 000040 000041 000042 000043 000044 000045 000046 000047
120/ 000050 000051 000052 000053 000054 000055 000056 000057
140/ 000060 000061 000062 000063 000064 000065 000066 000067

                 .
                 .
                 .

.DIR TEST2.DAT
 21-Mar-84
TEST2 .DAT    10  21-Mar-84
 1 Files, 10 Blocks
 502 Free blocks
```

CHAPTER 20

20-1. MACRO-11

```
.EDIT/CREATE PR2001.MAC

        .TITLE  PR2001  Solution to 20-1
        .ENABL  LC
        .MCALL  .DATE   .PRINT  .EXIT   .TWAIT  .QSET
        .MCALL  .TTINR  .PEEK   .POKE   .MRKT   .RCTRL
        .GLOBL  TIME    DATE    ;SYSLIB routines
        JSW     =    44         ;Job Status Word
        LOWER   = 40000         ;Lower Case bit
        SPEC    = 10000         ;Special TT mode bit
        RETRN   =   100         ;Inhibit TT wait bit
EMTBLK: .BLKW   4               ;EMT Argument block
TIMARG: .BYTE   1,0             ;TIME argument block
        .WORD   TIMBF
```

```
DATARG: .BYTE   1,0             ;DATE argument block
        .WORD   DATBF
DELAY:  .WORD   0,2.*60.        ;2 second delay (tics)
LIMIT:  .WORD   0,20.*60.       ;Time limit (20 secs)
FLAG:   .WORD   0               ;20 secs delay expired
QEL:    .BLKW   20.*10.         ;Queue element buffer
MSG:    .ASCII  "The time is: "
TIMBF:  .ASCII  "HH:MM:SS, on "
DATBF:  .ASCIZ  "DD-MMM-YY"
NOTIM:  .ASCIZ  "Please set the date and time"
NOQUE:  .ASCIZ  "Not enough queue elements"
INSTR:  .ASCII  "Type as much as you can in 2"
        .ASCIZ  " seconds, starting now:"
OUT:    .ASCII  <15><12>"You managed to type:"
        .ASCII  <15><12><12>
INBF:   .BLKB   80.
        .EVEN
START:  .PEEK   #EMTBLK,#JSW    ;Get JSW
        MOV     R0,R1           ;Copy JSW into R1
        BIS     #LOWER,R1       ;Enable lower case
        .POKE   #EMTBLK,#JSW,R1 ;Set JSW
        .RCTRLO                 ;Reset Control/O
        MOV     #TIMARG,R5      ;Load argument block
        JSR     PC,TIME         ;CALL TIME
        .DATE                   ;Get date
        TST     R0              ;Date specified?
        BNE     GO1             ;Branch if so
        .PRINT  #NOTIM          ;No, ask for user
        .EXIT                   ; to set date/time
GO1:    MOV     #DATARG,R5      ;Load argument block
        JSR     PC,DATE         ;CALL DATE
        .PRINT  #MSG            ;Print date and time
        .QSET   #QEL,#20.       ;Allocate queues
        .MRKT   #EMTBLK,#LIMIT,#CRTN,#1
        BCC     LOOP            ;Branch on success
NO:     .PRINT  #NOQUE          ;Print error message
        .EXIT                   ;And exit
LOOP:   TST     FLAG            ;Timer expired?
        BNE     STOP            ;Branch if so, quit
        .PRINT  #INSTR          ;Display instructions
        .PEEK   #EMTBLK,#JSW    ;Enable special TT
        MOV     R0,R1           ; mode and inhibit
        BIS     #SPEC!RETRN,R1  ; TT wait
        .POKE   #EMTBLK,#JSW,R1 ;Set JSW
        .RCTRLO                 ;Reset Control/O
        .TWAIT  #EMTBLK,#DELAY  ;Wait for 2 seconds
        BCS     NO              ;Branch on error
        MOV     #INBF,R2        ;Load buffer address
10$:    .TTINR                  ;Read a character
        BCS     20$             ;Branch if none
        MOVB    R0,(R2)+        ;Add char to buffer
        BR      10$             ;Read next char
20$:    .PEEK   #EMTBLK,#JSW    ;Disable special TT
        MOV     R0,R1           ; mode and enable
        BIC     #SPEC!RETRN     ; TT wait
        .POKE   #EMTBLK,#JSW,R1 ;Set JSW
        .RCTRLO                 ;Reset Control/O
        CLRB    (R2)            ;Add terminator byte
        .PRINT  #OUT            ;Display text
        BR      LOOP            ;And repeat
STOP:   MOV     #TIMARG,R5      ;Load argument block
        JSR     PC,TIME         ;CALL TIME
        MOV     #DATARG,R5      ;Load argument block
        JSR     PC,DATE         ;CALL DATE
        .PRINT  #MSG            ;Display date & time
        .EXIT                   ;And exit
;
;       ** MARK TIME COMPLETION ROUTINE **
;
CRTN:   MOV     #1,FLAG         ;Set time expired
        RTS     PC              ;Return
        .END    START
```

```
.MACRO PR2001

.LINK PR2001,SY:FORLIB

.RUN PR2001
The time is: 14:40:02, on 20-MAR-84
Type as much as you can in 2 seconds, starting now:

You managed to type:

thi
Type as much as you can in 2 seconds, starting now:

You managed to type:

sabcd
Type as muech as you can in 2 seconds, starting now:

You maanaged to type:

efaaaaaaaaaaaaaaaaaaaa
Type asa much as you can in 2 seconds, starting now:

You managed to type:

aaaaaaaaaaaaaajjj
Type as much as you can in 2 seconds, starting now:

You managed to type:

jj
Type as much as you can in 2 seconds, starting now:

You managed to type:
```

```
Type as much as you can in 2 seconds, starting now:

You managed to type:

121
Type as much as you can in 2 seconds, starting now:

You managed to type:

21212121212121212121212
The time is: 14:40:23, on 20-MAR-84

.
```

20-1. FORTRAN IV

```
.EDIT/CREATE PR2001.FOR

        PROGRAM PR2001
C
C       Solution to 20-1.
C
C       Get and print time and date, exiting if no date.
C       Loop reading from the terminal every 2 seconds
C       for a total of 20 seconds.  Print the new time
C       and date and exit.
C
        EXTERNAL CRTN
        BYTE MSGBFR(100), TIMSTR(8), TODAY(9)
        INTEGER*2 AREA(4), DELAY(2), FLAG
        COMMON /DATA/ FLAG
C
        CALL IPOKE("44,IPEEK("44).OR."40000)
        CALL RCTRLO               ! Enable lower case
        CALL TIME(TIMSTR)         ! Get current time
        CALL DATE(TODAY)          !  and date
        IF (TODAY(1) .EQ. ' ')
     1     STOP 'Please set the date and time'
        TYPE 10,TIMSTR,TODAY
10      FORMAT (' The time is: ',8A1,', on ',9A1)
        IF (IQSET(3) .NE. 0)
     1     STOP 'Not enough queue elements'
        FLAG = 0                  ! Clear timer flag
        IERR = ITIMER(0,0,20,0,AREA,1,CRTN)
        DELAY(1) = 0              ! Build time delay
        DELAY(2) = 2*60           !  (delay 2 secs)
20      IF (FLAG .NE. 0) GO TO 60 ! Timer expired
        TYPE 25
```

```
25      FORMAT (' Type as much as you can in 2',
     1          'seconds, starting now:'/)
        CALL POKE("44,IPEEK("44).OR."10100)
        CALL RCTRLO              ! Change JSW bits
        IERR = ITWAIT(DELAY)     ! Wait for 2 seconds
        DO 40 I=1,100
        IERR = ITTINR()          ! Accept a character
        IF (IERR .LT. 0) GO TO 50 ! None available
        MSGBFR(I) = IERR         ! Store character
40      CONTINUE
50      MSGBFR(I) = 0            ! Mark end of buffer
        CALL IPOKE("44,IPEEK("44).AND..NOT."10100)
        CALL RCTRLO              ! Reset JSW bits
        TYPE 55
55      FORMAT (' You managed to type:',/)
        CALL PRINT(MSGBFR)       ! Display text read
        GO TO 20                 ! Repeat until timeout
60      CALL TIME(TIMSTR)        ! Get current time
        CALL DATE(TODAY)         !  and date
        TYPE 10,TIMSTR,TODAY     ! Output time & date
        CALL EXIT                ! Exit
        END
        SUBROUTINE CRTN(ID)
C
C       ** TIMER COMPLETION ROUTINE **
C
C       Runs after 20 seconds and sets a flag in the
C       common data region to indicate that the MAIN
C       PROGRAM should now stop.
C
        INTEGER*2 ID,FLAG
        COMMON /DATA/ FLAG
        FLAG = 1                 ! Set timer flag
        RETURN
        END
```

```
.EXECUTE/LINKLIBRARY:SY:FORLIB/FORTRAN PR2001
The time is: 15:02:49, on 20-MAR-84
Type as much as you can in 2 seconds, starting now:

You managed to type:
abc

Type as much as you can in 2 seconds, starting now:

You managed to type:
dthis is a test

Type as much as you can in 2 seconds, starting now:

You managed to type:
 of the emerg
```

```
Type as much as you can in 2 seconds, starting now:

You managed to type:
ency

Type as much as you can in 2 seconds, starting now:

You managed to type:
121212

Type as much as you can in 2 seconds, starting now:

You managed to type:
129999999

Type as much as you can in 2 seconds, starting now:

You managed to type:
9999999999999999999999999999999999999999

Type as much as you can in 2 seconds, starting now:

You managed to type:
9999999999

The time is: 15:03:11, on 20-MAR-84
```

Index

Programming with RT-11, Volume 2 teaches programmers to incorporate the RT-11 system services into the MACRO-11 or FORTRAN IV programs they create. The book examines the use of programmed requests to perform file and terminal input/output, foreground/background communication, and synchronous and nonsynchronous input/output. Applications and practice exercises encourage actual use of the RT-11 callable system facilities.

The RT-11 Technical User's Series presents up-to-date information on RT-11. Other books in the series are:

Programming with RT-11, Volume 1
Program Development Facilities

Working with RT-11

Tailoring RT-11
System Management and Programming Facilities

The authors develop courses for Educational Services Division of Digital Equipment Corporation in Reading, England.

digital

For a complete list of DECbooks, write to:

Digital Press
Digital Equipment Corporation
30 North Avenue, Burlington, MA 01803

ISBN 0-932376-33-9