EY-0072E-SG-0201

RSX-11M/M-PLUS Utilities and Commands

A Self-Paced Course

Volume II

digital

EY-0072E-SG-0201

RSX-11M/M-PLUS Utilities and Commands

A Self-Paced Course

Student Workbook
Volume II

Prepared by Educational Services
of
Digital Equipment Corporation

10/82-15

CONTENTS

SG STUDENT GUIDE

FIGURES

TABLES

EXAMPLES

PROGRAM DEVELOPMENT

INTRODUCTION

RSX-11M/M-PLUS operating systems provide complete facilities for all program development tasks. This module discusses program development in detail. MACRO-11, FORTRAN IV and FORTRAN IV-PLUS are emphasized since these languages are used in most applications.

OBJECTIVES

1. Assemble a MACRO-11 program.
2. Compile a FORTRAN IV or FORTRAN IV-PLUS program.
3. Task-build a program.
4. Use the task map to obtain basic information.
5. Run a task.

RESOURCES

1. _Introduction to RSX-11M/M-PLUS_
2. _RSX-11M/M-PLUS Command Language Manual_

OVERVIEW

Every task running on a computer is the solution to some well-defined problem. Someone said we need a method of automatically generating payroll checks (the problem) and a software developer wrote a program that reads payroll records and generates paychecks for everyone on the payroll (the solution). Or, someone said we need a method of entering our source code more efficiently and a programmer wrote a program that allows a terminal user to capture text into a file.

Program development is the process by which a problem solution is translated from a human-understandable form into a machine-understandable form. Some of the many steps necessary to do this conversion are shown in Figure 5-1. The process is longer than shown in the flowchart, but for our purposes, it begins at the point where the process needs the computer to complete the task.

Once the problem to be solved is defined and a solution is designed and coded, the first step (1 in Figure 5-1) is to enter the code into a file. Source statements are entered using a text editor. Output from the editor is an ASCII file containing the source statements that comprise the program (or a module of the program) written with correct syntax required by the programming language. This file is the human-understandable form of the program and the first step in the translation of the program into a machine-understandable form.

The next step (2 in Figure 5-1) is to take the output file from the editing session and assemble or compile it with the appropriate language processor. Table 5-2 shows the language processors available on the RSX-11M/M-PLUS operating systems. The language processor performs the following functions:

- checks the source statements for syntax errors
- generates program addresses for relocatable machine code
- produces a listing of the source statements
- lists any errors that exist

The primary output from this step is an object file that contains object modules of relocatable machine instructions.

If there are no assembly errors, the next step in the process is to build (link) the actual runnable image of the program (3 in Figure 5-1). This may require linking the program with other object modules located in the user's UFD, or in user or system libraries, as shown in Figure 5-3. It is the Task Builder's job to take all object modules and link them together, resolving references between modules, and incorporating modules from referenced system libraries. The Task Builder outputs an executable task image. An optional output file is a map file that contains information describing the allocation of addresses, program sections in task image, modules of origin, and the values of all global symbols. A symbol definition file is other optional output that contains information used to create a shared region. Shared regions and the use of the file are covered in another course.

Before releasing a task for general use, the task should be run and tested against original specifications to ensure that it does indeed run correctly and solves the original problem (4 in Figure 5-1). Debugging aids, like the On-Line Debugging Tool (ODT), can be used to help locate problems within the code. If any error conditions arise during this step it may be necessary to go back to the first step in the flowchart and repeat the steps once again until the task executes correctly.

When the task is ready for general use (5 in Figure 5-1), it can be permanently installed in the System Task Directory. Not all tasks are permanently installed. Your system manager makes this choice based upon how often a task is run and how critical its nature.

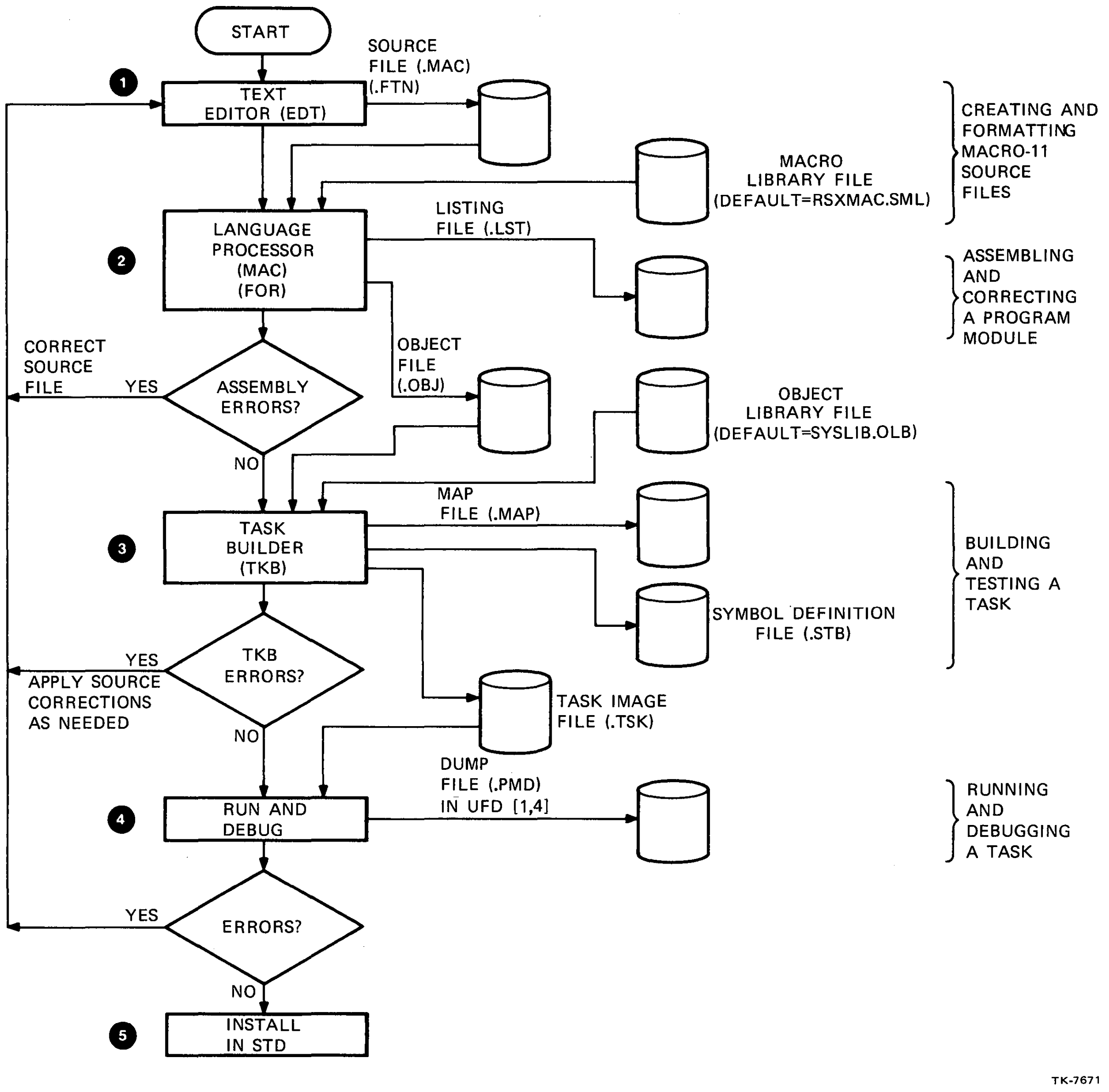

START
1
TEXT EDITOR (EDT)
SOURCE FILE (.MAC) (.FTN)
MACRO LIBRARY FILE (DEFAULT=RSXMAC.SML)
CREATING AND FORMATTING MACRO-11 SOURCE FILES
2
LANGUAGE PROCESSOR (MAC) (FOR)
LISTING FILE (.LST)
ASSEMBLING AND CORRECTING A PROGRAM MODULE
CORRECT SOURCE FILE
YES
ASSEMBLY ERRORS?
NO
OBJECT FILE (.OBJ)
OBJECT LIBRARY FILE (DEFAULT=SYSLIB.OLB)
3
TASK BUILDER (TKB)
MAP FILE (.MAP)
SYMBOL DEFINITION FILE (.STB)
BUILDING AND TESTING A TASK
YES
APPLY SOURCE CORRECTIONS AS NEEDED
TKB ERRORS?
NO
TASK IMAGE FILE (.TSK)
4
RUN AND DEBUG
DUMP FILE (.PMD) IN UFD [1,4]
RUNNING AND DEBUGGING A TASK
YES
ERRORS?
NO
5
INSTALL IN STD
TK-7671

Figure 5-1 The Program Development Process

PROGRAMMING LANGUAGES

There are three fundamental types of programming languages in use today; the difference is in the manner in which the translation of source statements into machine instructions occurs. These three types are interpreted, assembled and compiled languages.

In an interpreted language, such as BASIC, each source statement is translated and executed before the next line is read. There are no separate compile or task-build steps. The results of arithmetic expressions and input/output statements are immediate and can be checked for correctness. As there is no permanent output from the translation process, the cost of translating the source statements into machine instructions is borne each time the program is run. An interpreted language is most often used for applications that are run infrequently but need to be developed quickly. They are also used as an instructional language for beginning programmers.

In an assembled language, a program called an assembler translates source statements into machine instructions, which are saved in a file called an object file. Each source statement translates directly into one machine instruction. This intermediate file (called the object file) passes through another process, called linking, to make an executable image. Translation of source statements into machine instructions is not required each time the program is run. Therefore, execution time is less with an assembled language than an interpreted language.

A compiled language, often referred to as a high-level language, processes like the assembled language. However, the source statements are more English-like, and each source statement translates into one or more machine instructions. The program that converts high-level source code into object code is called a compiler.

Table 5-1 compares the three types of languages.

Table 5-1 Language Types

	Interpreted	Assembled	Compiled
Example	BASIC-11	MACRO-11	FORTRAN IV
Object Module	No	Yes	Yes
Task Image File	No	Yes	Yes
Translation	Each time	Once	Once
Development Time	Short	Longest	Medium
Execution Speed	Slow	Fastest	Medium
Application	Short, seldom run programs	Time-critical tasks	Most technical applications

Table 5-2 Available Language Translators for RSX Use

	Operating Systems		
Languages	PDP-11 Family		
	RSX-11M	RSX-11M-PLUS	RSX-11S
MACRO Assembler	X	X	X
BASIC	X	X	
BASIC PLUS			
MU BASIC			
BASIC-PLUS-2	X	X	
COBOL	X	X	
FORTRAN			
FORTRAN IV	X	X	X
FORTRAN IV-PLUS	X	X	X
APL	X	X	
FORTRAN-77	X	X	
CPL			
Standard MUMPS			
RPG II	X	X	
CORAL 66	X		X
BLISS-32			
PASCAL			

LEARNING ACTIVITIES

1. READ Chapter 4, How to Do Work on the System, in the Introduction to RSX-11M/M-PLUS Manual.

2. DO Written Exercises 1 through 6 for this module.

ASSEMBLING/COMPILING

Figure 5-2 illustrates the process of translating a source file into machine instructions. Although a MACRO-11 program is used as an example, the process is similiar for a compiled language. The reference numbers in the following text refer to the numbers on the figure.

The language translator performs the following functions:

❶ Identifies symbols to be known to other modules (global symbols).

❷ Inserts macro definitions in MACRO-11 programs.

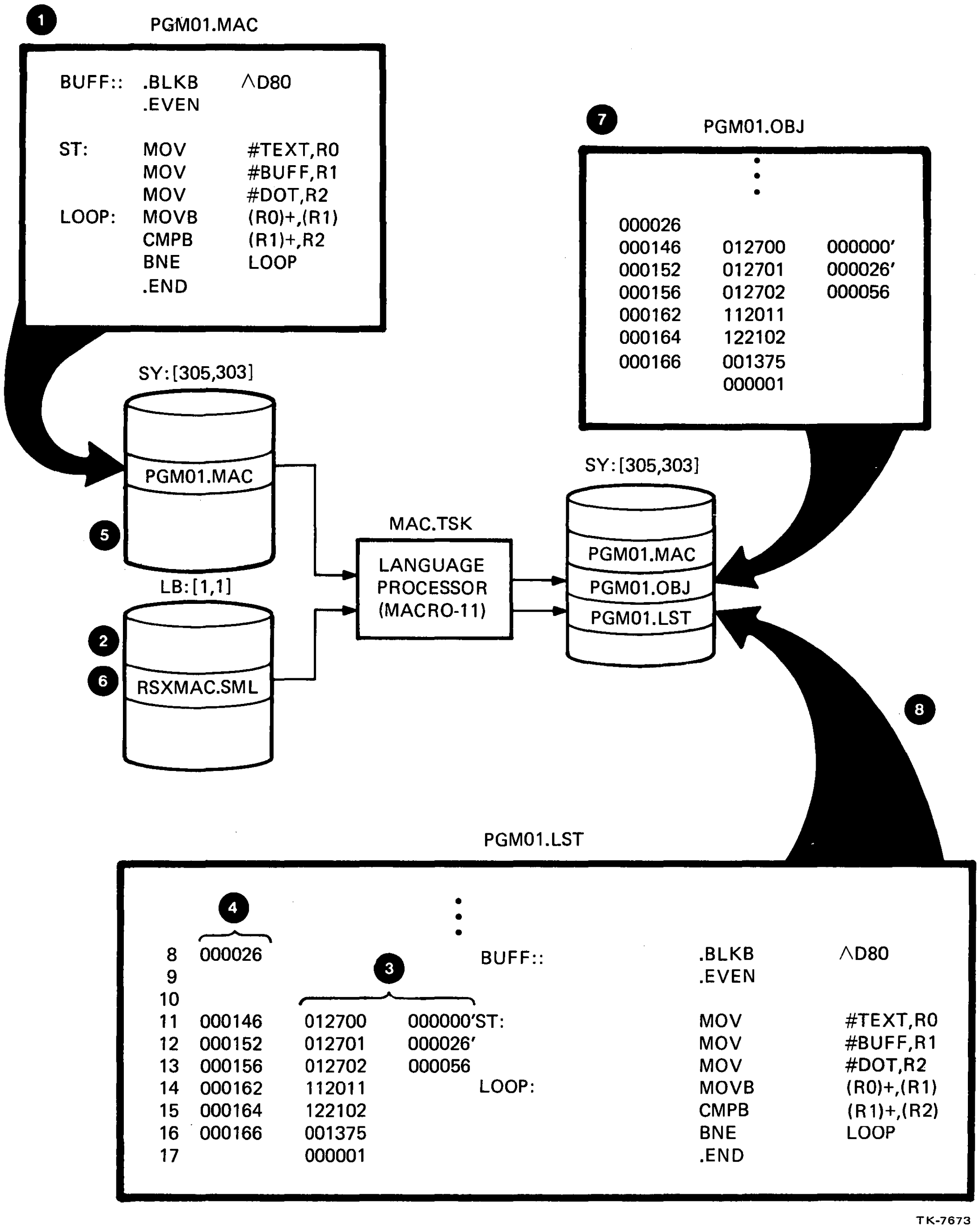
❸ Translates source language instructions into machine instructions.

❹ Assigns program addresses (relocatable virtual addresses) to each machine instruction.

Input to the language translator consists of one or more modules written in the source language (item ❺). In the case of MACRO-11, input may also come from macro definition libraries (item ❻). The MACRO-11 assembler automatically searches RSXMAC.SML for any undefined symbols that remain after processing all the input files.

Output from the language translator consists of an object module file and an optional listing file. The object file (item ❼) has a default file type of .OBJ, and contains the machine instructions and information needed in the next step of the program development process. The listing file (item ❽)has a default file type of .LST and contains the following:

- source instructions
- machine instructions (item ❸)
- relocatable virtual addresses (item ❹)
- error messages
- symbol listings
- assemble/compile statistics



```
BUFF::  .BLKB     ^D80
        .EVEN

ST:     MOV       #TEXT,R0
        MOV       #BUFF,R1
        MOV       #DOT,R2
LOOP:   MOVB      (R0)+,(R1)
        CMPB      (R1)+,R2
        BNE       LOOP
        .END
```

```
            .
            .
            .
000026
000146   012700   000000'
000152   012701   000026'
000156   012702   000056
000162   112011
000164   122102
000166   001375
         000001
```

```
                              .
                              .
                              .
 8  000026                        BUFF::      .BLKB     ^D80
 9                                            .EVEN
10
11  000146   012700   000000'ST:              MOV       #TEXT,R0
12  000152   012701   000026'                 MOV       #BUFF,R1
13  000156   012702   000056                  MOV       #DOT,R2
14  000162   112011          LOOP:            MOVB      (R0)+,(R1)
15  000164   122102                           CMPB      (R1)+,(R2)
16  000166   001375                           BNE       LOOP
17           000001                           .END
```

Figure 5-2 Translating a Program Source File Into Machine Language

MACRO-11 Language

The command format below shows how to invoke the MACRO-11 Assembler. The MACRO-11 Assembler takes MACRO source files and library files as input. The default input file type is .MAC. In specifying a series of input files, the last file specified cannot be a library file. You should place a library specification in front of the source file that requires it. After processing all the input files, the MACRO Assembler automatically searches the system macro library, RSXMAC.SML, for unsatisfied macro definitions. The last file name in the command is used as the default name for the output files. Table 5-3 lists some of the more frequently used command qualifiers. For more qualifiers, refer to Chapter 6 of the <u>RSX-11M/M-PLUS Command Language Manual</u>. Table 5-4 shows some examples of MACRO command usage.

MACRO-11 Assembler Command Format

$$>\underbrace{\text{MACRO}}_{\text{❶}}\underbrace{\text{/LIST}}_{\text{❷}}\ \underbrace{\text{USER}}_{\text{❸}}\underbrace{\text{/LIBRARY}}_{\text{❺}}\underset{\text{❹}}{,}\underbrace{\text{PROG}}_{\text{❸}}$$

❶ Command name

❷ Command qualifier

❸ Input file specification (default file type = .MAC)

❹ Input file specification delimiter

❺ Input file specification qualifier

Table 5-3 MACRO Command and File Qualifiers

User Wants	MACRO Command Qualifier to Use
A listing file	/LIST
A cross-reference file	/CROSS_REFERENCE
To specify an object file name other than the default	/OBJECT:ALPHA.REL
To specify a terminal width listing file	/NOWIDE
User Wants	**MACRO File Qualifier to Use**
To indicate that an input file is a library file	/LIBRARY

Table 5-4 Examples Using the MACRO Command

Command	Output			
	Object File	List File	Diagnostic Message Terminal	Equivalent MCR Command
>MACRO PROG	PROG.OBJ	None	Yes	>MAC PROG=PROG
>MACRO/LIST PROG	PROG.OBJ	PROG.LST	No	>MAC PROG,PROG/SP=PROG
>MACRO/LIST:FILE.DAT PROG	PROG.OBJ	FILE.DAT	No	>MAC PROG,FILE.DAT=PROG
>MACRO/OBJECT:FILE.OBJ PROG	FILE.OBJ	None	Yes	>MAC FILE.OBJ=PROG
>MACRO/NOOBJECT PROG	None	None	Yes	>MAC =PROG
>MACRO/LIST/NOWIDE PROG	PROG.OBJ	PROG.LST (8Ø character width listing)	No	>MAC PROG,PROG/LI:TTM/SP=PROG

Common Error Messages

The following are common MACRO-11 error messages. See pages 8-14 through 8-17 of the MACRO-11 Reference Manual for a description of these and other error messages.

```
MAC -- COMMAND SYNTAX ERROR

MAC -- ILLEGAL FILENAME

MAC -- ILLEGAL SWITCH

MAC -- I/O ERROR ON INPUT FILE

MAC -- I/O ERROR ON MACRO LIBRARY FILE

MAC -- I/O ERROR ON OUTPUT FILE

MAC -- OPEN FAILURE ON INPUT FILE

MAC -- OPEN FAILURE ON OUTPUT FILE
```

Notes on Example 5-1

Example 5-1 shows a section of a MACRO-11 listing file. The following comments are keyed to the example.

1. The title of the program

2. Assembly instructions that control the type of output. These may be overridden at assembly time by using the appropriate MACRO command qualifier.

3. Source line numbers. Diagnostic messages refer to these numbers.

4. Virtual addresses assigned to the machine instruction

5. The resulting machine instruction

6. The MACRO-11 source instruction

7. Comments to explain the logic of the program

8. A table of all symbols defined and/or referenced in the file. Asterisks in the address field denote unsatisfied references. Their definitions need to be satisfied at link time. Those symbols with an R following the address are relocatable addresses. Those symbols with a G are global symbols. A global symbol is one that may be referenced from another source file.

9. Diagnostic section

10. Assembly statistics, including the assembly time and command line (in MCR format) that invokes the assembly.

```
❶
HIYA    MACRO M1200  01-DEC-81 15:22  PAGE 1

❸   1                                          .TITLE  HIYA
    2                      ❷ .LIST   TTM
    3                        .NLIST  BEX
    4
    5
    6                  ; MACRO LIBRARY CALLS
    7
    8                        .MCALL  EXIT$S,QIOW$,DIR$
    9
   10 000000           INDPB:  QIOW$   IO.RLB,5,1,,IOST        ; REST TO BE FIL
   11 000030           OUTDPB: QIOW$   IO.WLB,5,1,,IOST,,<,,40>
   12
   13                  ; LOCAL EQUATES
   14
   15         000120   BSIZE=80.                       ; ACCEPTS NAMES UP TO 80
   16
   17
   18                  ; LOCAL DATA BUFFERS
   19
   20 000060     103   MSG1:   .ASCII  /COULD I HAVE YOUR NAME PLEASE?/
   21         000036   MSG1L=.-MSG1            ; THE LENGTH OF MSG1
   22
   23 000116     122   MSG2:   .ASCII  /RSX-11M-PLUS CALLING /
   24         000025   MSG2L=.-MSG2            ; THE LENGTH OF MSG2
   25 000143           BUFF:   .BLKB   BSIZE   ; SET UP BUFFER LENGTH = BSIZE
   26
   27 000263     111   OBUFF:  .ASCII  /IO ERROR WITH STATUS/<12><15>
   28 000311     104   I1:     .ASCII  /DSW =        /
   29 000326     111   I2:     .ASCII  /IOST =       /
   30         000060   OSIZ=.-OBUFF
   31
   32
   33                          .EVEN
   34 000344           IOST:   .BLKW   2
   35
   36
   37
   38                  ; MAIN PROGRAM
   39           ❺           ❻                        ❼
❹  40 000350  012700   HIYA:   MOV     #MSG1,R0        ; SET UP CALL
   41 000354  012701           MOV     #MSG1L,R1       ;  TO WRITE SUBROUTINE
   42 000360  004767           CALL    WRITE           ; OUTPUT MSG1
   43 000364  012700           MOV     #BUFF,R0        ; SET UP CALL
   44 000370  012701           MOV     #BSIZE,R1       ;  TO READ SUBROUTINE
   45 000374  004767           CALL    READ            ; READ NAME INTO BUFFER
   46 000400  012700           MOV     #MSG2,R0        ; SET UP CALL
   47 000404  012701           MOV     #MSG2L,R1       ;  TO WRITE SUBROUTINE
   48 000410  060201           ADD     R2,R1           ; LENGTH OF MSG2 + NAME
   49 000412  004767           CALL    WRITE           ; OUTPUT MSG2 WITH NAME
   50 000416                   EXIT$S                  ; LEAVE
   51                  ;+
   52                  ; WRITE - SUBROUTINE TO WRITE A MESSAGE TO THE TERMINAL
   53                  ;
   54                  ; INPUTS:
   55                  ;       R0 - ADDRESS OF STRING TO BE PRINTED ON SCREEN
   56                  ;       R1 - LENGTH OF STRING TO PRINT
   57                  ;

                              •
                              •
                              •
```

Example 5-1 Sample MACRO Assembly Listing (Sheet 1 of 2)

```
   HIYA    MACRO M1200  01-DEC-81 15:22  PAGE 1-3
8  SYMBOL TABLE

   BSIZE = 000120         MSG1     000060R        Q.IOLU= 000004
   BUFF    000143R        MSG1L  = 000036         Q.IOPL= 000014
   ERR1    000510R        MSG2     000116R        Q.IOPR= 000007
   HIYA    000350R        MSG2L  = 000025         Q.IOSB= 000010
   INDPB   000000R        OBUFF    000263R        READ    000454RG
   IOST    000344R        OSIZ   = 000060         WRITE   000424RG
   IO.RLB= ****** GX      OUTDPB   000030R        $CBDSG= ****** GX
   IO.WLB= ****** GX      Q.IOAE= 000012          $DSW  = ****** GX
   I1      000311R        Q.IOEF= 000006          $$$ARG= 000003
   I2      000326R        Q.IOFN= 000002          $$$OST= 000014


   . ABS.  000000     000
           000574     001
9  ERRORS DETECTED:  0

10 VIRTUAL MEMORY USED:  8774 WORDS  ( 35 PAGES)
   DYNAMIC MEMORY:  10316 WORDS  ( 39 PAGES)
   ELAPSED TIME:  00:00:06
   HIYA1,HIYA1=HIYA1
```

Example 5-1 Sample MACRO Assembly Listing (Sheet 2 of 2)

LEARNING ACTIVITIES

1. READ the following sections in the <u>RSX-11M/M-PLUS Command Language Manual</u>:

   - 6.1, Introduction
   - 6.2, Source Language

2. DO Written Exercises 7 through 1Ø for this module.

FORTRAN Languages

RSX-11M/M-PLUS systems support three FORTRAN source language compilers: FORTRAN IV, FORTRAN IV-PLUS and FORTRAN-77. Check with your system manager to determine which compiler (if any) your system has. Input source file(s) must contain source statements that comply with the rules of the compiler that you plan to use.

The command format below shows how to invoke a FORTRAN compiler. If you wish to use the FORTRAN IV-PLUS or FORTRAN-77 compiler, you must use the appropriate command qualifier to specify which one (Table 5-5). FORTRAN IV is the default FORTRAN compiler. The default input file type is .FTN for all three compilers. Output from the FORTRAN compiler can be an object file and/or a listing file.

Table 5-6 gives examples of how to construct a command to compile a FORTRAN source file called PROG.FTN, as well as the equivalent MCR commands.

FORTRAN Compiler Command Format

$$>\underbrace{\text{FORTRAN}}_{❶}\underbrace{\text{/F4P}}_{❷}\ \underbrace{\text{CALCA}}_{❸}\underset{❹}{,}\underbrace{\text{SINEX}}_{❸}\underset{❹}{,}\underbrace{\text{SINEY}}_{❸}$$

❶ Command name

❷ Command qualifiers

❸ File specification (default file type = .FTN)

❹ File specification delimiter

Table 5-5 FORTRAN Command Qualifiers

User Wants To	Command Qualifier To Use	Defaults
Specify use of FORTRAN IV-PLUS compiler	/F4P	
Specify use of FORTRAN-77 compiler	/F77	
Specify use of FORTRAN IV compiler	/FOR	/FOR
Control the generation of a listing file	/LIST /NOLIST	/NOLIST
Control the generation of an object file	/OBJECT /NOOBJECT	/OBJECT
Specify listing file to include binary machine code and diagnostics	/MACHINE_CODE	/NOMACHINE_CODE

Table 5-6 Examples Using The FORTRAN Command

DCL Command	Compiler	Object File	List File	List File Contents	Diagnostic Message at Terminal	Equivalent MCR Commands
>FORTRAN PROG	FORTRAN IV	PROG.OBJ	None		Yes	FOR PROG=PROG
>FORTRAN/F4P PROG	FORTRAN IV-PLUS	PROG.OBJ	None		Yes	F4P PROG=PROG
>FORTRAN/F77 PROG	FORTRAN 77	PROG.OBJ	None		Yes	F77 PROG=PROG
>FORTRAN/LIST PROG	FORTRAN IV	PROG.OBJ	PROG.LST	Source Map Diagnostics	No	FOR PROG,PROG/SP/LI:3=PROG
>FORTRAN/NOOBJECT PROG	FORTRAN IV	None	None		Yes	FOR =PROG
>FORTRAN/MACHINE CODE PROG	FORTRAN IV	PROG.OBJ	PROG.LST	All	No	FOR PROG,PROG/SP/LI:7=PROG

Common FORTRAN Error Messages

The following are common FORTRAN error messages. See Appendix C in the FORTRAN IV User's Guide for a complete description of all error messages.

```
FOR -- BAD SWITCH

FOR -- BAD SWITCH VALUE

FOR -- ERROR READING SOURCE FILE

FOR -- ERROR WRITING LISTING FILE

FOR -- ERROR WRITING OBJECT FILE

FOR -- OPEN FAILED FOR FILE

FOR -- SYNTAX ERROR

FOR -- TOO MANY INPUT FILES

FOR -- TOO MANY OUTPUT FILES

FOR -- WILD CARD NOT ALLOWED
```

FORTRAN Compiler Listing

Example 5-2 is a sample of a FORTRAN compiler listing. The listing file is optional compiler output. To generate the file, the /LIST qualifier must be specified in the FORTRAN command line. The default file type is .LST. The following comments are keyed to the example.

1. Compiler name and version number, time and date of compile, and the compiler command (in MCR) invoking the compilation.

2. Source statement line numbers. Error diagnostics refer to these numbers.

3. FORTRAN source statements.

4. Diagnostics indicating problem areas in the code.

5. Storage map providing information on symbols, common blocks, arrays, and subroutines.

```
❶ FORTRAN IV      V02.2-1          TUE 01-DEC-81 15:51:32          PAGE 001
  PROG,PROG=PROG

❷ 0001      ❸ COMMON/COM/I(10)
  0002         DO 100, J=1,10
  0003         READ(5,900)I(J)
  0004         CALL SUB(J,M)
  0005         WRITE(5,901)M
  0006  100    CONTINUE
  0007         WRITE(5,902)
  0008         CALL EXIT
  0009         SUBROUTINE SUB(J,K)
  0010         COMMON/COM/I(10)
  0011         K=2*I(J)
  0012         RETURN
  0013  900    FORMAT(I3)
  0014  901    FORMAT(' ',I6)
  0015  902    FORMAT(' THIS IS THE END')
  0016         END


❹ FORTRAN IV DIAGNOSTICS FOR PROGRAM UNIT .MAIN.

  IN LINE 0002, WARNING:  POSSIBLE MODIFICATION OF INDEX "J"
  IN LINE 0009,  ERROR:   SUBPROGRAM STATEMENT MUST BE FIRST
  IN LINE 0010,  ERROR:   MULTIPLE DECLARATION FOR VARIABLE "I"


❺ FORTRAN IV STORAGE MAP FOR PROGRAM UNIT .MAIN.

  LOCAL VARIABLES, .PSECT $DATA, SIZE = 000006 (     3. WORDS)

  NAME    TYPE  OFFSET      NAME    TYPE  OFFSET      NAME    TYPE  OFFSET
  J       I*2   000000      K       I*2   000004      M       I*2   000002

  COMMON BLOCK /COM   /, SIZE = 000024 (    10. WORDS)

  NAME    TYPE  OFFSET      NAME    TYPE  OFFSET      NAME    TYPE  OFFSET
  I       I*2   000000

  LOCAL AND COMMON ARRAYS:

  NAME      TYPE   SECTION OFFSET  ------SIZE----- DIMENSIONS
  I       I*2      COM     000000  000024 (    10.) (10)

  SUBROUTINES, FUNCTIONS, STATEMENT AND PROCESSOR-DEFINED FUNCTIONS:

  NAME    TYPE   NAME    TYPE   NAME    TYPE   NAME    TYPE   NAME    TYPE
  EXIT     R*4   SUB      R*4
```

Example 5-2 Sample FORTRAN Compiler Listing

LEARNING ACTIVITIES

1. READ Section 6.2.3, FORTRAN, in the RSX-11M/M-PLUS Command Language Manual.

2. DO Written Exercises 11 through 15 for this module.

TASK-BUILDING/LINKING

Overview

Figure 5-3 illustrates the next step in the program development process, linking object modules into an executable task image. This step is called task-building or linking. A special system utility, the Task Builder, performs this function.

Generally, a task image will be built using many object modules that may reside in more than one object file. In an object module, a reference can be made to a symbol or routine located in another object module. The symbol or routine can also be located in a special file called an object library. It is the Task Builder's function to bring together all the object modules from object files and libraries, and join them together logically into one output file called the task image file (item ❶ in Figure 5-3). The Task Builder also resolves references to symbols and routines in different modules by placing the appropriate address in the instruction where the reference was made. In this way, all the separate modules that make up a task are linked together.

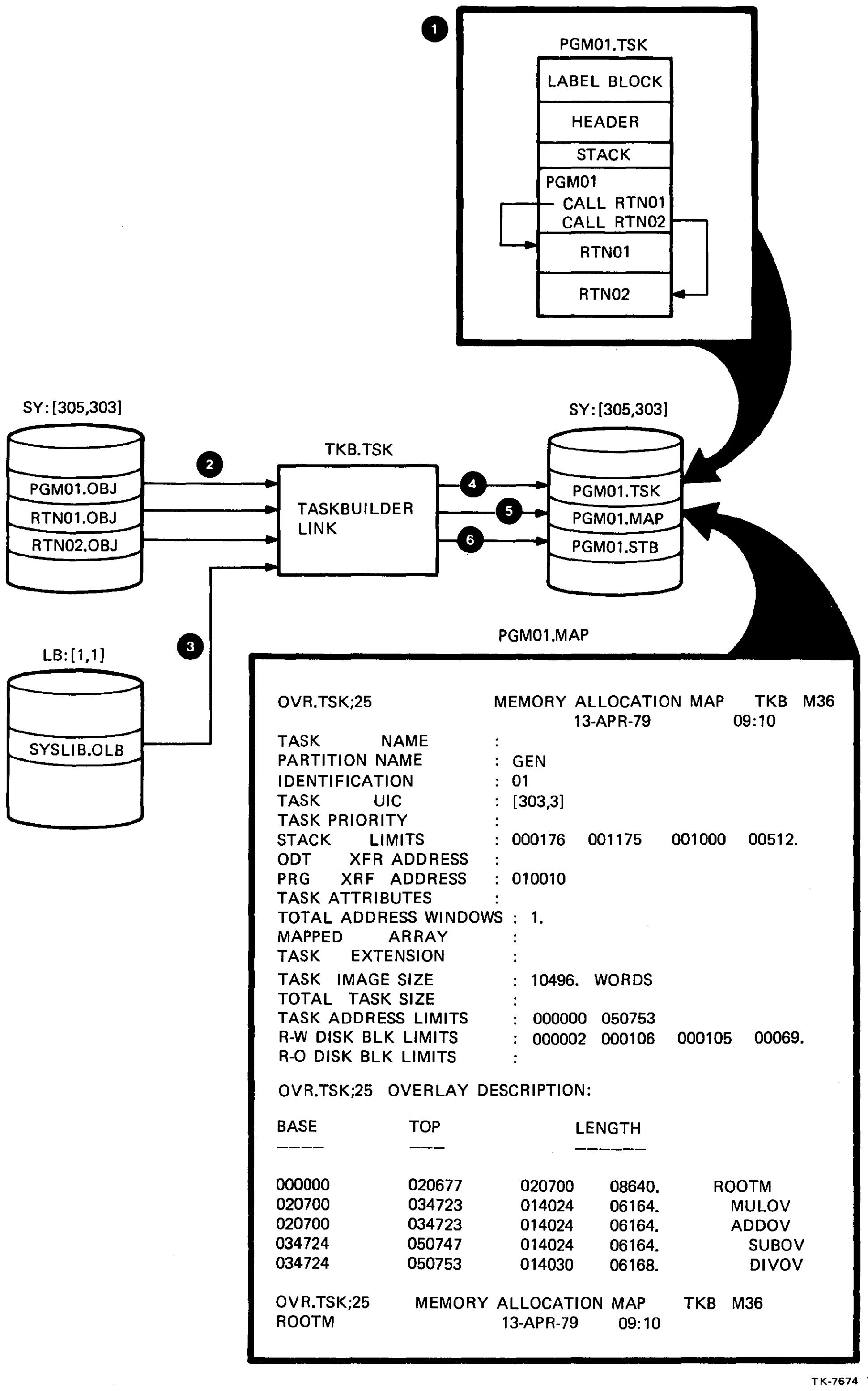
Input to the Task Builder (item ❷) are the object files that are output from the assembler or a compiler. The default input file type is .OBJ. Another source of input is user or system object libraries. These libraries contain code for frequently used routines that have been developed and debugged. If after processing all the input files there remain symbols that have not been defined, the Task Builder automatically searches the system library looking for the code. This library, SYSLIB.OLB, is located in LB:[1,1] (item ❸). If the search is successful, the Task Builder includes in the task the object module from the library.

Task Builder output can include a task image file, a map file, and a symbol definition file. Each output file is optional, and the Task Builder will generate any combination of the three. The task image file (item ❹) contains the actual machine instructions that load into memory and execute in the CPU. The Task Builder creates the file in the special format shown in Figure 5-4. (See the notes that refer to this figure.)

Another output file of the Task Builder, and a useful tool for the programmer, is the MAP file (item ❺). This file contains information on the allocation of address space in the task image, the program sections created in the task image, and the module of origin and the value of each global symbol. We will discuss this file in more detail later.

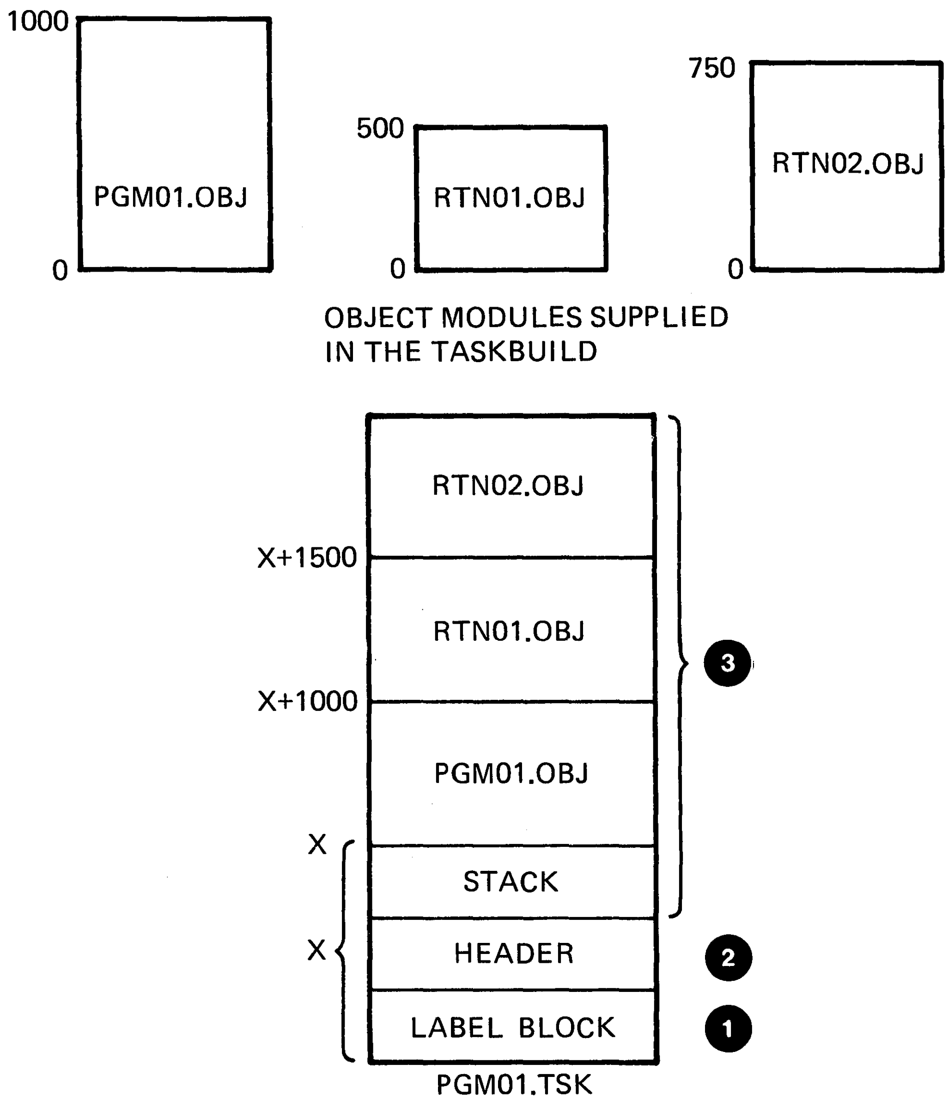
The symbol table file is the third output file (item ❻). This file contains global symbols defined in the task, and their virtual or relocatable addresses. It is in a format suitable for reprocessing by the Task Builder. You specify this file when you are building a resident library or common. Resident libraries and commons are discussed in the Task Builder Manual, and are advanced topics covered in the RSX-11M/M-PLUS Programmer course. They will not be discussed here.

Table 5-9 lists some of the characteristics of a task that the operating system must know before the task can be run. These characteristics, supplied at task-build time, are included in the label and header portions of the task image. If these are not specified, the task is built with the default values listed in the table. In Module 8, we discuss building a task with different characteristics.



```
OVR.TSK;25                    MEMORY ALLOCATION MAP     TKB  M36
                                   13-APR-79           09:10
TASK         NAME           :
PARTITION NAME              : GEN
IDENTIFICATION              : 01
TASK         UIC            : [303,3]
TASK PRIORITY               :
STACK        LIMITS         : 000176   001175    001000   00512.
ODT     XFR ADDRESS         :
PRG     XRF  ADDRESS        : 010010
TASK ATTRIBUTES             :
TOTAL ADDRESS WINDOWS       :  1.
MAPPED        ARRAY         :
TASK      EXTENSION         :
TASK  IMAGE SIZE            :  10496.  WORDS
TOTAL  TASK SIZE            :
TASK ADDRESS LIMITS         :  000000  050753
R-W DISK BLK LIMITS         :  000002  000106   000105   00069.
R-O DISK BLK LIMITS         :

OVR.TSK;25  OVERLAY DESCRIPTION:

BASE          TOP              LENGTH
----          ---              ------

000000        020677     020700   08640.      ROOTM
020700        034723     014024   06164.        MULOV
020700        034723     014024   06164.        ADDOV
034724        050747     014024   06164.          SUBOV
034724        050753     014030   06168.          DIVOV

OVR.TSK;25        MEMORY ALLOCATION MAP      TKB  M36
ROOTM                   13-APR-79     09:10
```

Figure 5-3 Task-Building an Executable Module



TK-7677

Figure 5-4 Task Image Structure

Notes on Figure 5-4

The following comments are keyed to the figure.

❶ Label Block

Contains task information necessary for running the task (name, partition, size, priority). The INSTALL command uses this information to create a Task Control Block (TCB) in the STD, and to initialize the task header.

❷ Header

Contains information the Executive uses to run the task. Also provides a storage area for saving essential data when the task is checkpointed. The INSTALL command initializes the part of the header not initialized by the Task Builder.

❸ Stack, Code and Data

This part of the task image includes the task stack and linked modules of code and data. The task header along with the stack, code and data are loaded into memory when a task is requested to run.

Task Builder Command Format

The command format below shows the basic structure used to invoke the Task Builder. Input to the Task Builder consists of object files and library files. The default input file type for object modules is .OBJ. Library file specifications must include the /LIBRARY qualifier, and are of default type .OLB. The default output file types include .TSK for the task image file, .MAP for the map file, and .STB for the symbol table file.

The LINK command can take qualifiers that modify the linking of modules. The most frequently used qualifiers are listed in Table 5-7. The RSX-11M/M-PLUS Command Language Manual describes these and others in more detail. Examples of the LINK command and equivalent MCR commands are shown in Table 5-8.

$$\texttt{>}\underbrace{\texttt{LINK}}_{\text{❶}}\underbrace{\texttt{/MAP}}_{\text{❷}}\ \underbrace{\texttt{ROOT}}_{\text{❸}}\underset{\text{❹}}{,}\underbrace{\texttt{RTN1}}_{\text{❸}}\underset{\text{❹}}{,}\underbrace{\texttt{USOBJ}}_{\text{❸}}\underbrace{\texttt{/LIB}}_{\text{❺}}$$

❶ Command name

❷ Command qualifier

❸ Input file specification

❹ File specification delimiter

❺ File specification qualifier

Table 5-7 LINK Command and File Qualifiers

User Wants To	LINK Command/File Qualifier to Use	
Produce a cross-reference in map file	/CROSS_REFERENCE	C
Produce a map file	/MAP	C
Use the fast Task Builder	/FAST	C
Name the task image file other than the default	/TASK:VIP.TSK	C
Define options other than the default	/OPTIONS	C
Build the task to be checkpointable	/CHECKPOINT	C
Include ODT in the task image	/DEBUG	C
Indicate that an input file is a library file	/LIBRARY	F
Indicate that an input file is an overlay descriptor file	/OVERLAY_DESCRIPTION	F

Table 5-8 Examples Using the Task Builder

Command	Task Builder Version	Task Image File	Map File	Task Name	Priority	Diagnostic Messages on Terminal?	Equivalent MCR Commands
>LINK PROGRAM	Default	PROGRA.TSK	None	PROGRA	5Ø	Yes	TKB PROGRAM=PROGRAM
>LINK/MAP PROG	Default	PROG.TSK	PROG.MAP	PROG	5Ø	Yes	TKBPROG,PROG/SP=PROG
>LINK/FAST PROG	Fast	PROG.TSK	None	PROG	5Ø	Yes	FTB PROG=PROG
>LINK/OPTION PROG OPTIONS?TASK=XXX OPTIONS?PRI=1ØØ OPTIONS?<CR>	Default	PROG.TSK	None	XXX	1ØØ	Yes	>TKB TKB>PROG=PROG TKB>/ ENTER OPTIONS: TKB>TASK=XXX TKB>PRI=1ØØ TKB>//

Table 5-9 Standard Task Builder Defaults

Option	Default
Task Name	First 6 characters of first file name
Task UIC	Terminal UIC
Task Priority	5Ø
Task Partition	GEN
Number of Active Files	4
Logical Units	6 (7 for FORTRAN)
Device Assignment	SYØ:1:2:3:4,TIØ:5,CLØ:6

Common Task Builder Error Messages

The following is a list of common Task Builder error messages. See Appendix F of the Task Builder Manual for a complete description of all error messages.

ALLOCATION FAILURE ON FILE file-name

COMMAND SYNTAX ERROR

FILE file-name HAS ILLEGAL FORMAT

ILLEGAL FILENAME

INVALID PARTITION/COMMON BLOCK SPECIFIED

I/O ERROR ON INPUT FILE file-name

I/O ERROR ON OUTPUT FILE file-name

LOOKUP FAILURE ON FILE file-name

OPEN FAILURE ON FILE file-name

REQUIRED INPUT MISSING

TASK IMAGE FILE file-name IS NONCONTIGUOUS

n UNDEFINED SYMBOLS SEGMENT seg-name

Interpreting a Task Map

Example 5-3 is a sample of the Task Builder map file. It tells which modules were used to build the task, in what order they were linked together, and at what virtual address they begin. It also gives other information that is useful when debugging a program. Read the notes for more details on the example. For further information, refer to the discussion of the /SH switch in Chapter 1Ø of the Task Builder Manual. Not all of this information is of value to you now. It is important, however, that you know what the map file is and what type of information it contains.

```
HIYA1.TSK   Memory allocation map  TKB M39.7D       Page 1
                      1-DEC-81   15:23


Partition name : GEN ❶
Identification : 0351
Task  UIC      : [305,303] ❷
Stack    limits: 000254 001253 001000 00512. ❸
PRG xfr address: 001624 ❹
Total address windows: 1.
Task  image  size  : 640. words ❺
Task address limits: 000000 002377 ❻
R-W disk blk limits: 000002 000004 000003 00003.

*** Root segment: HIYA1 ❼


R/W mem  limits: 000000 002377 002400 01280. ❽
Disk blk limits: 000002 000004 000003 00003.


Memory allocation synopsis:

Section                                ❾                Title  Ident  File
-------                                                 -----  -----  ----
. BLK.:(RW,I,LCL,REL,CON)       001254 001012 00522.
                                001254 000574 00380.    HIYA          HIYA1.OBJ;1 ❿
$$RESL:(RO,I,LCL,REL,CON)       002266 000112 00074.


Global symbols:

READ   001730-R  WRITE  001700-R


                 ⓫
*** Task builder statistics:

    Total work file references: 1099.
    Work  file  reads: 0.
    Work  file writes: 0.
    Size of core pool: 7086. words (27. pages)
    Size of work file: 1024. words (4. pages)

    Elapsed time:00:00:03
```

Example 5-3 Sample Task Map

Notes on Example 5-3

1. The partition in which the task will be loaded.

2. The UIC under which the task will be run for time-based schedule requests. This determines which files the task can access.

3. The low and high limits of the task stack, followed by its length in octal and decimal bytes.

4. The virtual address at which the program will begin executing.

5. The task image size in decimal words.

6. The lowest and highest virtual address allocated to the task.

7. The name of the root segment, in this case, the task name.

8. From left to right:

   Beginning virtual address of root segment
   Virtual address of the last segment byte
   Length in octal bytes
   Length in decimal bytes

9. From left to right:

   Program section name
   Program section attributes
   Starting virtual address of the section
   Length in octal bytes
   Length in decimal bytes

10. This line contains the same first fields as 9 and also a title and identification number obtained from the source code, and the name of the file containing the source code.

11. This section contains statistics that are of no interest to the general user.

LEARNING ACTIVITIES

1. READ the following sections in the RSX-11M/M-PLUS Command Language Manual:

   - 6.4, Linking the Task
   - 6.4.1, Introduction to the Link Command
   - 6.4.2, Link

2. DO Written Exercises 16 through 19 for this module.

RUNNING THE TASK

After linking, you run a task by supplying the task image file specification as a parameter to the RUN command. The operating system will then go through the procedure of installing, running and removing your task, as shown in Figure 5-5. This is the most frequently used version of the RUN command, as most tasks are not permanently installed in the STD. We will discuss the other versions of the RUN command in Module 8, Controlling Task Execution.

LEARNING ACTIVITIES

1. READ the following sections in the RSX-11M/M-PLUS Command Language Manual:

   - 7.1, Task Installation and Execution
   - 7.1.1, Task Naming
   - 7.2, Introduction to the RUN Command
   - 7.3, Abort Command

2. DO Written Exercises 2Ø and 21 for this module.

3. DO the Lab Exercises for this module.

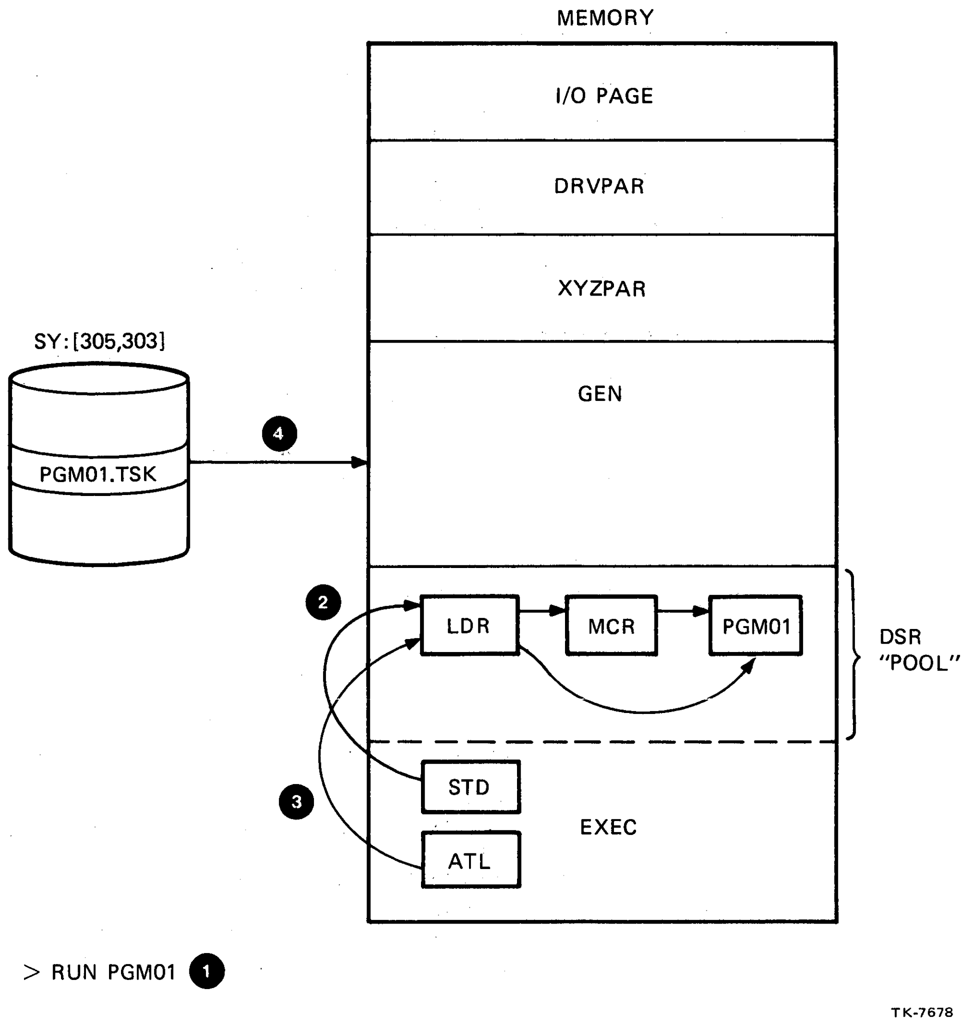


Figure 5-5 Requesting a Task to Run

Notes on Figure 5-5

The following comments are keyed to the figure.

1. User requests PGM01 to run
2. PGM01 added to the list of known tasks
3. PGM01 added to the lists of active tasks
4. PGM01 loaded into memory ready-to-run

On completion, entries in STD and ATL for PGM01 are removed.

USING THE EDITOR EFFECTIVELY

INTRODUCTION

EDT, the DEC editor, creates and modifies files. As you grow more accustomed to using it, you will appreciate many additional features that make it quicker to use. A number of these additional features is presented in this module.

OBJECTIVES

1. Move or copy sections of text from one buffer to another.
2. Read in text from another file.
3. Write a subset of the text to an output file.
4. Write a Macro to execute EDT commands.
5. Set editor characteristics for special purposes.
6. Use character mode with a keypad.

RESOURCES

1. EDT Editor Manual

USING THE EDITOR EFFECTIVELY

There are many features of the editor that will make your editing sessions easier. Such features as alternate buffers, user-defined macros, and the ability to set terminal characteristics, do character searches, and perform cut and paste operations can shorten the time it would normally take to create a file. Suppose, for example, that you had the responsibility of writing a user's manual for a piece of hardware. After capturing your first thoughts using the editor, you decide to order the paragraphs differently. The cut and paste feature of character mode allows you to select a paragraph, phrase or word and easily move it to another location. You can guess how long an editing session would be if you had to delete the lines and retype them in their new location! This is one of many features discussed in this module that will make your editing session easier.

Editor Buffers

From our previous discussion on the editor, you learned that the editor creates workspaces called buffers. At the start of an editing session, EDT automatically creates two standard buffers for use during the editing session. The first buffer, called MAIN, is for general editing. The second buffer, called PASTE, is for the cut and paste operations done in character mode.

EDT also allows for the creation of additional buffers as they are needed.

Buffer Use

You may use these buffers in the following ways:

- To divide one or more files into sections
- To include part or all of another file
- To create another file from part or all of the text in a buffer

Creating a Buffer

You create a buffer by naming it, i.e., by referencing a 1-3Ø character name preceded by an equal sign. (See example below.) You can make the reference on a line by itself, or in a range specification. The reference automatically creates the buffer and places your cursor at the beginning of the buffer. For example, issuing the command =NEWBUF alone on a line creates a buffer with the name NEWBUF. This becomes your current buffer, and the current line is now pointing to the beginning of the buffer. You can also create a buffer while performing editor operations such as MOVE or COPY. For example, you may reference a buffer name in a range specification in the COPY command. EDT will first create the buffer and then copy the lines into the buffer. Once again, the buffer is now the current buffer and the current line is in the new buffer, not MAIN.

```
Example:  *=NEWBUF
```

Referring to a Buffer

Part of a range specification is the buffer name. If you do not supply a buffer name in a range specification, the editor assumes you are referring to the current buffer. If you wish to specify another buffer, it is done this way:

```
*MOVE 1:3 TO =DOC 5
```

This example moves lines numbered 1 through 3 from the current buffer to the line before line number 5 in the buffer named DOC.

You determine the current buffer by using the SHOW BUFFER command:

```
*SHOW BUFFER
=NEWBUF  2Ø LINES
MAIN    26Ø LINES
PASTE     Ø LINES
```

The buffer preceded by the equal sign is the current buffer.

LINE MODE FEATURES

Searching for a Character String

The line mode search function is useful for locating a word or character string when the line number of the string location is not known. The search direction can be either forward or backward from the current line position. It is also possible to search for all occurrences of a string within a range of lines, or through the complete file.

When you enclose a character string with single or double quotes and type a carriage return, the editor searches forward in the file to locate the first occurrence of the string. Table 6-1 lists the formats for this and other types of character searches.

Table 6-1 String Search Commands in Line Mode

User Wants To	Example
Search ahead in a file to locate the first occurrence of a string	'RSX-11M' or "PROGRAM"
Search back in a file to locate the first occurrence of a string	-'RSX-11M' or -"PROGRAM"
Locate all lines containing a string	%ALL 'EDITOR'
Locate all lines within a range of lines containing a string	15:60 ALL 'EDITOR'

Reading and Writing Files

The editor has additional file-handling facilities to aid the user during the editing session. You will find the INCLUDE command useful when you want to include the contents of another file in the one you are creating. This is done by specifying INCLUDE with the name of the file to be included, and giving a range specification to indicate where it is to be placed in the new file. One use of this command is to include a file in an alternate buffer, and search the alternate buffer for specific lines of text. Once located, the lines can be copied into the correct position in the MAIN buffer. There is no need to clean up the alternate buffer as the exit command saves only the contents of MAIN.

The WRITE command will create a file containing all or part of the text that you are editing. Once the file is created you may continue work in the editing session. The /SEQUENCE qualifier of the WRITE command allows you to save the editing line numbers along with the text. These line numbers will not be displayed when you type the file to the terminal, but will appear when you print the file on a line printer. WRITE is also useful when you wish to break up a file into many files. Suppose you have a FORTRAN source file containing many subroutines. You can divide that file into one file per subroutine by issuing the WRITE command the appropriate number of times with the appropriate range specifications.

The PRINT command allows you to create and print files in one operation. When you issue this command with a file name and range specification, the editor will create a file containing the lines of text specified, and then automatically print the file for you.

Table 6-2 lists examples of the commands used for each of these features.

Table 6-2 Line Mode Commands to Read and Write Files from Within EDT

User Wants To	Command Example
Include text from another file	*INCLUDE CLAIM.TXT =BUF 90
Save a range of lines in a file	*WRITE SPECIAL.TXT /SEQ
Print a range of lines	*PRINT NOTICE.TXT

Generating EDT Macros

A Macro

The DEFINE MACRO command is another useful EDT feature. Before we discuss the command, we need to define the term macro in relation to the editor. A macro is a sequence of editing commands with a name assigned to it. You can issue the macro name just like a line mode command. Every time you do so the editor executes the sequence of commands. This facility allows you to extend the range of commands available for your use in EDT.

Defining a Macro

DEFINE MACRO establishes the sequences of commands to a macro name. When you issue this command, the editor sets up a buffer with the name you supply in the DEFINE MACRO command. Using the INSERT command, you can enter the EDT commands of your choice into the buffer. This macro name then becomes a part of the EDT command list for the duration of the editing session, and can be used just like any other command. It can also be used in character mode by using the GOLD COMMAND function. If you want to save this macro for use during future editing sessions, perform the following steps:

1. Use WRITE to put the contents of the macro buffer into a file.
2. In your next editing session, use DEFINE MACRO again to set up the buffer.
3. Then use INCLUDE to read the file created above into the buffer created by DEFINE MACRO.

Notes on Example 6-1

Example 6-1 shows the process of creating and executing a macro in line mode. The following notes are keyed to the example.

❶ Command to define the macro. The name of the macro is DOC.

❷ SHOW BUFFER command issued to show all existing buffers.

❸ Command to make DOC the current buffer so that text is stored in it, not MAIN.

❹ Enter input mode and type the editor commands.

❺ Set the current buffer back to MAIN (which is empty at this time).

❻ Execute the macro.

❼ Display the contents of MAIN.

❽ Execute the macro again.

❾ Display the contents of MAIN again.

❿ Save the macro DOC in a file called DOC.MAC.

```
    >
    >
    >EDIT/EDT MANUAL.TXT
    Input file does not exist
    [EOB]
1   *DEFINE MACRO DOC
2   *SH BUF
     DOC     0        lines
    =MAIN    0        lines
     PASTE   0        lines
3   *=DOC
    [EOB]
4   *I
                      INSERT E;;CHAPTER
                      INSERT E;;
                      INSERT E;;SECTION
                      INSERT E;;
                      INSERT E;;
                      ^Z
    [EOB]
5   *=MAIN
    [EOB]
6   *DOC
    *=MAIN
        1             ;CHAPTER
        2             ;
7       3             ;SECTION
        4             ;
        5             ;
    [EOB]
8   *DOC
    *=MAIN
        1             ;CHAPTER
        2             ;
        3             ;SECTION
        4             ;
        5             ;
9       6             ;CHAPTER
        7             ;
        8             ;SECTION
        9             ;
       10             ;
    [EOB]
10  *WRITE DOC.MAC =DOC
    DR0:[305,303]DOC.MAC;1 5 lines
    *EXIT
    DR0:[305,303]MANUAL.TXT;1 15 lines
```

Example 6-1 Defining a Macro

Setting Editor Parameters

There are many operating characteristics of EDT that you can set for an editing session. These characteristics control how information is displayed at your terminal. The SET and SHOW commands allow you to alter the default characteristics, or to display their settings. Table 6-3 lists some parameters you may find useful. Table 6-4 summarizes other useful line mode commands.

Table 6-3 SET and SHOW Commands

User Wants To	SET Parameter	Default	SHOW Parameter
Set the editor for his terminal type (if you want to override what EDT gets from the operating system).	TERMINAL type	From operating system	TERMINAL
Set search matches exactly (e.g., m is not equal to M)	SEARCH EXACT	GENERAL	none
Set window size (in character mode)	LINES length SCREEN width	LINES 22 SCREEN 8Ø	SCREEN only
Display charac- ters on succeed- ing lines for lines longer than the screen width (in character mode)	NOTRUNCATE	TRUNCATE	none
Wrap to a new line with the last word if a specified number of characters (e.g., screen width) is exceeded (in character mode)	WRAP n	NOWRAP	none
Set the first tab stop other than 8 characters out (in charac- ter mode)	TAB n	TAB 8, same as NOTAB	none

Additional Line Mode Commands

Table 6-4 Other Useful Line Mode Commands

User Wants To	Line Mode Commands To Use
Delete all lines in a buffer	CLEAR
Define or redefine a function for a character mode key	DEFINE KEY
Reformat a line when line width has been changed	FILL
Delete a range of lines and replace with other lines	REPLACE
To replace next occurrence of an already searched for string with another string	Substitute Next

The Startup Command File

When you start your editing session, EDT automatically searches your UFD for a file called EDTINI.EDT. This file, if it exists, is then read by EDT for commands that would customize the editing session. If this file does not exist, EDT initializes the editing session in the standard way.

You create this file, called a startup command file, using an editor. It contains line mode commands that you want to have executed automatically when EDT is started. Such commands as SET, DEFINE MACRO, DEFINE KEY and INCLUDE may be used to create your own editing environment. By using SET MODE CHANGE in the file, EDT will automatically change to character mode every time you start the editor. The process of using an existing EDT macro discussed before in the section, Defining a Macro, can be done automatically by putting the DEFINE MACRO and INCLUDE commands in the startup command file. Then each time you use the editor, the macro would be available without your having to recreate it.

If you have an EDTINI.EDT file in your UFD and wish to edit a file without first initializing EDT, use the /NOCOMMAND qualifier shown below. EDT will start up normally without reading the start up command file.

```
EDIT/EDT/NOCOMMAND filespec
```

CHARACTER MODE FEATURES

Tables 6-5 and 6-6 summarize some of the available features of the character mode.

Additional Keypad Functions

Table 6-5 Additional Commands to Move the Cursor

User Wants To	Key Name
Move cursor 16 lines in current direction	SECTION
Move cursor to beginning of buffer	TOP
Move cursor to end of buffer	BOTTOM
Move cursor to top of next or preceding page	PAGE
(VT1ØØ only) Move cursor 1 character in current direction	CHAR

LEARNING ACTIVITIES

1. READ the following in the EDT Editor Manual:

   - Chapter 6, Line Numbers, Text Buffers and Ranges

     - Section on Text Buffers

   - Chapter 9, Set and Show Commands

   - Chapter 7, Line Editing

     - Sections on Clear, Define Key, Replace and Substitute Next, Define Macro

   - Chapter 4, The Command Line and Startup Command Files

2. DO Lab Exercises 1 through 1Ø for this module.

Other Character Mode Functions

Table 6-6 Additional Character Mode Function Keys

User Wants To	Character Mode Command
Append a string of text to current contents of PASTE buffer	APPEND
Change a character or string of characters from uppercase to lowercase, or vice versa	CHNGCASE
Redefine a key function	CTRL/K
Fill a line with text when changing line width from one length to another	FILL (CTRL/F on VT52s)
Insert a nonprinting ASCII character	SPECINS
Delete a selected text string and replace with contents of PASTE buffer	REPLACE
Delete all occurrences of a selected text string and replace with contents of PASTE buffer	SUBS

String Searches in Character Mode

Character mode also has string search capability. This is done by the following procedure:

1. Position the cursor at one end of a range of lines to be searched.
2. User types: $\boxed{\text{ADVANCE/BOTTOM}}$ or $\boxed{\text{BACKUP/TOP}}$ to establish the direction of the search
3. User types: $\boxed{\text{GOLD}}\boxed{\text{FINDNEXT/FIND}}$
4. EDT responds: SEARCH FOR:
5. User types: $\boxed{\text{THIS IS ENTER/SUBS}}$

The string is remembered, and additional searches can be done using the FINDNEXT key:

1. User types: $\boxed{\text{FINDNEXT/FIND}}$

Cutting and Pasting Text

Cutting and pasting text in character mode is equivalent to the copy command in line mode. However, because you are in character mode, you can select a section of text that begins in the middle of a line, and move it to a location in the middle of another line. The text you select is stored in the PASTE buffer; more text can be appended to this buffer using the APPEND function. The following procedure shows how to use the CUT and PASTE feature:

1. Move the cursor to one end of the portion of text to be moved.
2. Type $\boxed{\text{SELECT/RESET}}$
3. Move the cursor to the other end of the portion of text to be moved.
4. Type $\boxed{\text{CUT/PASTE}}$. The text is now saved in the PASTE buffer.
5. Move the cursor to the desired new location.
6. Type $\boxed{\text{GOLD}}$ $\boxed{\text{CUT/PASTE}}$

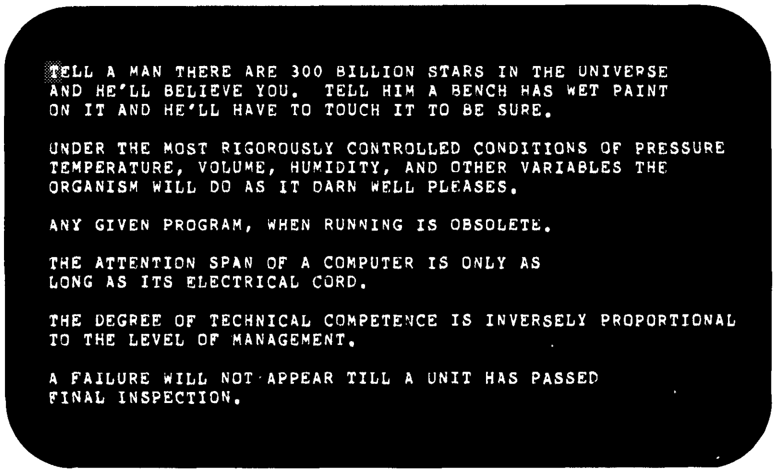
Repeating Functions Automatically

Many times, you would like to delete more than one consecutive word or character at a time. You can do this by typing the DEL L or DEL W key an appropriate number of times, but there is another way that requires fewer key strokes. Suppose you want to delete the next five words. After ensuring that the ADVANCE direction is set, you type the sequence of keys shown in the example. Note that the 5 key is the one on the standard keyboard and not the one on the keypad. The 5 on the keypad already has a special meaning, other than the number 5.

Example:

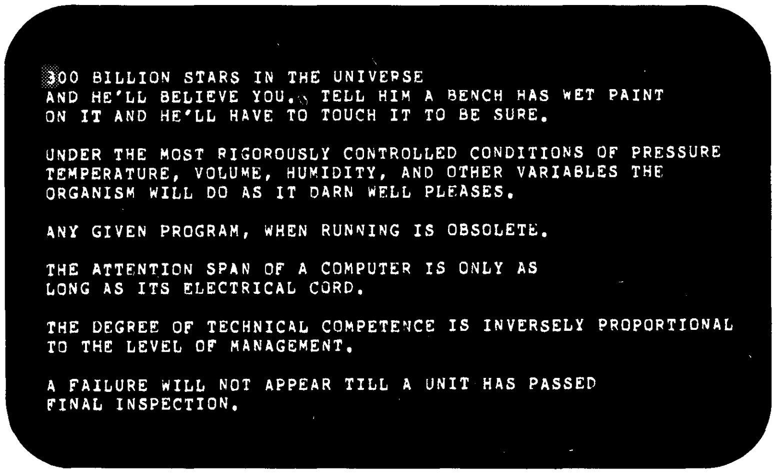
Keys : (GOLD) 5 (DEL WORD)

Before Command:



```
TELL A MAN THERE ARE 300 BILLION STARS IN THE UNIVERSE
AND HE'LL BELIEVE YOU.  TELL HIM A BENCH HAS WET PAINT
ON IT AND HE'LL HAVE TO TOUCH IT TO BE SURE.

UNDER THE MOST RIGOROUSLY CONTROLLED CONDITIONS OF PRESSURE
TEMPERATURE, VOLUME, HUMIDITY, AND OTHER VARIABLES THE
ORGANISM WILL DO AS IT DARN WELL PLEASES.

ANY GIVEN PROGRAM, WHEN RUNNING IS OBSOLETE.

THE ATTENTION SPAN OF A COMPUTER IS ONLY AS
LONG AS ITS ELECTRICAL CORD.

THE DEGREE OF TECHNICAL COMPETENCE IS INVERSELY PROPORTIONAL
TO THE LEVEL OF MANAGEMENT.

A FAILURE WILL NOT APPEAR TILL A UNIT HAS PASSED
FINAL INSPECTION.
```

After Command:



```
300 BILLION STARS IN THE UNIVERSE
AND HE'LL BELIEVE YOU.  TELL HIM A BENCH HAS WET PAINT
ON IT AND HE'LL HAVE TO TOUCH IT TO BE SURE.

UNDER THE MOST RIGOROUSLY CONTROLLED CONDITIONS OF PRESSURE
TEMPERATURE, VOLUME, HUMIDITY, AND OTHER VARIABLES THE
ORGANISM WILL DO AS IT DARN WELL PLEASES.

ANY GIVEN PROGRAM, WHEN RUNNING IS OBSOLETE.

THE ATTENTION SPAN OF A COMPUTER IS ONLY AS
LONG AS ITS ELECTRICAL CORD.

THE DEGREE OF TECHNICAL COMPETENCE IS INVERSELY PROPORTIONAL
TO THE LEVEL OF MANAGEMENT.

A FAILURE WILL NOT APPEAR TILL A UNIT HAS PASSED
FINAL INSPECTION.
```

TK-7875

Entering Line Mode Commands

In addition to the character mode commands, line mode commands are also available from within character mode. The following are examples using two line mode commands in character mode:

- SET NOTRUNCATE

  User types: $\boxed{\text{GOLD}}$ $\boxed{\text{PAGE/COMMAND}}$

  EDT responds: COMMAND: SET NOTRUNCATE ENTER/SUBS

- SUBSTITUTE

  User types: $\boxed{\text{GOLD}}$ $\boxed{\text{PAGE/COMMAND}}$

  EDT responds: COMMAND: S/BIGGEST/BIGGER/WH $\boxed{\text{ENTER}}$

This feature also extends the set of commands available to you in character mode to include macros that you have developed.

Nokeypad Character Mode

For users whose terminals do not have a keypad, there is a nokeypad editor mode that can be used for character mode editing. Although it is somewhat less convenient to use, it does offer the functionality available in character mode. In fact, keypad functions in character mode are actually defined as sequences of one or more nokeypad commands. Table 6-7 shows examples of the nokeypad operations that can be performed. The user types the commands on the standard keyboard. The commands appear at the bottom of the screen, while the results are shown at the appropriate place in the upper part of the screen.

Invoking Nokeypad Mode

To invoke nokeypad mode, the user must issue the SET NOKEYPAD command in line mode, and then using the CHANGE command enter character mode.

Exiting Nokeypad Mode

To exit nokeypad mode, type EXIT. The editor returns to line mode. Typing <CTRL/Z>, the usual method for exiting character mode, does not work for nokeypad mode.

Those who may have to use a terminal without a keypad should now read Chapter 8, Nokeypad Editing, in the EDT Editor Manual.

Table 6-7 Nokeypad Character Mode Commands

User Wants To	Nokeypad Command
Advance cursor two lines	2L
Move cursor backward 2 characters	-2C
Move cursor forward 3 lines, then delete 2 words	3LD2W
Delete a character to right of cursor	D+C
Exit from nokeypad character mode back to line node	EXIT

LEARNING ACTIVITIES

1. READ the following sections in Chapter 5 of the <u>EDT Editor Manual</u>:

   - Locating Text
   - Movement Throughout the buffer
   - Replacing and Substituting Text
   - Using Line Editing Commands
   - Special Characters, Changing Case and Filling Lines

2. DO Lab Exercises 11 through 16 for this module.

USING INDIRECT COMMAND FILES

INTRODUCTION

Once you become adept at using the system, you will notice many constant operations that could be automated to save continuous entering of commands.

RSX-11M/M-PLUS operating systems can execute commands contained in a file. In addition, special commands can be added to control execution flow and make the command process more flexible.

The greatest benefit of this facility is being able to control complex processes, to eliminate typos and save the user from tedious errors.

OBJECTIVES

1. Create an indirect command file that will execute a series of task commands.
2. Create a simple indirect command file that will execute a series of DCL commands.
3. Create an indirect command file that asks the user for input and controls execution flow depending on user input.
4. Invoke an indirect task and CLI command file.

RESOURCES

1. RSX-11M/M-PLUS MCR Operations Manual

WHAT IS AN INDIRECT COMMAND FILE

During the course of a terminal session, there are many operations performed repeatedly, for example, file maintenance and program development. Automating processes can save you a lot of typing and time.

An indirect command file is a text file that contains a series of commands (a process) for one or more tasks. The command file is processed by the task(s) as though the commands came directly from the terminal.

There are two types of indirect command files: task and CLI (Command Line Interpreter).

A task indirect command file contains commands that the task understands. For example, the Peripheral Interchange Program (PIP) is the utility task DCL uses to implement file maintenance commands like COPY, DIR and PURGE. (To perform file maintenance commands when MCR is your current CLI, use PIP directly.) All the commands in the file must be commands that are understood by PIP.

A CLI indirect command file contains commands that the CLI understands. For example, if we are using DCL, the file contains commands such as COPY, DIRECTORY, PURGE, FORTRAN, and LINK. In addition, a CLI command file may contain special commands that control the flow of execution, ask the user for input, and perform tests and other operations. These commands, called directives, require special processing before being passed on to the operating system. The Indirect Command Processor is the task responsible for processing these files.

CREATING SIMPLE CLI COMMAND FILES

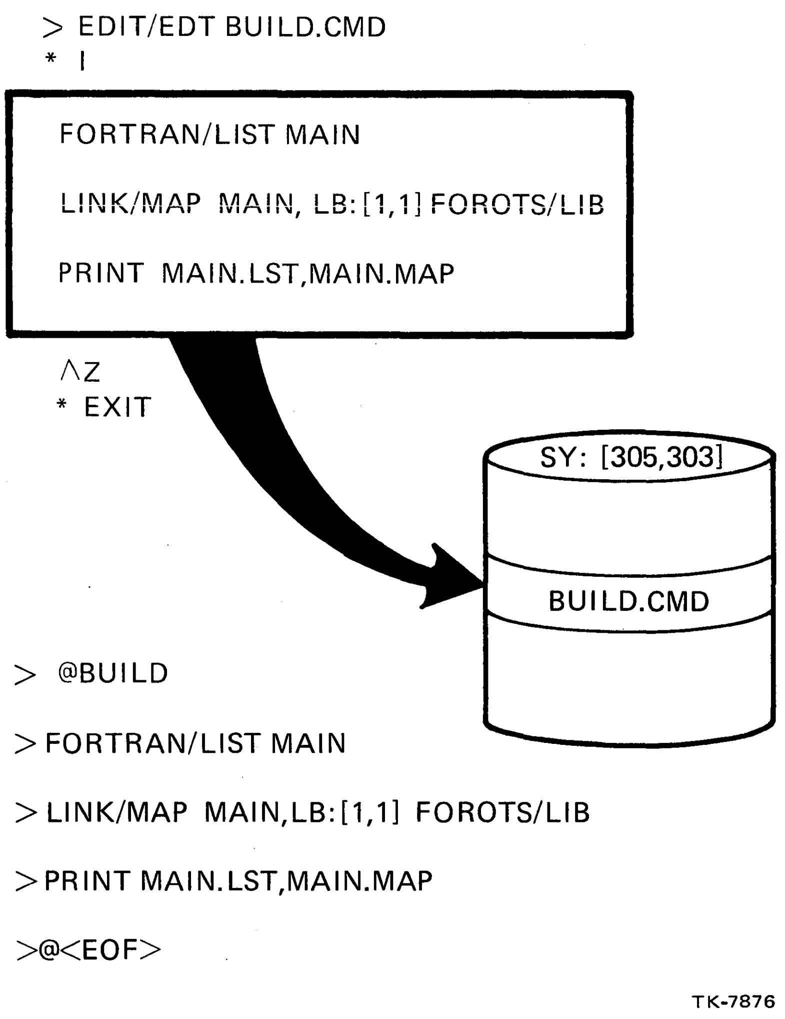
Figure 7-1 illustrates the procedure to create an indirect command file. The following notes are keyed to the figure.

1. Use an editor to create a file with a file type of .CMD. In the example, the file name is BUILD.CMD. The commands that you normally issue from the terminal are typed as input to the file. In the example, the DCL FORTRAN, LINK and PRINT commands are a typical sequence for building a task. You issue these commands over and over again during a terminal session. At the end of the editing session, you have a file that contains these commands.

2 To have the Indirect Command Processor read the file and execute the commands in the file, you type an @ followed by the file name. The default file type is .CMD.

3 The commands are read from the file and processed just as though they have been issued from the terminal. The Indirect Command Processor reads the command. If it does not contain directives that require interpretation by Indirect, the command is passed on to the CLI for processing.

4 This line is the end-of-file indicator that shows that processing of the file is complete.



```
> EDIT/EDT BUILD.CMD
* I

  FORTRAN/LIST MAIN

  LINK/MAP  MAIN, LB:[1,1] FOROTS/LIB

  PRINT  MAIN.LST,MAIN.MAP

  ^Z
  * EXIT

>  @BUILD

> FORTRAN/LIST MAIN

> LINK/MAP  MAIN,LB:[1,1] FOROTS/LIB

> PRINT MAIN.LST,MAIN.MAP

>@<EOF>
```

Figure 7-1 Creating and Invoking a CLI Indirect Command File

CREATING SIMPLE TASK INDIRECT COMMAND FILES

Figure 7-2 shows how to create and use an indirect command file for a task. Typing MAC from MCR invokes the MACRO assembler. The format of the command line is:

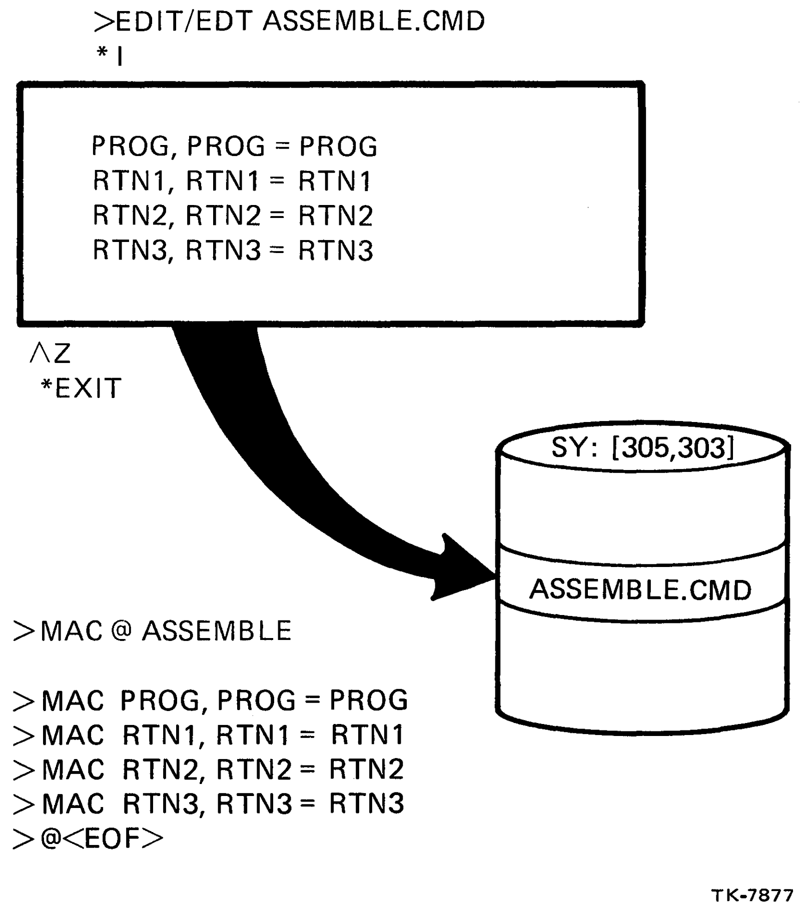
```
MAC taskfile,listfile=sourcefile
```

Suppose you have four source files to assemble: PROG.MAC, RTN1.MAC, RTN2.MAC, RTN3.MAC. If you want to assemble more than one source file at a time, you issue the following sequence of commands:

```
>MAC<RET>

MAC>PROG,PROG=PROG<RET>

MAC>RTN1,RTN1=RTN1<RET>

MAC>RTN2,RTN2=RTN2<RET>

MAC>RTN3,RTN3=RTN3<RET>

MAC>^Z<RET>

>
```

This is the multiple line input format.

Figure 7-2 shows the process to use if you want to assemble more than one source file by using an indirect command file. The first step is to create a file containing the commands for each source file. Then the indirect command file is specified in the MAC command line.



```
>EDIT/EDT ASSEMBLE.CMD
*I

    PROG, PROG = PROG
    RTN1, RTN1 = RTN1
    RTN2, RTN2 = RTN2
    RTN3, RTN3 = RTN3

^Z
 *EXIT
```

```
>MAC @ ASSEMBLE

>MAC PROG, PROG = PROG
>MAC RTN1, RTN1 = RTN1
>MAC RTN2, RTN2 = RTN2
>MAC RTN3, RTN3 = RTN3
>@<EOF>
```

Figure 7-2 A MCR Task Indirect Command File

(Macro Commands are Put into a File and the File Name is Passed to MACRO Assembler)

Benefits of Using an Indirect Command File

Makes the user's job easier by:

- Reducing the amount of typing needed to do frequently performed tasks
- Allowing the command syntax to be corrected before execution
- Reducing the possibility of human error for complex, error-prone procedures
- Allowing for user interaction and program flexibility
- Automating housekeeping tasks

LEARNING ACTIVITIES

1. READ the following sections in Chapter 4 of the RSX-11M/M-PLUS Operations Manual:

   - 4.1, Indirect Command Files
   - 4.2, Indirect Command Processor
   - 4.5, Switches

INCLUDING DIRECTIVES

Directives

A directive is a special command that is interpreted by the indirect command processor. It is distinguished by a period (.) as the first character in the command line, and forms a procedural language that allows you to:

- Define labels
- Define and assign values to symbols of three types: logical, numeric, and string
- Create and access data files
- Control the logical flow within a command file
- Perform logical tests
- Enable or disable any of several operating modes
- Increment or decrement a numeric symbol
- Control time-based and parallel task execution
- Use special symbols for obtaining system information
- Allow for nesting of command files

Tables 7-1 through 7-7 list the available directives, and give examples of each.

Symbols

A symbol is a string of ASCII characters that serve as a name for a memory location where a value is stored. The symbol name consists of a string of 1 to 6 ASCII characters. It must start with a letter (A-Z) or a dollar sign ($). The remaining characters can be alphanumeric (A-Z,Ø-9) or a $. The following are legal symbol names:

```
START

$A3

ATOZEE
```

A symbol can be one of three types, which describes the kind of value stored in the symbol. Symbols can be

- Logical
- String
- Numeric

A logical symbol is one whose value is either true or false (represented by a 1 or Ø respectively). A string symbol is one that contains ASCII characters. A numeric symbol has a number stored in its memory location. The symbol type is defined when a value is first assigned to the symbol.

Values Given to Symbols

There are two sets of directives that give a value to a symbol, .SET and .ASK.

.SETS, .SETN, .SETT, .SETF and .SETL directives give a literal value to a symbol. For example, the values "2" and "PURGE" are literal values that can be assigned to a symbol; they do not vary. The directives .SETO and .SETD redefine the radix of a numeric symbol.

.ASK, .ASKS and .ASKN directives allow for interaction with the user of the indirect command file. They pose a question to the terminal, and wait for a response to be typed. These directives allow a variable value to be assigned to the symbol. That is, each time the same .ASK directive is executed, the resulting value stored in the symbol varies.

Tables 7-1 through 7-7 list some of the available directives. Read through these tables to familiarize yourself with their structure and contents. The sections following Table 7-7 discuss how to use some of the more frequently used directives.

Table 7-1 Directives to Define Symbol Values

User Wants To	Directive	Example	Values
Define a value for a numeric symbol	.SETN	.SETN N1 2 .SETN N2 3 .SETN N3 N1+N2*4	Ø-65,535
Define a value for a string symbol	.SETS	.SETS S1 "A" .SETS S2 "CDEF" .SETS S3 S1 + "B"+S2[1:3]	Ø-8Ø characters
Define a true value for a logical symbol	.SETT	.SETT X	True (1)
Define a false value for a logical symbol	.SETF	.SETF TOLIST	False (Ø)
Ask the user to input a logical value	.ASK	.ASK CONT Do you want to continue .ASK TOLIST Do you want a listing	
Ask the user to input a numeric value	.ASKN	.ASKN SYM Define Numeric Symbol A .ASKN[2:35:16]NUMSYM Define Numeric Symbol A .ASKN[::5]NUM Please enter a number	
Ask the user to input a string value	.ASKS	.ASKS NAME Please enter your name .ASKS[1:15]MIDNAM Please enter your middle name .ASKS[1:6]TASK Enter task to run	
Mark a spot in the command file for future reference	.LABEL:	.START: .1Ø:	

Logical Test Directives

Table 7-2 Directives Used to Perform Tests

User Wants To	Directive	Examples
Compare the values of two symbols	.IF	.IF X LT Y .GOTO 200 .IF N1 <= N2 DIR .IF S1 >= S2+S3[1:1] .INC N
Test if a symbol is true or false	.IFT/.IFF	.IFT A .GOTO 100 .IFF B .GOTO 200
Determine the length of a string symbol	.TEST	.TEST A
Determine if a symbol has been defined	.IFDF/.IFNDF	.IFDF A .GOTO 100 .IFNDF A .ASK A DO YOU WANT TO SET TIME
Determine if a task is installed in the system	.IFINS/.IFNINS	.IFINS PIP .GOTO 400 .IFNINS PIP INS [1,50]PIP
Determine if a driver is loaded	.IFLOA/.IFNLOA	.IFLOA DK: .GOTO 250 .IFNLOA DK: LOA DK:
Determine if a task is active	.IFACT/.IFNACT	.IFACT REPORT .GOTO 350 .IFNACT REPORT RUN REPORT

Table 7-3 Relational Operators

Mnemonic	Meaning
EQ or =	equal to
NE or <>	not equal to
GE or >=	greater than or equal to
LE or <=	less than or equal to
GT or >	greater than
LT or <	less than

Table 7-4 Compound Logical Operators

Mnemonic	Meaning
.AND	Both must meet test
.OR	Either must meet test

Logical Control Directives

Table 7-5 Directives Used to Control Command File Execution

User Wants To	Directive	Example
Define a block structure	.BEGIN .END	.BEGIN .END
Exit current command file	.EXIT	.EXIT N+2
Execute a subroutine	.GOSUB	.GOSUB EVAL .GOSUB STEP 2
Branch around commands	.GOTO	.GOTO 1ØØ .GOTO START
Branch to another command when an error has been detected	.ONERR	.ONERR 1ØØ
Return from a subroutine	.RETURN	.RETURN
Terminate command file processing	.STOP	.STOP

More Useful Directives

Table 7-6 Other Directives

User Wants To	Directive	Examples
Turn off an operating mode	.DISABLE	.DISABLE DATA 2 .DISABLE LOWERCASE .DISABLE ESCAPE
Turn on an operating mode	.ENABLE	.ENABLE QUIET .ENABLE GLOBAL .ENABLE SUBSTITUTION
Decrement a numeric symbol by 1	.DEC	.DEC N
Increment a numeric symbol by 1	.INC	.INC COUNT
Delay processing for a specified period of time	.DELAY	.DELAY 2ØM
Wait for user action before continuing	.PAUSE	.PAUSE
Wait for a task to terminate	.WAIT	.WAIT MAC .WAIT TKB
Initiate a task and continue processing the command file	.XQT	.XQT MAC TEST,TEST=TEST .XQT TKB BLD,BLD=BLD
Put a line into a file	.DATA	
Continue processing using another file	.CHAIN	.CHAIN OUTPUT
Close a file	.CLOSE	.CLOSE 2
Open a file to write	.OPEN	.OPEN 1 SECOUT
Open a file to append data to it	.OPENA	.OPENA 1 SECOUT

Special Symbols

Table 7-7 Special Symbols to Obtain System and User Information

Symbol		Value
L	<ESCAPE>	True, if last query answered
O		with a single escape or ALTmode
G	<DEFAUL>	True, if answer to last numery
I		query defaulted or a timeout occurred
C	<ALPHAN>	True, if answer to last string query
A		only alphanumeric characters
L	<RAD50>	True, if answer to last string query
		only Radix-50 characters
	<MAPPED>	True, if system mapped
	<RSX11D>	True, if RSX-11D operating system
N	<MEMSIZ>	Current memory size (K words)
U	<SYUNIT>	Unit number of user's default device
M	<EXSTAT>	Exit status from last command
E	<STRLEN>	String length of response to .ASKS or .TEST
R	<SPACE>	Free bytes in internal symbol table
I	<SYSTEM>	Operating system (octal number)
C		
S	<CLI>	Current CLI
T	<DATE>	Current date dd-mon-yr
R	<LIBUIC>	Current library UIC
I	<SYDISK>	User's default device
N	<SYSUIC>	System UIC [ggg,mmm]
G	<TIME>	Current time hr:min:sec
	<UIC>	Current UIC [ggg,mmm]

LEARNING ACTIVITIES

1. READ the following sections in Chapter 4 of the RSX-11M/M-PLUS Operations Manual:

   - 4.4, Symbols
   - 4.6, Description of Indirect Directives

Establishing Symbols

The following sections will discuss how to use the directives and symbols. The indirect command files used in the examples have been provided with the course materials on magnetic media. If you have trouble understanding the examples, you may want to execute the indirect command files at your terminal.

Notes on Example 7-1

The following comments are keyed to the example.

❶ The .SETN directive gives a symbol a value, and establishes that symbol as a numeric symbol. The symbols N1, N2 and N3 are numeric symbols; their contents are interpreted as numbers, not ASCII characters nor Boolean (True, False) values. N1 contains the value 2(10); N2 contains the value 3(10). N3 contains the value of the numeric expression N1+N2*4. The expression is evaluated from left to right with the values of the symbols substituted. There is no hierarchy of numeric operators. As shown in item ❻, the value of N3 is 20. Different results will occur if the expression is written .SETN N3 N1+(N2*4). The value of N3 under these circumstances will be 14.

❷ The .SETS directive gives a symbol a value and establishes that symbol as a string symbol. A string symbol is one whose value is a string of ASCII characters. In other words, the content of the symbol is interpreted as ASCII character(s), not a number or a true/false value.

In item ❷, the expression S1+"B"+S2[1:3] has the following meaning:

Take the value of S1 (which is A), concatenate (append) that with the ASCII character B, and concatenate that string with the first three characters contained in S2 (CDE) to produce a value (ABCDE) and store it in symbol S3.

This feature of the indirect command processor is very powerful for the creation of command lines, parsing of command lines, and general manipulation of ASCII strings.

❸ The .SETT and .SETF directives set a symbol value to true and false respectively, and establish the symbol as a logical symbol. You use a logical symbol for testing a condition or situation. For example, if X is set to true, then print the file EXAMPLE.CMD.

❹ These statements display the values of the symbols on the terminal. The .ENABLE SUBSTITUTION directive statement will be discussed later.

❺ All the commands shown thus far are in a file called SYMBOLS.CMD. To execute these commands, the file name preceded by an @ symbol is issued. There is no need to supply the file type (.CMD), as the Indirect Command Processor uses .CMD as the default file type.

❻ These statements display the values of the symbols as they were assigned by the various .SET directives. Of particular interest are the values of N3 and S3, as they are a result of the evaluation of an expression.

```
.;+
.;+
.;        THIS COMMAND FILE SHOWS HOW TO DECLARE A SYMBOL
.;        AND HOW TO GIVE IT A VALUE, THEREBY ESTABLISHING
.;        THE SYMBOL TYPE
.;-
.;-
.;        NOW TO ESTABLISH A NUMERIC SYMBOL
.;
.;
.;
        .SETN N1 2.            !IF THE DOT IS NOT USED TO INDICATE DECIMAL
      1 .SETN N2 3.            !THE NUMBER WILL BE CONSIDERED OCTAL AND
        .SETN N3 N1+N2*4.      !OUTPUT WILL BE IN OCTAL
.;
.;
.;        NOW TO ESTABLISH A STRING SYMBOL
.;
.;
        .SETS S1 "A"
      2 .SETS S2 "CDEF"
        .SETS S3 S1+"B"+S2[1:3]
.;
.;
.;        NOW TO ESTABLISH A LOGICAL SYMBOL
.;
.;
      3 .SETT X
        .SETF Y
.;
.;
.ENABLE SUBSTITUTION
;         N1 = 'N1'
;         N2 = 'N2'
4 ;         N3 = 'N3'
;         S1 = 'S1'
;         S2 = 'S2'
;         S3 = 'S3'
;         X  = 'X'
;         Y  = 'Y'

5 >@SYMBOLS
>;        N1 = 2
>;        N2 = 3
>;        N3 = 20
>;        S1 = A
6 >;        S2 = CDEF
>;        S3 = ABCDE
>;        X  = T
>;        Y  = F
>@ <EOF>
>
>
```

Example 7-1 .SET Directive - Used to Define Symbol Value

Asking the User for Input

Notes on Example 7-2

The following comments are keyed to the example.

❶ A symbol may also be given a value by asking the user to supply that value. The .ASKN, .ASKS and .ASK directives query the user for information, and wait for the user to type a response. On completion, the symbol contains the information the user supplies. Like the .SET directives, the .ASK directives also establish the symbol type as numeric, string or logical. In the syntax of the directive, the question you wish to ask immediately follows the symbol name. The entire command line cannot exceed 132(1Ø) characters. This directive, .ASKN, asks the user to supply a number.

❷ You may also specify a range of values that are acceptable for user input, and a default value if the user wants to take the default by responding with only a carriage return.

❸ This example specifies just the default value for input, with no specification of a range for input.

❹ This example asks for a string of characters to be input.

❺ This example asks for a string of characters of no less than one character and no more than six characters.

❻ The .ASK directive asks for a true/false value.

```
  ;+
  ;+
  ;          THIS COMMAND FILE DEMONSTRATES HOW TO ASK THE USER
  ;          FOR INPUT, THEREBY MAKING A COMMAND FILE MORE FLEXIBLE
  ;          IN IT'S USE.
  ;-
  ;-
  ; ASK THE USER FOR A NUMERIC INPUT
  ;
  ;
1            .ASKN   SYM     DEFINE NUMERIC SYMBOL A
  ;
  ;
2            .ASKN [2:35:16] NUMSYM DEFINE NUMERIC SYMBOL A
  ;
  ;
3            .ASKN [::5] NUM GIVE ME A NUMBER
  ;
  ;
  ; ASK THE USER FOR A STRING INPUT
  ;
  ;
4            .ASKS NAME PLEASE ENTER YOUR NAME
5            .ASKS [1:6] MIDNAM PLEASE ENTER YOUR MIDDLE NAME
  ;
  ;
  ;
  ; ASK THE USER FOR A TRUE/FALSE INPUT
  ;
  ;
6            .ASK CONT DO YOU WANT TO CONTINUE
  ;
  ;
             .ENABLE SUBSTITUTION
  ;          SYM = 'SYM'
  ;          NUMSYM = 'NUMSYM'
  ;          NUM = 'NUM'
  ;          NAME = 'NAME'
  ;          MIDNAM = 'MIDNAM'
  ;          CONT = 'CONT'
```

Example 7-2 .ASK Directive - Used to Define Symbol Value

Notes on Example 7-3

The following comments are keyed to the example.

❶ This is the execution of command file INPUT.CMD, which contains the commands shown in Example 7-2. It was run on a hard-copy terminal to capture the actual activity produced by its execution.

❷ Comments from the command file were displayed at the user's terminal to inform the user of the purpose of the command file.

❸ This line is the result of the interpretation and execution of the first .ASKN directive (item ❶ in Example 7-2). Indirect prompts with the question supplied in the .ASKN command line, and informs the user that the input should be an octal number ([O]). In response, the user types the number 8, which is not an octal number. Indirect performs checks on the input to see if it meets the specification of the .ASK directive.

❹ Indirect comes back and asks again for proper input. The user supplies the value 7 which is acceptable, and execution continues to the next command line in the file.

❺ This example shows the results of item ❷ in Example 7-2. Indirect prompts the user:

- with a question "DEFINE NUMERIC SYMBOL A"
- with the range of values for acceptable input (2 - 35)
- with the default values (16 (8)) if the user just types a carriage return
- that the number should be octal (O)

The user types 36, which is not in range.

❻ Indirect comes back and asks again for the proper input. The user types 34, which is acceptable.

❼ This is the result of executing the .ASKN directive, item ❸ in Example 7-2. The user response was a carriage return, so the symbol will contain the default value of 5.

❽ These two examples show the results of executing items ❹ and ❺ of Example 7-2. The first example asks for a string of characters. The second example asks for a string of 1 through 6 characters. Indirect will check the supplied string for the correct number of characters.

❾ This is the result of item ❻ in Example 7-2. The user is to respond with either a [y] for yes or an [n] for no in answer to the question. The symbol, CONT, will contain a 1 for true.

❿ These are the symbol names and the values they contained after executing this command file. If we executed the command file once again, supplying different answers, those new answers would be reflected here.

```
   >
❶  >@INPUT
   >;+
   >;+
   >;      THIS COMMAND FILE DEMONSTRATES HOW TO ASK THE USER
   >;      FOR INPUT, THEREBY MAKING A COMMAND FILE MORE FLEXIBLE
   >;      IN IT'S USE.
❷  >;-
   >;-
   >; ASK THE USER FOR A NUMERIC INPUT
   >;
   >;
❸  >*      DEFINE NUMERIC SYMBOL A [O]: 8

   AT.T56 -- Invalid answer or terminator
❹  >*      DEFINE NUMERIC SYMBOL A [O]: 7
   >;
   >;
❺  >* DEFINE NUMERIC SYMBOL A [O R:2-35 D:16]: 36

   AT.T56 -- Value not in range
❻  >* DEFINE NUMERIC SYMBOL A [O R:2-35 D:16]: 34
   >;
   >;
❼  >*      GIVE ME A NUMBER [O D:5]:
   >;
   >;
   >; ASK THE USER FOR A STRING INPUT
   >;
   >;
❽  >* PLEASE ENTER YOUR NAME [S]: ELIZABETH
   >* PLEASE ENTER YOUR MIDDLE NAME [S R:1-6]: JANNA
   >;
   >;
   >;
   >; ASK THE USER FOR A TRUE/FALSE INPUT
   >;
   >;
❾  >* DO YOU WANT TO CONTINUE? [Y/N]: Y
   >;
   >;
   >;      SYM = 7
   >;      NUMSYM = 34
❿  >;      NUM = 5
   >;      NAME = ELIZABETH
   >;      MIDNAM = JANNA
   >;      CONT = T
   >@ <EOF>
   >
```

Example 7-3 Execution of INPUT.CMD Showing User Response

Making Logical Tests

Notes on Example 7-4

The following comments are keyed to the example.

1. The .IF directive provides the feature of comparing two symbols to determine the relationship between them. For example, is the value of one symbol larger or smaller than another? Is it equal to the other? It also provides for taking some action depending upon that relationship. In this example, if the value of the first symbol is less than the second symbol, then Indirect will continue processing the command with the label 2ØØ.

2. This is an example of a label. A label marks a place in the command file where control can be transferred in executing the file. A label begins with a period as the first character, followed by 1 to 6 characters, and terminates with a colon.

3. The action taken as a result of comparing two symbols may be another directive, or an operating system command stated in either DCL or MCR. In this example, if N1 is less than or equal to (<=) N2, the DCL DIRECTORY command will be passed to the operating system for processing.

4. The .IF directive in this example controls how many times to execute a series of commands. If the symbol N is equal to or greater than the number 3, processing will continue at the label 2Ø. Otherwise, processing continues at the label STEP2.

5 This is an example of a logical test at label "2Ø". Three questions are asked where the answer to each is either yes or no. The next statement tests the answer to the first question. If the answer is yes, the command file processor continues processing at label "1ØØ". The next statement tests the answer to the second question; if it is false, processing continues at label "15Ø". If a test does not meet the condition specified, processing continues at the next statement in line.

6 The next two statements are examples of compound tests. Using logical operators "OR" and "and", you are able to test for more than one simple condition. For processing to continue at Label "D" by executing the first statement, either A or B must be true, and C must be true. By using parentheses you can change the meaning of an expression. The expression contained within parentheses is evaluated before other operators are evaluated. The second statement changes the meaning to: A must be true, or both B and C must be true for processing to continue at D.

```
  ;
  ;        THIS COMMAND FILE SHOWS HOW TO USE
  ;        DIRECTIVES TO MAKE TESTS ON SYMBOLS
  ;
  ;                                                     ;TESTING CHARACTERS
           .ENABLE SUBSTITUTION
           .SETN N 0
           .ASKS X GIVE ME A CHARACTER
           .ASKS Y GIVE ME A CHARACTER
❶          .IF X LT Y .GOTO 200

❷ .NEXT:                                                ;TESTING NUMBERS
           .ASKN N1 GIVE ME A NUMBER
           .ASKN N2 GIVE ME A NUMBER
❸          .IF N1 <= N2 DIR

           .SETS S1 "AAb"
           .SETS S2 "AA"
  .STEP2:
           .ASKS S3 GIVE ME A CHARACTER
           .IF S1 >= S2+S3[1:1]  .INC N
           ;       N NOW EQUALS 'N'
❹          .IF N >= 3. .GOTO 20
           .GOTO STEP2
                                                        ;LOGICAL TESTS
  .20:
           .ASK A ARE YOU A PROGRAMMER
           .ASK B DO YOU WORK FOR A LIVING
❺          .ASK C ARE YOU RETIRED
           .IFT A .GOTO 100
           .IFF B .GOTO 150
                                                        ;COMPOUND TESTS
❻          .IFT A .OR .IFT B .AND .IFT C .GOTO D
           .IFT A .OR (.IFT B .AND .IFT C) .GOTO D
           .GOTO 20
  .100:    ;A WAS TRUE SO B MUST BE TRUE
           .EXIT
  .150:    ;B WAS FALSE
           .EXIT
  .D:      ;THIS IS D
           .EXIT

  .200:    ;X WAS LESS THAN Y
           .GOTO NEXT
```

Example 7-4 Using the .IF Directive to Test Symbols

Notes on Example 7-5

The following comments are keyed to the example.

❶ This is the result of item ❶ in Example 7-4. The user responds to the first question by typing the letter H. The symbol X now contains the letter H. In the second question, the user responds by typing a J. The symbol Y now contains the letter J. The character test (X less than Y) is true so control passes to the label 200, which displays the message at the terminal.

❷ This is the result of item ❷ in Example 7-4. When the question prompts the user to enter an octal number, the user responds with 5. Symbol N1 now contains the value 5. In response to the second question, the symbol N2 contains the value 4. As a result, the DIRECTORY command does not execute because the test is not met. N1 is not less than or equal to N2.

❸ This is the result of item ❹ in Example 7-4. The value of the symbol N is incremented by 1 each time the value of S1 is greater than or equal to the evaluation of the expression. When N is incremented to three, processing continues at the label 20.

❹ This is the result of item ❺ in Example 7-4, which performs a logical test.

```
    >@LOGICAL
    >;
    >;      THIS COMMAND FILE SHOWS HOW TO USE
    >;      DIRECTIVES TO MAKE TESTS ON SYMBOLS
    >;
    >;                                                ;TESTING CHARACTERS
    >* GIVE ME A CHARACTER [S]: H
❶   >* GIVE ME A CHARACTER [S]: J
    >;X WAS LESS THAN Y
    >;TESTING NUMBERS
❷   >* GIVE ME A NUMBER               [O]: 5
    >* GIVE ME A NUMBER  [O]: 4
  ┌ >* GIVE ME A CHARACTER   [S]: S
  │ >;      N NOW EQUALS 1
❸ │ >* GIVE ME A CHARACTER   [S]: D
  │ >;      N NOW EQUALS 2
  │ >* GIVE ME A CHARACTER   [S]: E
  └ >;      N NOW EQUALS 3
  ┌ >* ARE YOU A PROGRAMMER          ? [Y/N]: Y
  │ >* DO YOU WORK FOR A LIVING? [Y/N]: N
❹ │ >* ARE YOU RETIRED ? [Y/N]: Y
  │ >;A WAS TRUE SO B MUST BE TRUE
  └ >@ <EOF>
    >
```

Example 7-5 Execution of LOGICAL.CMD and the Results of User Input

Controlling Execution Flow

```
.;
.;
.;      THIS COMMAND FILE SHOW HOW TO USE CONTROL
.;      DIRECTIVES TO CONTROL THE EXECUTION OF A
.;      COMMAND FILE
.;
.;
        .ASKN SYM #1: DEFINE NUMERIC SYMBOL A
        .ASKN[2.:35.:16.] NUMSYM #2: DEFINE NUMERIC SYMBOL A
        .ASKN[::5.] NUM #3: GIVE ME A NUMBER
        .ASKS NAME #4: PLEASE ENTER YOUR NAME
        .ASKS[1:6] MIDNAM #5: PLEASE ENTER YOUR MIDDLE NAME
        .ASKS[1:30] LASTNM #6: PLEASE ENTER YOUR LAST NAME
        .ASK CONT #7: DO YOU WANT TO CONTINUE
        .IFF CONT .EXIT
        .ENABLE SUBSTITUTION
        .IF NUMSYM LT 17. .GOSUB STEP2
        .IF NUM EQ 5. .GOTO STEP3
.STEP1:
        ;SYM = 'SYM'
        ;NUMSYM = 'NUMSYM'
        ;NUM = 'NUM'
        .STOP
.STEP2:
;
;
;
        ;HELLO 'NAME' 'MIDNAM' 'LASTNM'
        ;
        ;
        ;HERE IS HOW YOU RESPONDED TO THE QUESTIONS:
        ;
.RETURN
.STEP3:
        ;YOU TOOK THE DEFAULT ON QUESTION #3 SO I WON'T
        ;SHOW YOU HOW YOU RESPONDED TO THE OTHER QUESTIONS.
        .STOP
```

Example 7-6 Command File Showing How to Control Execution Flow

```
>@CONTROL
>* #1: DEFINE NUMERIC SYMBOL A [O]: 3
>* #2: DEFINE NUMERIC SYMBOL A [D R:2.-35. D:16.]: 12.
>* #3: GIVE ME A NUMBER [D D:5.]:
>* #4: PLEASE ENTER YOUR NAME [S]: ELIZABETH
>* #5: PLEASE ENTER YOUR MIDDLE NAME [S R:1-6]: JANNA
>* #6: PLEASE ENTER YOUR LAST NAME [S R:1-30]: VAN
>* #7: DO YOU WANT TO CONTINUE? [Y/N]: Y
>;
>;
>;
>;HELLO ELIZABETH JANNA VAN
>;
>;
>;HERE IS HOW YOU RESPONDED TO THE QUESTIONS:
>;
>;YOU TOOK THE DEFAULT ON QUESTION #3 SO I WON'T
>;SHOW YOU HOW YOU RESPONDED TO THE OTHER QUESTIONS.
>@ <EOF>
>


>
>
>@CONTROL
>* #1: DEFINE NUMERIC SYMBOL A [O]: 55
>* #2: DEFINE NUMERIC SYMBOL A [D R:2.-35. D:16.]: 2
>* #3: GIVE ME A NUMBER [D D:5.]: 32
>* #4: PLEASE ENTER YOUR NAME [S]: JOHN
>* #5: PLEASE ENTER YOUR MIDDLE NAME [S R:1-6]: JACOB
>* #6: PLEASE ENTER YOUR LAST NAME [S R:1-30]: JINGLEHEIMER
>* #7: DO YOU WANT TO CONTINUE? [Y/N]: Y
>;
>;
>;
>;HELLO JOHN JACOB JINGLEHEIMER
>;
>;
>;HERE IS HOW YOU RESPONDED TO THE QUESTIONS:
>;
>;SYM = 55
>;NUMSYM = 2
>;NUM = 32
>@ <EOF>
>@CONTROL
>* #1: DEFINE NUMERIC SYMBOL A [O]: 72
>* #2: DEFINE NUMERIC SYMBOL A [D R:2.-35. D:16.]: 26
>* #3: GIVE ME A NUMBER [D D:5.]: 62
>* #4: PLEASE ENTER YOUR NAME [S]: PAUL
>* #5: PLEASE ENTER YOUR MIDDLE NAME [S R:1-6]: H.
>* #6: PLEASE ENTER YOUR LAST NAME [S R:1-30]: HARVEY
>* #7: DO YOU WANT TO CONTINUE? [Y/N]: N
>@ <EOF>
>
```

Example 7-7 Execution of CONTROL.CMD and User Response

Setting Operating Modes

Notes on Example 7-8

The following comments are keyed to the example.

1. Quiet mode suppresses the echoing of operating system command lines at the terminal. In this example, if the user answers yes to the question, the .ENABLE QUIET statement turns quiet mode on. In this case, the SHOW TASKS ACTIVE command would not echo at the terminal. However, the results of issuing the command would be displayed. If the user answers no to the question, the SHOW TASKS ACTIVE command would be echoed, and the results of issuing the command would be displayed. You use .ENABLE and .DISABLE to turn the operating modes on and off.

2. Substitution mode allows for the substitution of a symbol's value for the symbol in the command line. For example, if the user answers the question, SPECIFY SOURCE FILE, with NOTES.TXT, the statement TYPE 'FILE' would become TYPE NOTES.TXT. Two steps must be accomplished for this mode to work correctly. Substitution mode must be enabled, and the symbol must be enclosed in single quotes where the substitution is desired.

3. Lowercase mode allows characters read from the terminal in response to .ASKS directives to be stored in the string symbol without lower- to uppercase conversion.

4. This mode allows an escape character to be an acceptable response to a .ASK directive.

5. This mode allows the user to output lines to a secondary file. Enabling this mode directs Indirect to output the lines between the .ENABLE DATA and .DISABLE DATA statements to a secondary file. This is most useful for dynamically creating another indirect command file, or a data file for another task.

6. Global symbol mode provides for symbols whose names begin with a $ to be recognized in all levels of command files. Those symbols whose name do not begin with a $ are only recognized in the level of the command file in which they are defined.

For more information on the various operating modes, refer to Chapter 4 of the _MCR Operations Manual_.

```
    ;+
    ;+
    ;       THIS COMMAND FILE SHOWS HOW THE OPERATING MODES
    ;       WORK.
    ;
    ;
    ;               LOWER CASE
    ;               DATA
    ;               GLOBAL
    ;               SUBSTITUTION
    ;               ESCAPE
    ;               QUIET
    ;-
    ;-
    ;
 ❶  ;   QUIET MODE
    ;
            .ASK QUIET DO YOU WANT COMMAND LINES SUPPRESSED
            .IFT QUIET .ENABLE QUIET
            .IFF QUIET .DISABLE QUIET
            SHOW TASKS ACTIVE
    ;
 ❷  ;   SUBSTITUTION MODE
    ;
            .ENABLE SUBSTITUTION
            .ASKS FILE SPECIFY SOURCE FILE
            TYPE 'FILE'
    ;
 ❸  ;   LOWER-CASE MODE
    ;
            .DISABLE QUIET
            .ASK CASE DO YOU WANT LOWER-CASE ENABLED
            .IFT CASE .ENABLE LOWERCASE
            .IFF CASE .DISABLE LOWERCASE
            .ASKS A DEFINE STRING SYMBOL A
    ;'A'
    ;
 ❹  ;   ESCAPE RECOGNITION MODE
    ;
    ;IF YOU WANT A LIST OF OPTIONS, TYPE <ESC>
            .ENABLE ESCAPE
            .ASKS A ENTER OPTION
            .IFT <ESCAPE> .GOTO LIST
            .STEP1:
    ;
 ❺  ;   DATA MODE
    ;
            .OPEN ASSEMBLE.CMD
            .ENABLE DATA
            .ENABLE SUBSTITUTION
            MACRO/LIST ''$X''
    .DISABLE DATA
            .CLOSE FILE.TXT
    ;
    ;   GLOBAL MODE
 ❻  ;
            .ENABLE GLOBAL
            .SETS $X "TEST"
            @ASSEMBLE
            .STOP
    .LIST:  ;OPTIONS ARE: A(ADD), S(SUBTRACT),M (MULTIPLY)
            .GOTO STEP1
```

Example 7-8 Command File Showing Operating Mode Usage

Notes on Example 7-9

The following comments are keyed to the example.

1. In this example, the use of quiet mode is shown. In response to the question, the user answers yes. The DCL command, SHOW TASKS ACTIVE, which executes, does not echo at the terminal. However, the results of the command do.

2. In this example, quiet mode is still active. The TYPE TEST.MAC command does not echo on the terminal. However, the string "TEST.MAC" replaces 'FILE' in the TYPE command in item 2 of Example 7-8. This feature allows your indirect command file to work for variable file names rather than just one file.

3. In this example, lowercase characters entered in response to a question remain in lowercase. If this mode was not enabled, lowercase letters would be converted to uppercase on input. This could be a problem later when comparing characters in symbols if the case did not match (i.e., uppercase to lowercase).

4. Escape recognition mode permits the response to an .ASK, .ASKN, or .ASKS directive to be an escape character. A question answered this way sets the special logical symobl <ESCAPE> to true. You can then test this symbol for true or false.

Ex 7.8

5. This example creates an indirect command file that will be used in the following example. The .OPEN directive opens a file with the name supplied with the directive. In this case, the file name is ASSEMBLE.CMD. The lines following the .ENABLE DATA statement up to the .DISABLE DATA statement are output to the file ASSEMBLE.CMD. The .CLOSE statement closes the file. A DIR command issued after this operation would show a file called ASSEMBLE.CMD in the user's UFD.

6 In this example, ASSEMBLE.CMD is invoked from the indirect command file. This is a second level of indirection. The .ENABLE GLOBAL statement allows for the recognition of the symbol $X from another indirect command file. The second statement gives a value of the string TEST to the symbol $X. The next statement invokes the indirect command file that was created in the previous example. Processing now continues from the indirect command file ASSEMBLE.CMD. The symbol $X is recognized in this file, and substitution takes place. This creates the command line of MACRO/LIST TEST, which is sent to the operating system for processing. Quiet mode was disabled in example 3 above, so the command line is echoed at the terminal. The results of the assemble are also shown.

```
  >@OPERATING
  >;+
  >;+
  >;      THIS COMMAND FILE SHOWS HOW THE OPERATING MODES
  >;      WORK.
  >;
  >;      THERE ARE SIX OPERATING MODES:
  >;              LOWER CASE
  >;              DATA
  >;              GLOBAL
  >;              SUBSTITUTION
  >;              ESCAPE
  >;              QUIET
  >;-
  >;-
  >;
  >;   QUIET MODE
  >;
1 >* DO YOU WANT COMMAND LINES SUPPRESSED? [Y/N]: Y
  MCR...
  SHOT56
  AT.T56
2 >* SPECIFY SOURCE FILE [S]: TEST.MAC
          .MACRO  EXIT$S ERR
          .MCALL  DIR$
          MOV     (PC)+,-(SP)
          .BYTE   51.,1
          DIR$
          .IIF NB <ERR>,  CALL    ERR
          .ENDM   EXIT$S
3 >* DO YOU WANT LOWER-CASE ENABLED? [Y/N]: Y
  >* DEFINE STRING SYMBOL A [S]: This is Lower Case
  >;This is Lower Case
  >;
  >;   ESCAPE RECOGNITION MODE
  >;
4 >;IF YOU WANT A LIST OF OPTIONS, TYPE <ESC>
  >* ENTER OPTION [S]:
  >;OPTIONS ARE: A(ADD), S(SUBTRACT),M (MULTIPLY)
  >;
5 >;   DATA MODE
  >;
  >;
  >;   GLOBAL MODE
  >;
6 >MACRO/LIST TEST
  ERRORS DETECTED:  1
  TEST,TEST/SP=TEST
  >@ <EOF>
```

Example 7-9 Execution of OPERATING.CMD and User Response

Using Special Symbols

Table 7-7 lists some of the special symbols available for your use. These symbols provide system information and replies to queries presented during command file execution. The symbol names are enclosed in brackets and are used in the same manner as user-defined symbols.

Notes on Example 7-10

The following comments are keyed to the example.

1. Nine different special symbols are used in this example. When the file is executed, these comment lines will be displayed at the terminal with the values of the special symbols substituted.

2. In this example, the user is asked to input a string of characters. The special symbols are tested and, if found true, the command following the statement is executed.

   If the user types an <ESC> (escape character), the indirect command file will exit. If all alphanumeric characters are typed, Indirect will display the message at the terminal that all characters typed were alphanumeric. If all characters were Radix-5Ø, another message is displayed. In the .SETN statement, the number of characters in the input string is saved in the symbol <STRLEN> and then output with a message.

```
  ;
  ;       THIS COMMAND FILE SHOWS HOW TO USE THE SPECIAL
  ;       SYMBOLS THAT ARE AVAILABLE TO THE USER
  ;

          .ENABLE SUBSTITUTION
  ;
  ;       THE USER OF THIS COMMAND FILE IS '<UIC>' AND HE
  ;       IS RUNNING ON AN RSX SYSTEM WITH '<MEMSIZ>' K OF CORE.
1 ;       HIS DEVICE IS '<SYDISK>''<SYUNIT>'.  HE IS PRESENTLY
  ;       USING '<CLI>' AS HIS CURRENT COMMAND LANGUAGE
  ;       INTERPRETER.  THE DATE IS '<DATE>' AND THE TIME THIS
  ;       COMMAND FILE IS RUNNING IS '<TIME>'.  THE LIBRARY
  ;       FOR THIS SYSTEM IS IN '<LIBUIC>' AND THE SYSTEM UIC IS
  ;       '<SYSUIC>'.
  ;

          .ASKS USER GIVE ME A STRING
          .IFT <ESCAPE> .EXIT
2         .IFT <ALPHAN> ;YOU TYPED ALL ALPHA/NUMERIC CHARACTERS
          .IFT <RAD50>  ;YOU TYPED ALL RAD50 CHARACTERS

          .SETN STRLEN <STRLEN>
          .SETD STRLEN
  ;
  ;YOU JUST TYPED 'STRLEN' CHARACTERS
  ;
```

Example 7-1Ø Command File Showing Special Symbol Usage

Notes on Example 7-11

1. The indirect command file, SPECIAL.CMD, is executed three times in this example. Each time, Indirect substitutes the system information for the special symbols. For example, the special symbol <UIC> represents the UIC in which the indirect command file is processing. The value of this symbol in this example contains the character string [3Ø5,3Ø3]. Indirect substitutes this value in the comment line before displaying the line at the terminal. You may want to try running this example on your system to see how it works and how the information changes.

```
>
>@SPECIAL
>;
1 >;        THIS COMMAND FILE SHOWS HOW TO USE THE SPECIAL
>;        SYMBOLS THAT ARE AVAILABLE TO THE USER
>;
>;
>;        THE USER OF THIS COMMAND FILE IS [305,303] AND HE
>;        IS RUNNING ON AN RSX SYSTEM WITH 512 K OF CORE.
>;        HIS DEVICE IS DB0.  HE IS PRESENTLY
>;        USING DCL AS HIS CURRENT COMMAND LANGUAGE
>;        INTERPRETER.  THE DATE IS 22-SEP-81 AND THE TIME THIS
>;        COMMAND FILE IS RUNNING IS 14:09:07.  THE LIBRARY
>;        FOR THIS SYSTEM IS IN [1,54] AND THE SYSTEM UIC IS
>;        [4,54].
>;
>* GIVE ME A STRING [S]: THISISASTRING
>;YOU TYPED ALL ALPHA/NUMERIC CHARACTERS
>;YOU TYPED ALL RAD50 CHARACTERS
>;
>;YOU JUST TYPED 13 CHARACTERS
>;
>@ <EOF>
>@SPECIAL
>;
>;        THIS COMMAND FILE SHOWS HOW TO USE THE SPECIAL
>;        SYMBOLS THAT ARE AVAILABLE TO THE USER
>;
>;
>;        THE USER OF THIS COMMAND FILE IS [305,303] AND HE
>;        IS RUNNING ON AN RSX SYSTEM WITH 512 K OF CORE.
>;        HIS DEVICE IS DB0.  HE IS PRESENTLY
>;        USING DCL AS HIS CURRENT COMMAND LANGUAGE
>;        INTERPRETER.  THE DATE IS 22-SEP-81 AND THE TIME THIS
>;        COMMAND FILE IS RUNNING IS 14:09:39.  THE LIBRARY
>;        FOR THIS SYSTEM IS IN [1,54] AND THE SYSTEM UIC IS
>;        [4,54].
>;
>* GIVE ME A STRING [S]: THIS IS A STRING
>;
>;YOU JUST TYPED 16 CHARACTERS
>;
>@ <EOF>
>@SPECIAL
>;
>;        THIS COMMAND FILE SHOWS HOW TO USE THE SPECIAL
>;        SYMBOLS THAT ARE AVAILABLE TO THE USER
>;
>;
>;        THE USER OF THIS COMMAND FILE IS [305,303] AND HE
>;        IS RUNNING ON AN RSX SYSTEM WITH 512 K OF CORE.
>;        HIS DEVICE IS DB0.  HE IS PRESENTLY
>;        USING DCL AS HIS CURRENT COMMAND LANGUAGE
>;        INTERPRETER.  THE DATE IS 22-SEP-81 AND THE TIME THIS
>;        COMMAND FILE IS RUNNING IS 14:09:54.  THE LIBRARY
>;        FOR THIS SYSTEM IS IN [1,54] AND THE SYSTEM UIC IS
>;        [4,54].
>;
>* GIVE ME A STRING [S]: TH65IS32A5STRING
>;YOU TYPED ALL ALPHA/NUMERIC CHARACTERS
>;YOU TYPED ALL RAD50 CHARACTERS
>;
>;YOU JUST TYPED 16 CHARACTERS
>;
>@ <EOF>
>
```

Example 7-11 Execution of SPECIAL.CMD

Examples 7-12 and 7-13 show how to use the commands you have learned to automate processes that you perform every day. In Example 7-12, the first command file will change your current CLI to either MCR or DCL, depending on which CLI is presently active at your own terminal. The second example will print the time of day, and the third will send a message file to a terminal.

Example 7-13 is a file maintenance task that you perform every day. It will edit a file, print a copy of the edited files, and then clean up your work area by puring unnecessary copies of the file when you are finished. You may want to try these at your terminal to see the results.

```
.;
.;      THIS COMMAND FILE WILL CHANGE THE CURRENT
.;      CLI TO EITHER DCL OR MCR DEPENDING UPON WHICH IS
.;      CURRENT.
.;
.;
        .ENABLE QUIET
        .IF <CLI> EQ "DCL" .GOTO 10
        SET /DCL=TI:
        .EXIT
.10:    SET TERMINAL MCR
```

```
.ENABLE SUBSTITUTION
.SETS t <time>
;THIS IS THE CORRECT TIME: 'T'
```

```
.;      THIS FILE WILL SEND A MESSAGE FILE
.;      TO A SPECIFIED TERMINAL
.;
        .ENABLE SUBSTITUTION
        .ASKS FI TERMINAL NUMBER
        .ENABLE QUIET
        PIP TT'FI':=TEXT.TXT
;
```

Example 7-12 An Indirect Command File to Send Message to Terminal, and Change Current CLI

```
.;      This command file supports file editing and
.;      management, allowing user to edit, purge,
.;      print,  and run files.

        .ENABLE LOWERCASE        !We want lowercase
        .ENABLE SUBSTITUTION     !and symbol substitution
.;
.;      Some definitions:
        .SETF   IND

.;      Get file type
.10$:   .ASKS   TYPE    FILE TYPE EXTENSION (E.G., "CMD")
        .TEST   TYPE    !Null type?
        .IF <STRLEN> EQ 0        .GOTO   ERR1    !Then error

.;      Get file name
.15$:   .ASKS   NAME    NAME FILE YOU WISH TO EDIT ( NO TYPE)

.;      Make complete filespec by concatenating name and type
        .SETS   FILE    NAME+"."+TYPE

.;      Valid name?
        .TEST   NAME
        .IF <STRLEN> EQ 0        .GOTO   ERR2    !If no, report error

.;      Edit file
.20$:   EDT     'FILE'

.;      After EDT exit, ask if another edit is desired
        .ASK    REPEAT  DO YOU WANT TO EDIT THE SAME FILE AGAIN
        .IFT    REPEAT  .GOTO   20$

.;      If filetype was .CMD (indirect command file)
.;      see if user wants to execute file
        .IF TYPE EQ "cmd" .ASK  IND     DO YOU WANT TO RUN 'FILE'
        .IF TYPE EQ "CMD" .ASK  IND     DO YOU WANT TO RUN 'FILE'
        .IFT    IND     @'FILE' !If yes, run it

.;      to RUNOFF and print .MEM file on LP:

.;      See if user wants to print file
        .ASK    PRINT   DO YOU WANT TO PRINT THE FILE
        .IFT    PRINT   PRINT   'FILE'  !If yes, print it

.;      See if user wants to purge previous versions of file
.30$:   .ASK    PURGE   DO YOU WANT TO PURGE 'FILE'

.;      If yes, send to PIP
        .IFT    PURGE   PIP 'FILE'/PU/LD

.;      See if user wants to edit another file
        .ASK    OTHER   DO YOU WANT TO EDIT ANOTHER FILE
        .IFT    OTHER   .GOTO   10$     !If so, loop
        .GOTO   100
;
;       ERROR PROCESSING
;
.ERR1:
;
;       FILE TYPE ERROR - PLEASE ENTER THE FILE TYPE (EXTENSION
;                         OF THE FILE YOU WISH TO EDIT)
        .GOTO   10$
.ERR2:
;
;       FILE NAME ERROR - PLEASE ENTER THE FILE YOU WISH TO EDIT
;
        .GOTO   15$
.100:
```

Example 7-13 An Indirect Command File to Edit, Print and Purge Files

CONTROLLING TASK EXECUTION

INTRODUCTION

You have seen how the operating system manages resources and controls memory sharing and CPU time. In this module you will learn how to set up and/or change various task parameters that affect how the task competes for memory and CPU time.

OBJECTIVES

1. Know when and how to change task parameters, such as priority, that affect task execution.
2. Use the RUN command for immediate and time-based scheduling of a task.
3. Install a task as an MCR spawnable task, and how to invoke such a task using an MCR command.

RESOURCES

1. Introduction to RSX-11M/M-PLUS Systems
2. RSX-11M/M-PLUS Command Language Manual
3. RSX-11M/M-PLUS MCR Operations Manual
4. RSX-11M/M-PLUS Task Builder Manual

HOW RSX MANAGES TASKS

Priority and Scheduling

In module 1, Overview, a discussion on how RSX-11M/M-PLUS manages tasks indicates that a task's priority and its state govern when it will get CPU control. The highest-priority, ready-to-run task is given control of the CPU. Once a task controls the CPU, it continues to execute until it completes, becomes blocked, or a task of higher priority is ready-to-run. The current task is the task that holds CPU control. The scheduler changes the current task to the higher priority task through the occurrence of a significant event. When a significant event occurs, the scheduler looks at the Active Task List (ATL) to select the current task. Only those tasks in the ATL are considered for CPU scheduling.

In Figure 8-1, the arrows weaving through the System Task Directory point to the tasks that are active and competing for resources. TASKB, TASKD, TASKE, TASKG, TASKI, and TASKJ are the active tasks. The scheduler, in searching this list, will select TASKE as the current task. It is the highest-priority, ready-to-run task. If TASKE completes and no other task state changes occur in the list, TASKJ will then be the next task selected as the current task. Resource scheduling only occurs at the time of a significant event. If you need to, refer to Table 1-3 to refresh your memory on what events constitute significant events.

Figure 8-2 illustrates the various task states.

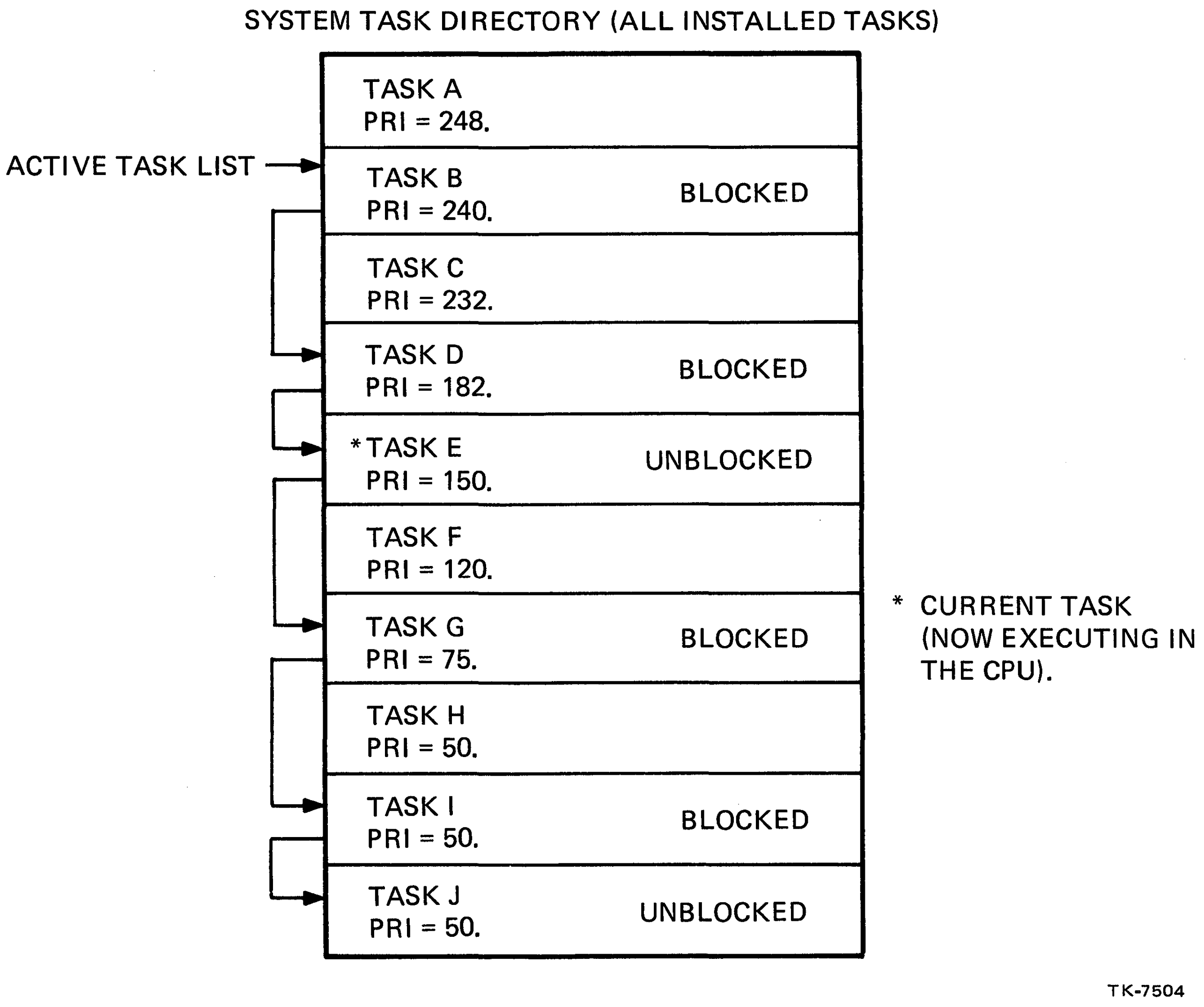


Figure 8-1 Highest Priority Ready-To-Run Task Gains CPU Control

Task States

The following is a review of task states and their meaning:

1	UNKNOWN	- A task for which there is no STD entry
2	KNOWN	- A task for which there is a STD entry
3	DORMANT	- A task for which there is a STD entry but no ATL entry (no request to run)
4	ACTIVE	- A task for which there is a STD and an ATL entry (a request to run has been made)
5	READY-TO-RUN	- An active task not waiting for any event
6	BLOCKED	- An active task waiting for an event
7	CURRENT	- An active task with current CPU control

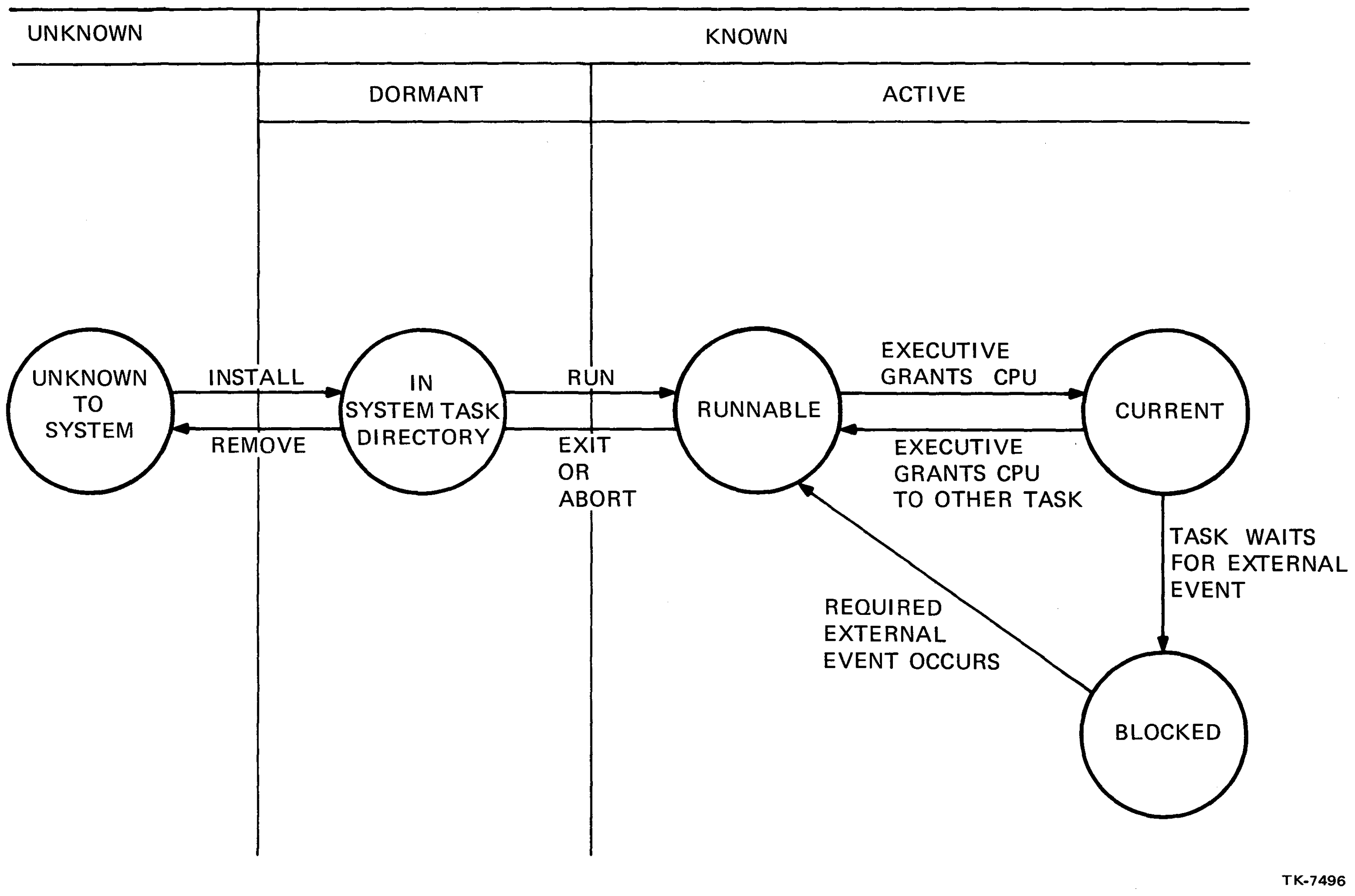


Figure 8-2 Task States

(Every Task is Allocated CPU, and Memory-Dependent Upon its State)

Memory Allocation

Before an active task can gain control of the CPU, it must be memory-resident. Memory allocation occurs on a partition-by-partition basis. A task's state, priority, and the partition in which the task was built to run are all considered when a task memory allocation request is made. The task loads into the partition if there is room for it, and no higher-priority task is competing for the same spot. If there is no room in the partition, the task must wait until the operating system can provide memory space. The task cannot be loaded into another partition.

The operating system checks to see if it can provide memory space for the task by searching the partition for a memory-resident task of lower-priority that is checkpointable. If found, that task will be checkpointed to allow loading of the higher-priority task. The memory allocator checkpoints as many tasks as necessary to load the higher-priority task. However, all checkpointed tasks will be of lower-priority, and have the checkpoint characteristic built into the task.

LEARNING ACTIVITY

1. DO Written Exercises 1 through 3 for this module.

USING TASK BUILDER FACILITIES

The Task Builder defaults assume typical usage and storage requirements. You can override these defaults by using switches and options, thus tailoring a task for its own input/output and storage requirements.

Some attributes that can be altered:

- Task Name
- Priority
- Partition
- Logical Units
- Checkpointability

Others can be found in the RSX-11M/M-PLUS Task Builder Manual.

Assigning a Task Name

Task names by default are the first six characters of the task image file name. Sometimes, this naming convention is not suitable. If so, the name can be specified at task-build time by using the /OPTIONS qualifier on the LINK command, as shown in the example below. The Task Builder will prompt with Option?. The user then supplies the TASK option with a 1 to 6 character name. The Task Builder will than prompt again with Options?. More options can be entered if needed. Complete the process by typing a carriage return in response to the prompt. The Task Builder will then process the command data to create a task image file.

```
>LINK/OPTIONS PROG
Option?TASK=QRST
Option?<CR>
```

Setting Task Priority

The priority of a task is also established at task-build time. If not overridden at install time, the task-build priority becomes the default priority at which the task will run. If a task is run on a multiuser protection system, it will not run at a priority greater than the system default priority (5Ø), unless it is installed or run from a privileged terminal.

The priority is a decimal integer in the range of 1-25Ø. The default priority is 5Ø. If a real-time task is being created, its priority should be in the range of 15Ø-25Ø. To specify the priority when building your task, use the following commands:

```
>LINK/OPTIONS PROG
Option?PRI=15Ø
Option?<CR>
```

Indicating a Partition

A third consideration when building your task is the partition in which the task is to run. Partition selection is dependent upon the task's function. If it is a real-time task, you will probably build the task to run in its own special partition so that it does not compete with general tasks for a position in memory. If the task is for general use, you will probably build it to run in the general partition called GEN. GEN is the default partition used if you do not specify a partition. The partition in which a task runs can be overridden when the task is installed, without having to rebuild the task. The following example shows how to specify a different partition when building your task.

```
LINK/OPTIONS TEST
Option?PAR=DRVPAR
Option?<CR>
```

Making a Task Checkpointable

Generally, tasks are built checkpointable. This means the task can be swapped out of memory and stored on disk to make room for a higher-priority task. When the system checkpoints a task, it places the task's memory image out on disk. There is a file on each disk volume called CORIMG.SYS. The system uses this file to temporarily store checkpointed task images. When space becomes available again, the system reads the task image from the checkpoint file and writes it back into memory. The task is again competing for CPU time.

In addition, space can be allocated within the task image file to store the memory image of a checkpointed task. This guarantees available checkpoint space even when the system checkpoint file is full. The following command examples show how to build a task to be checkpointable. The default is not checkpointable.

```
LINK/CHECKPOINT:SYSTEM VIPPROG

LINK/CHECKPOINT:TASK VIPPROG
```

In the first example, the task is built to be checkpointed to the system checkpoint file.

In the second example, the task is built with space reserved within the task image file for checkpointing space. The size of the task image file created with this qualifier will be approximately twice the size of the task image created with the /CHECKPOINT:SYSTEM qualifier.

In both cases, the tasks, if checkpointed, will first be written to the System Checkpoint File.

Assigning Logical Units

Example 8-1 shows the process of associating logical units with the actual physical device that is to be used for input/output. A source program uses logical units for reading and writing to devices (I/O). They are called logical units because at this point in the program development process, the physical device to which the input/output will be done is not known. It may be known that a certain type of disk drive is to be used, but the specific unit probably is not. Furthermore, you would not want to "hardwire" the real device name into the program. It is advantageous to delay association with the real physical device as long as possible.

In item ❶ in Example 8-1, in the statement WRITE (1,1Ø) VAR1, "1" is the logical unit number. There are three logical units shown in this short program fragment. Two of them are output devices (WRITE statements) and one is an input device (READ statement). This program will read from one file and write to two devices. At this point, you do not know what devices will be used for the files.

When the task is linked using the /OPTIONS qualifier, you must further inform the Task Builder about the devices. First, you must tell how many I/O units the program has (item ❷ in the example). The Task Builder will assign six units by default if you do not specify a number. In the case of FORTRAN programs, the Task Builder allocates seven units.

Next, you must give the Task Builder the physical, logical or pseudo device assignments for the logical units. This is shown in items ❸, ❹, and ❺ of the example. The ASG keyword accepts assignments with the following syntax: Physical device:Logical unit. In this example, the logical unit number 1 is associated with the pseudo device SY: (the user's default device). Every time this program writes to logical unit 1, output will appear on the user's default device (item ❸). The program will read from logical device LZ:, which correlates to logical unit 2. As this is a logical device, the connection to a physical device has not been completed. After task-building, another step is required before this program can be successfully run. The logical device LZ: must be assigned to a physical device using the ASSIGN command.

Logical unit 3 has been associated with the physical device MMØ:, the first magnetic tape unit on the system. Every time the program does a write to logical unit 3, output goes to MMØ:.

After the task has been built, and before it can be run properly, the association between logical device LZ: and some physical device must be accomplished. This is done with the DCL ASSIGN command, as shown in item ❻ of the example. Only then can the program be run.

By delaying logical unit assignments until just before running the program, you provide the program with device independence. This is especially useful when a device is not working, or is busy. For example, suppose the magnetic tape unit is being used by someone else, and you wish to run this program, which was built to use the magnetic tape unit. You might either wait until the tape unit becomes free (which could be a long time), or rewrite and rebuild your program to write to another magnetic tape unit. These two solutions are time-consuming and tedious. By using the procedure in Example 8-1, you save time and frustration.

When you build your task you may specify up to 25Ø units.

```
       PROG.SRC
   ┌─────────────────────────┐
   │      . ❶                │
   │      .                  │
   │      .                  │
   │  WRITE (1,1Ø) VAR1      │
   │  READ  (2,15) VAR2      │
   │      .                  │
   │      .                  │
   │      .                  │
   │  WRITE (3,25) VAR3      │
   │      .                  │
   │      .                  │
   │      .                  │
   └─────────────────────────┘

   >LINK/OPTIONS  PROG
     OPTIONS?  UNITS=3❷
     OPTIONS?  ASG=SY:1,LZ:2,MMØ:3
     OPTIONS?  <RET>❸   ❹     ❺

   >ASSIGN DK:  LZ: ❻
   >RUN PROG
```

Example 8-1 Associating Logical Units with Physical Devices

Notes on Example 8-1

The following comments are keyed to the example.

❶ Logical unit numbers represent a device for I/O

❷ Telling the Task Builder there are 3 input/output units

❸ Assigning logical unit 1 to user's default disk (pseudo device)

❹ Assigning logical unit 2 to the logical device LZ:

❺ Assigning logical unit 3 to a physical device (the first magnetic tape unit)

❻ Before running the task, the association between the logical device LZ: and some physical device must be made for the program to execute correctly.

The association between SY: and the user's default physical device was made when the user logged on, or through the DCL ASSIGN command.

Table 8-1 shows how to override some of these task attributes. For instance, it is not necessary to rebuild a task to run it at a different priority. Nor is it necessary to reinstall a task to have it run at a different priority.

Table 8-1 Overriding Task Attributes

When	Priority	Partition	Checkpoint-ability	Task Name	Task Size
At Link/TKB Time	yes default: 50.	yes default: GEN	yes default: not check-pointable	yes default: from first input file	yes default: no over-lays
Override					
At Install Time	yes	yes	yes	yes	no
Once Installed					
Without Running	yes, use ALT	no	no, but can FIX	no	no
At Run Time	no	no	no	no	no
Not Yet Installed					
If Using Install-Run-Remove Form of the Run Command	yes	yes	yes	yes	no

Characteristics of a task can be changed without rebuilding the task.

Table 8-2 shows the MCR commands that are the equivalent of the DCL commands used earlier. By typing TKB<RET>, you are invoking the multiple input line format of the Task Builder. You must use this form to enter options. (Input files can also be typed on multiple lines.) When you type (/), TKB prompts with ENTER OPTIONS. Typing (//) terminates option mode and the Task Builder creates the task image file.

Table 8-2 MCR Commands to Invoke the Task Builder to Override Task Builder Defaults

User Wants To	MCR Commands
Override task attributes at task-build time	>TKB<RET> TKB>PROG/CP, PROG=PROG<RET> TKB>/<RET> ENTE TKB>PRI=15Ø<RET> TKB>PAR=SYSPAR<RET> TKB>TASK=VIPTSK<RET> TKB>UNITS=8<RET> TKB>ASG=SYØ:1:2:3:4, TIØ:5:6, MMØ:7:8<RET> TKB>//<RET>

There are many more task characteristics that can be specified at task-build time. The few mentioned here are the most frequently used. For further information regarding the use of the Task Builder and the options and switches available, refer to the RSX-11M/M-PLUS Task Builder Manual.

LEARNING ACTIVITIES

1. READ the following in the RSX-11M/M-PLUS Task Builder Manual:

   - Chapter 1, Introduction and Command Specification

   - Chapter 1Ø, Switches

     - Section 1Ø.1, Switches

     - Section 1Ø.1.6, /CP--Checkpointable

   - Chapter 11, Options

     - Section 11.1.4, ASG--Device Assignment

     - Section 11.1.9, PAR--Partition

     - Section 11.1.2Ø, PRI--Priority

     - Section 11.1.27, TASK--Task Name

     - Section 11.1.3Ø, UNITS--Logical Unit Usage

2. READ the following sections in the RSX-11M/M-PLUS Command Language Manual:

   - Chapter 6, Linking the Task

3. DO Written Exercises 4 through 13 for this module.

INSTALLING A TASK

Installing a task in the System Task Directory (STD) is done by issuing the INSTALL command. It makes the task known to the system and records the task name, priority, partition and disk address of the task image file in the STD. Before any task can be run, it must be installed. Installing a task does not cause it to execute, but does cause the task's state to be KNOWN and DORMANT. A privileged user can install a task. Because memory is required for each task entered into the STD, the number of installed tasks should be minimized.

To remove a task from the STD, use the REMOVE command, which is privileged.

In Example 8-2, a task image file called PROGRAM.TSK is used to show how to install a task, run it and then remove it from the STD. This task accepts one line of input from the terminal. It then changes the input line from uppercase characters to lowercase. The new line is then output to the terminal.

```
  >
  >
  >
1 >INSTALL/PRIORITY:75 PROGRAM.TSK
  >SHOW TASKS:PROGRA/INSTALLED
  PROGRA 01      GEN      75. 00001700 DR0:-01321644
  >
  >
2 >INSTALL/TASK_NAME:JIM PROGRAM.TSK
  >SHOW TASKS:JIM/INSTALLED
  JIM    01      GEN      50. 00001700 DR0:-01321644
  >
3 >INSTALL/TASK_NAME:...JIM PROGRAM.TSK
  >SHOW TASKS:JIM/INSTALLED
  JIM    01      GEN      50. 00001700 DR0:-01321644
  >SHOW TASKS:...HIM/INSTALLL\L\ED\\\\
  TAS -- Task not in system
4 >SHOW TASKS:...JIM/INSTALLED
  ...JIM 01      GEN      50. 00001700 DR0:-01321644
5 >RUN JIM
  ABCDEFGHIJKL
  abcdefghijkl

6 >JIM
  ABCDEFGHIJKL
  abcdefghijkl
7 >RUN PROGRA
  NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR PARTY.
  now is the time for all good men to come to the aid of their party.

  >
8 >REMOVE JIM
  >REMOVE ...JIM
  >REMOVE PROGRA
  >
  >
  >
```

Example 8-2 Samples of Installing a Task

Notes on Example 8-2

The following comments are keyed to the example.

1. The task image from the file named PROGRAM.TSK is installed with a priority of 75. The task name, by default, the first six characters of the task image file name. The SHOW TASKS command shows that the task will be loaded into partition GEN, has a priority of 75, and the disk address of the image file is DRØ:-Ø1321644. To run the program, use the name PROGRA.

2. The same task is installed again, with a task name of JIM. The priority defaults to the priority with which the task was built. To run the program, use the name JIM.

3. In this case, the task is installed with the name ...JIM. This makes the task MCR spawnable. An MCR spawnable task is one that can be run by issuing just the last three characters of the task name. The RUN command is not used to invoke the task, as you will see in a later example.

4. Notice that SHOW TASK shows a task name of ...JIM. At this point in time, there are three entries in the STD, all pointing to the same task image file. We can run the task in three different ways.

5. To run an installed task that has not been installed as MCR spawnable, you must use the RUN command. In this example, type RUN JIM. The task waits for you to enter the string of characters. After a carriage return, the task converts the string to lowercase and outputs it to the terminal. The job is complete. Whenever you want to run the task again, you must say RUN JIM. There is no need to reinstall the task.

6. This is an example of invoking an MCR spawnable task. You do <u>not</u> have to use the RUN command. MCR utilities are invoked in this manner. For example, PIP, the Peripheral Interchange Program, is invoked by typing PIP. Its task name is ...PIP.

7. This shows how to invoke the task that was installed in note 1 above, using the first six letters of the task name.

8. This shows how to remove the entries from the STD. After their removal, the task's states return to UNKNOWN. Trying to run a removed task would result in the error message:

```
#####INS--file not found.
```

RUNNING A TASK

Tasks can be scheduled to run:

- Immediately
- At a specific time of day
- At some time interval from now
- Repeatedly, at a time interval synchronized on the next hour, minute, second or tick. (A "tick" is one pulse from the system clock. Usually, the system clock is synchronized with the ac line frequency.)

Running Tasks Immediately

Use the following format to run Uninstalled tasks immediately:

```
RUN filename
RUN $filename
```

All tasks must be installed to run on the system. Therefore, this form of the RUN command does an automatic Install-Run-Remove. Note that the command requires a file name, not a task name. The $ in the second example above is a short form representation for the System UFD (M systems) or the System and Library UICs (M-PLUS systems). The system will run the task located in the system UFD rather than one located in the user's default UFD.

Installed Tasks

Installed tasks can be invoked in two ways. If they are MCR spawnable tasks, invoke them by issuing the three character name derived from the last three characters of the task name. For instance, under MCR, the Compare program that compares two ASCII files is invoked by typing CMP. Its task name is ...CMP.

All other installed tasks are invoked using the following format:

```
RUN taskname
RUN $taskname
```

Note that the command requires a task name. The $, once again, means to look on the System UIC (and the library UIC on M-PLUS systems) for the task image file.

Tasks Scheduled to Run Later

Suppose you wish to run a task during nonprime time hours when you are not at work. You can do that using the RUN command, provided that the task is installed and you are a privileged user.

When you schedule a task to run at a later time, an entry is made in the system clock queue. The clock queue is a list of tasks that are waiting for time to expire before they are to run. This queue is checked frequently for tasks whose time has expired and are, therefore, to be activated.

To observe what jobs are in the queue, the SHOW CLOCK_QUEUE command will give information regarding the task. To remove a task from this queue, you must issue the CANCEL command.

In the following examples, the commands show how to run a task at a specific time, a delayed time or on an interval time basis.

```
>RUN/SCHEDULE:11:3Ø:ØØ BATCHRUN
>RUN/DELAY:1ØS VIPROG
>RUN/INTERVAL:1ØM EXERCISER
```

In the first example, an entry is made in the clock queue to run the task BATCHRUN at 11:3Ø. This is an example of running a task at a specific time of day.

In the second example, the command specifies that VIPPROG should be run ten seconds from the time you issue the RUN command.

In the third example, the command specifies that the task EXERCISER should run every ten minutes.

LEARNING ACTIVITIES

1. READ the following sections in the RSX-11M/M-PLUS Command Language Manual.

   - Chapter 7, Running Tasks

     - Section 7.1, Task Installation and Execution
     - Section 7.2, Introduction to the RUN command
     - Section 7.3, ABORT
     - Section 7.5, CANCEL
     - Section 7.8, INSTALL
     - Section 7.9, REMOVE
     - Section 7.12.3, SHOW TASKS
     - Section 7.12.5, SHOW CLOCK_QUEUE

2. DO Written Exercises 14 through 22 for this module.

LIBRARIES

INTRODUCTION

In this module you will learn how to create, maintain and use libraries.

A library file is a direct access file containing a number of modules, usually of the same type. Users can collect often used subroutines and macro definitions for ease of use and faster access during program development.

OBJECTIVES

To use libraries effectively, a user must be able to:

1. Create libraries
2. Maintain libraries
3. Use libraries during program development
4. Use libraries to collect files together

RESOURCES

1. RSX-11M/M-PLUS Command Language Manual
2. RSX-11M Utilities Manual

LIBRARIES

Libraries are specially formatted files that contain a collection of associated files. For example, Object libraries contain object modules for program code that has been processed with a language translator. The object code could be for routines that are commonly used like a READ or WRITE RECORD routine. These object modules are grouped together into one file for the convenience of the user. You save program development time by using code that has already been developed and debugged.

These files are maintained by a task called the Librarian. As the files are specially formatted for quick access and space saving, they cannot be displayed by using the DCL TYPE command. You must use the Librarian task to display the contents of a library file. There are three types of libraries:

- Macro Source libraries
- Object libraries
- Universal libraries

Macro Source libraries contain macro definitions written in the MACRO-11 language. They are source statements. You use these libraries as input files when assembling them as MACRO-11 source files.

Object libraries contain object modules. Object modules are the output from a language translator (like FORTRAN, COBOL, MACRO-11 Assembler). Before being placed in the library, the code has been used in a program and debugged. Object libraries are used at task-build time.

Universal libraries are general-use libraries. They may contain text files, indirect command files or whatever you wish to place in them. They can be used merely as a repository for files, or as in the case of indirect command files, as a central file for frequently used command files.

There are many DIGITAL-supplied System libraries available, and the user may create his own special libraries.

Benefits of Using Libraries

The greatest benefit of libraries comes from sharing code that has already been developed and debugged. Development costs are reduced when a programmer uses existing code.

Second, the MACRO-11 Assembler and the Task Builder automatically search the System library. After processing all the input files, both tasks search a System library for any unresolved references. The System library specification is not included in the input command line. User libraries, which may also be used with both of these tasks, however, are not automatically searched. Their file specifications must be included in the input command line.

Finally, having all modules in the same location saves time and overhead. Control can be applied to libraries; this assures that the latest version of a routine is being used. Only one entry for each module can be placed in a library. If a user wishes to have other versions of a routine maintained in a library, he must insert them with a different module name, or place them in another library.

Library File Format

All library files have the same format, as in Figure 9-1.

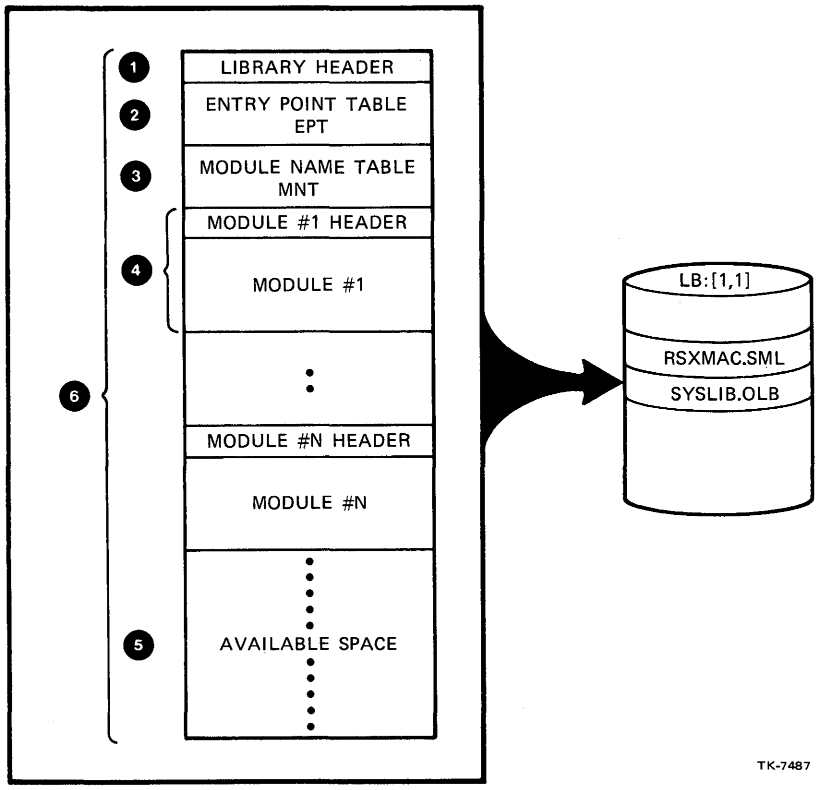


Figure 9-1 Library File Format

Notes on Figure 9-1

The following comments are keyed to the figure.

❶ Library Header

This section contains information on the current library status including the date and time of the last addition, the number of Entry Point Table entries, the number of Module Name Table entries, the amount of available space, and the number of logically deleted bytes. For Universal libraries only, the default file type is kept in the header.

❷ Entry Point Table

The Entry Point Table is used only with Object libraries. It is an alphabetic list of all the entry points in all the object modules in the library.

An entry point is an address within a piece of code to which a program can transfer control such as:

- a global label in MACRO
- a subroutine
- a function
- block data

This table is used to locate a piece of code in the library, when a request is made with the entry point name. See Table 9-1 for the source of the Entry Point Table name.

❸ Module Name Table

The Module Name Table contains the name of every module in the library. It is used to find the module in the library. See Table 9-1 for the source of the Module Name Table entry.

❹ Header and module body for every module inserted into the library

❺ Space available

❻ Total library size set at creation time. If space exhausted, library must be rebuilt.

Table 9-1 Sources of EPT, MNT Entries Used in Creating a Library File

Library Type	Source of EPT Entry		Source of MNT Entry	
Macro	Not Used		Macro Name	
Object	FORTRAN	MACRO	FORTRAN	MACRO
	Subroutine Function Name	Global Labels	Main PGM Subroutine Name Function Block Data	.TITLE STATEMENT
Universal	Not Used		First Six Char File Name	

LEARNING ACTIVITIES

1. READ the following sections in Chapter 1Ø of the RSX-11M/M-PLUS Utilities Manual:

   - 1Ø.1, Format of the Library File
   - 1Ø.2, LBR Restrictions

2. READ Section 6.3, Library, in Chapter 6 of the RSX-11M/M-PLUS Command Language Manual.

3. DO Written Exercises 1 through 4 for this module.

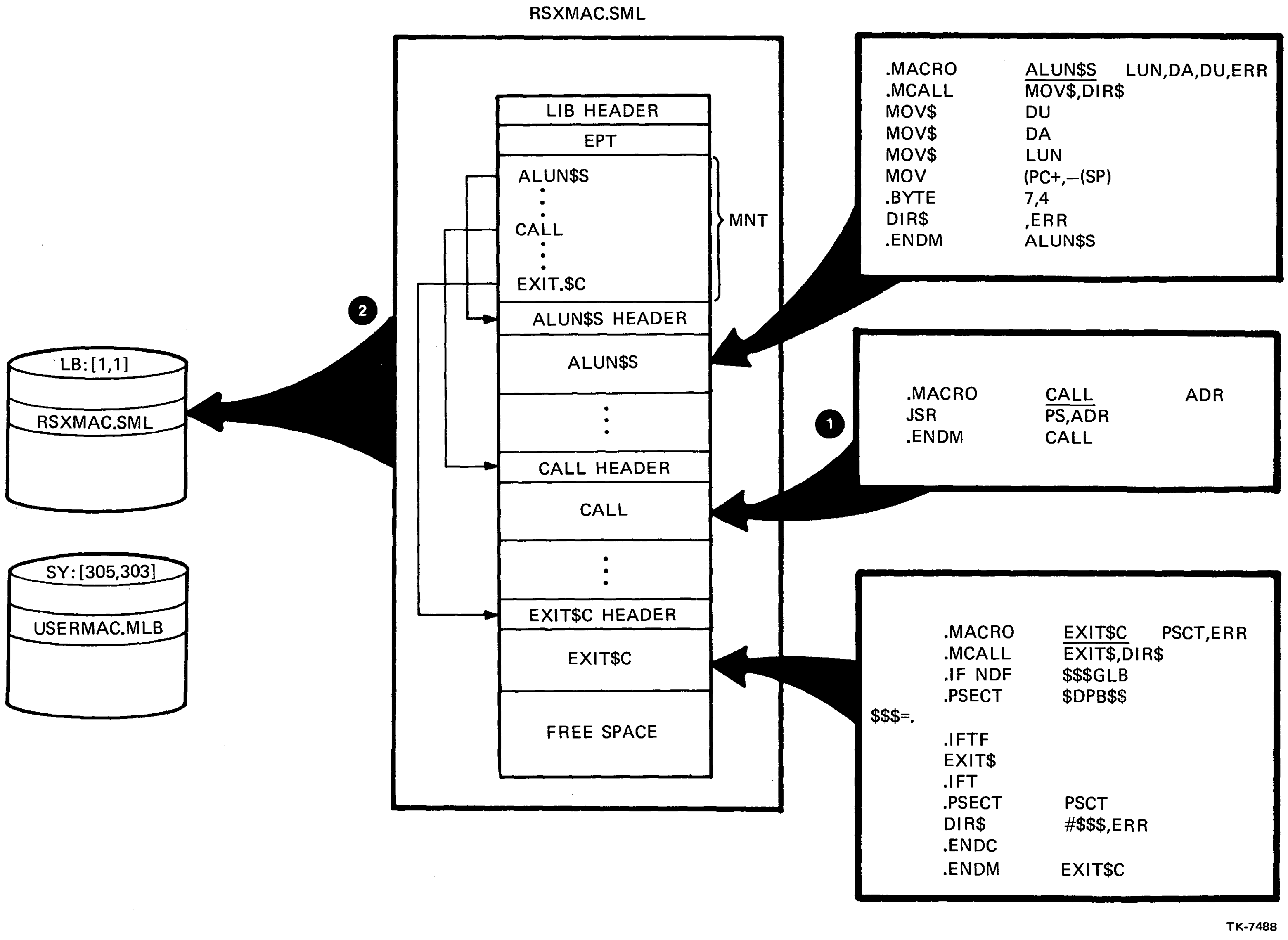
Macro Libraries

Figure 9-2 is an example of the type of information stored in a Macro library. There are three macro definitions shown here, ALUN$S, CALL, and EXIT$C. Each macro consists of all the instructions between the .MACRO and the .ENDM statements, and is written in MACRO-11 assembly language. Each of these was developed and tested before it was put into the library.

The format of the Macro library consists of the library header, and the Module Name Table (MNT), followed by a module header and module for each macro definition.

The module header contains information regarding the module itself. The size of the module, its status, and the date the module was inserted is stored here. For more detailed information regarding the module header format, refer to the RSX-11M/M-PLUS Utilities Manual.

The default file type for a Macro library is .MLB. The System Macro library is RSXMAC.SML, and is located on LB:[1,1].



```
.MACRO          ALUN$S    LUN,DA,DU,ERR
.MCALL          MOV$,DIR$
MOV$            DU
MOV$            DA
MOV$            LUN
MOV             (PC+,-(SP)
.BYTE           7,4
DIR$            ,ERR
.ENDM           ALUN$S
```

```
.MACRO          CALL            ADR
JSR             PS,ADR
.ENDM           CALL
```

```
          .MACRO          EXIT$C    PSCT,ERR
          .MCALL          EXIT$,DIR$
          .IF NDF         $$$GLB
          .PSECT          $DPB$$
$$$=.
          .IFTF
          EXIT$
          .IFT
          .PSECT          PSCT
          DIR$            #$$$,ERR
          .ENDC
          .ENDM           EXIT$C
```

Figure 9-2 A Macro Library

Object Libraries

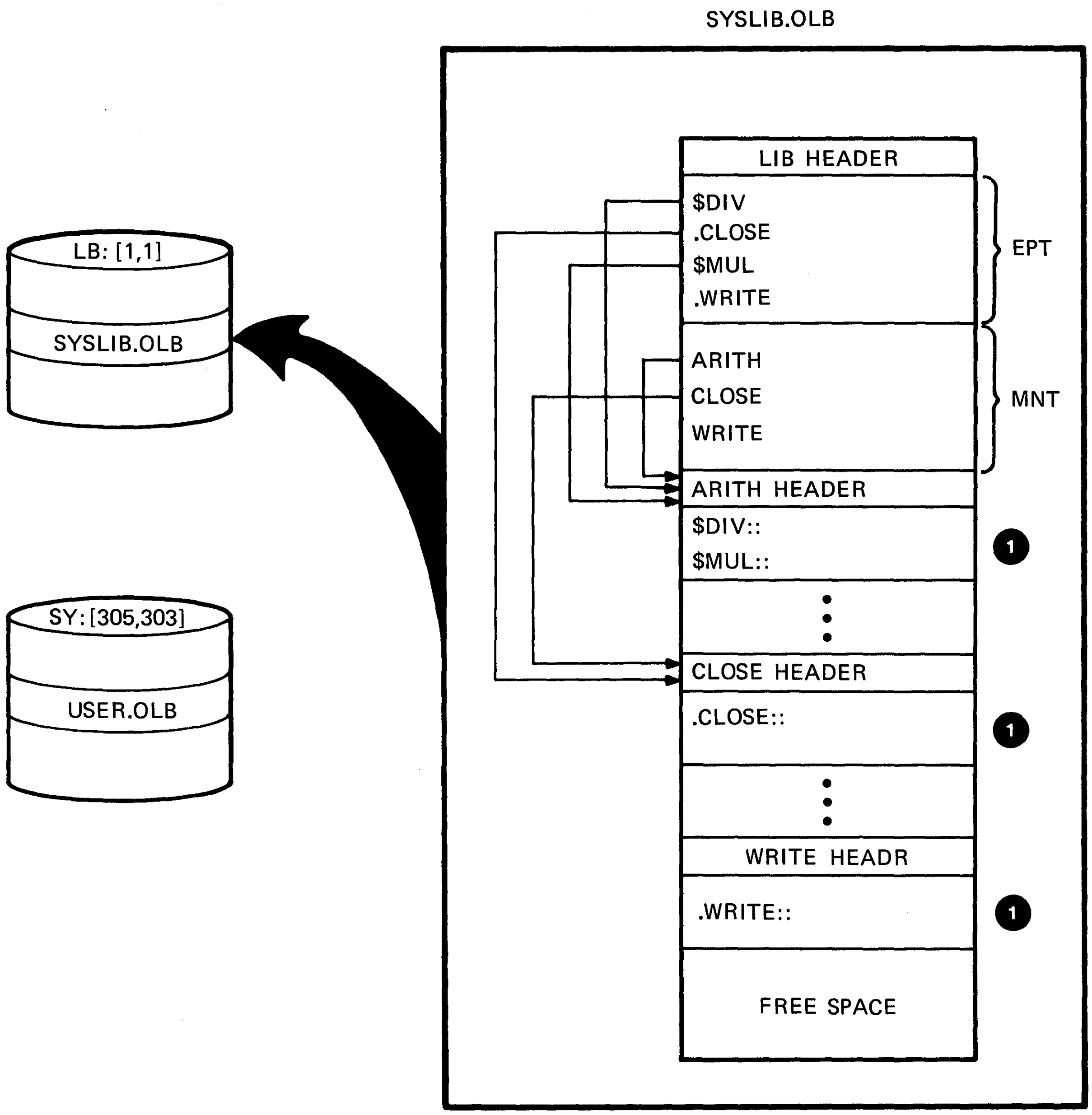
Figure 9-3 shows the format of an Object library.

An object module is difficult to visualize as it is not in human-readable form. Each object module contains instructions in object code. Each requires linking before it can be run.

In the figure, there are three object modules in the library: ARITH, CLOSE and WRITE. ARITH has two entry points, $DIV:: and $MUL::. CLOSE and WRITE have one each, .CLOSE:: and .WRITE::. Notice that the EPT has an entry for each entry point in the library, and a pointer to the header of the respective object module, where it can be located. The Object library is the only one of the three library types to use the Entry Point Table (EPT). Entry points tell where a program will transfer control when the program executes.

The default file type for an Object library is .OLB. The System Object library is LB:[1,1]SYSLIB.OLB. After processing all the input files, the Task Builder searches through this library for any unresolved references.

The Task Builder is able to search an Object library in two ways. The first way is when it tries to resolve unsatisfied references on global symbols. The global symbol is looked for in the Entry Point Table. The second way is when a module name has been specified with the /LIBRARY/INCLUDE qualifier of the LINK command. In both cases, when the module is found, the Task Builder extracts the complete module and includes it in the task image file.



1 OBJECT MODULES DERIVED FROM AN ASSEMBLER OR COMPILER

2 OBJECT FILE FORMAT INCLUDES ENTRIES IN THE EPT.

TK-7495

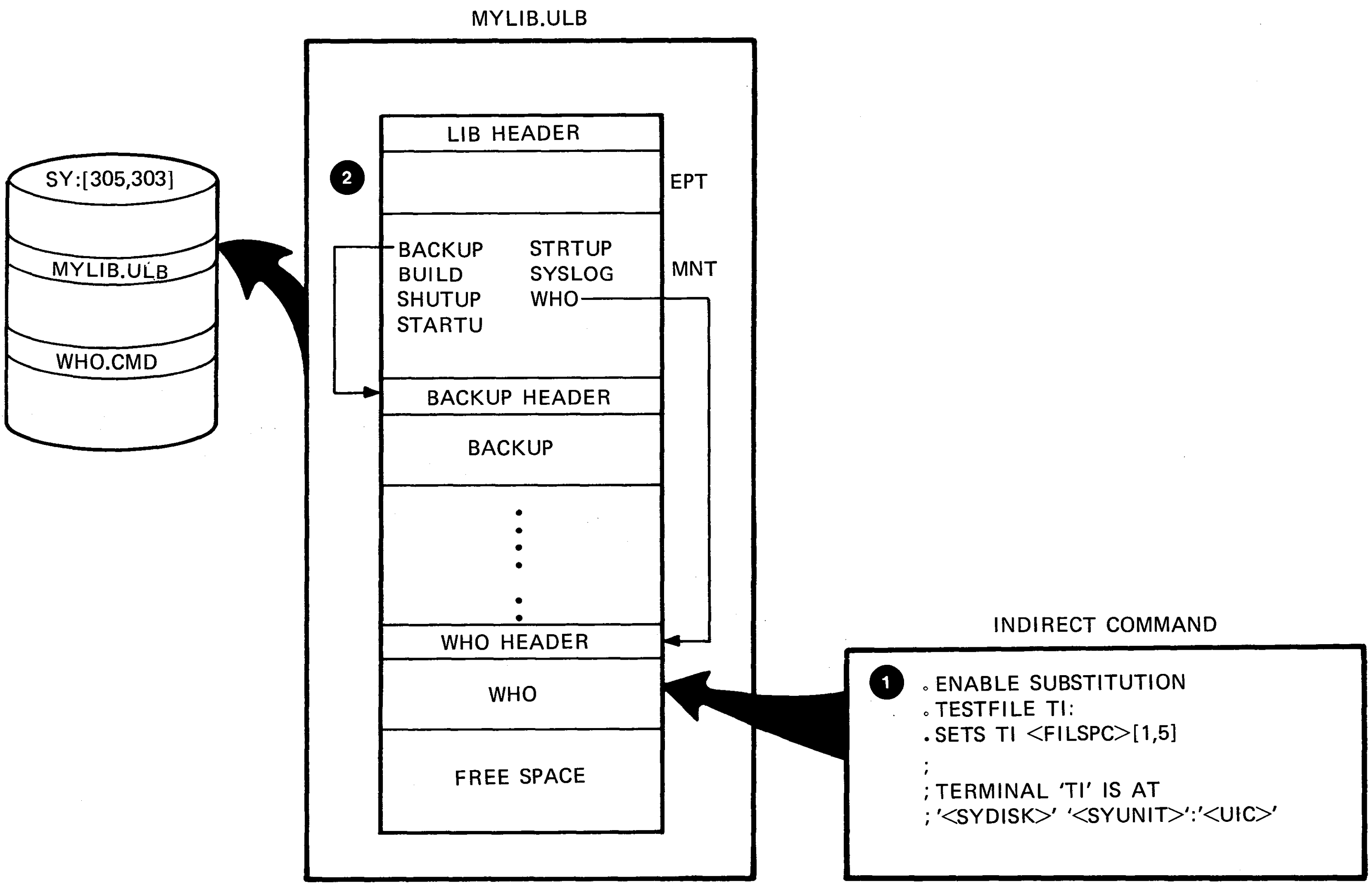
Figure 9-3 Object Libraries

Universal Libraries

Universal libraries have the same format as Macro libraries. Like the Macro library, however, the EPT is not used. Figure 9-4 shows this format and the type of input that a Universal library may have. This particular library is a collection of indirect command files that can be executed directly from the library. They do not need to be extracted first.

The Universal library can be used to collect listing files with their source and object files. This is one method of transporting a program package from one system to another. When transferring the file from one media to another, only one file needs to be specified in the copy operation. Universal libraries can be used to collect memos on a certain topic, like your programming project.

The default file type for Universal libraries is .ULB.



1 ENTRIES INTO THE LIBRARY MAY BE OF A UNIVERSAL TYPE. THIS ONE CONTAINS INDIRECT COMMAND FILES.

2 LIBRARY FILE FORMAT - EPT IS NOT USED.

TK-7490

Figure 9-4 Universal Libraries

DIGITAL supplies many libraries for your use. Table 9-2 lists the libraries that are available. Not all will be present on your system. You may perform a directory command on LB:[1,1] to see which ones your system does have. The most frequently used libraries are RSXMAC.SML, SYSLIB.OLB and the FORTRAN Object Time System libraries.

Table 9-2 DIGITAL-Supplied Libraries on LB:[1,1]

Library File Name/Type	Description of Contents
RSXMAC.SML	Default System Macro Library
EXEMAC.MLB	Executive Macro Library
RMSMAC.MLB	Record Management System RMS-11
SYSLIB.OLB	Default System Library
VMLIB.OLB	Virtual Memory Management Library
EXELIB.OLB	Executive Library
ANSLIB.OLB	ANSI Magnetic Tape Library (M only)
RMSLIB.OLB	Record Management Library
FOROTS.OLB	FORTRAN IV OTS Library
F4POTS.OLB	FORTRAN IV-PLUS OTS Library
F77OTS.OLB	FORTRAN-77 OTS Library
VMLIB.OLB	Virtual Memory Management

Using Macro Libraries

Figure 9-5 illustrates how to use Macro libraries.

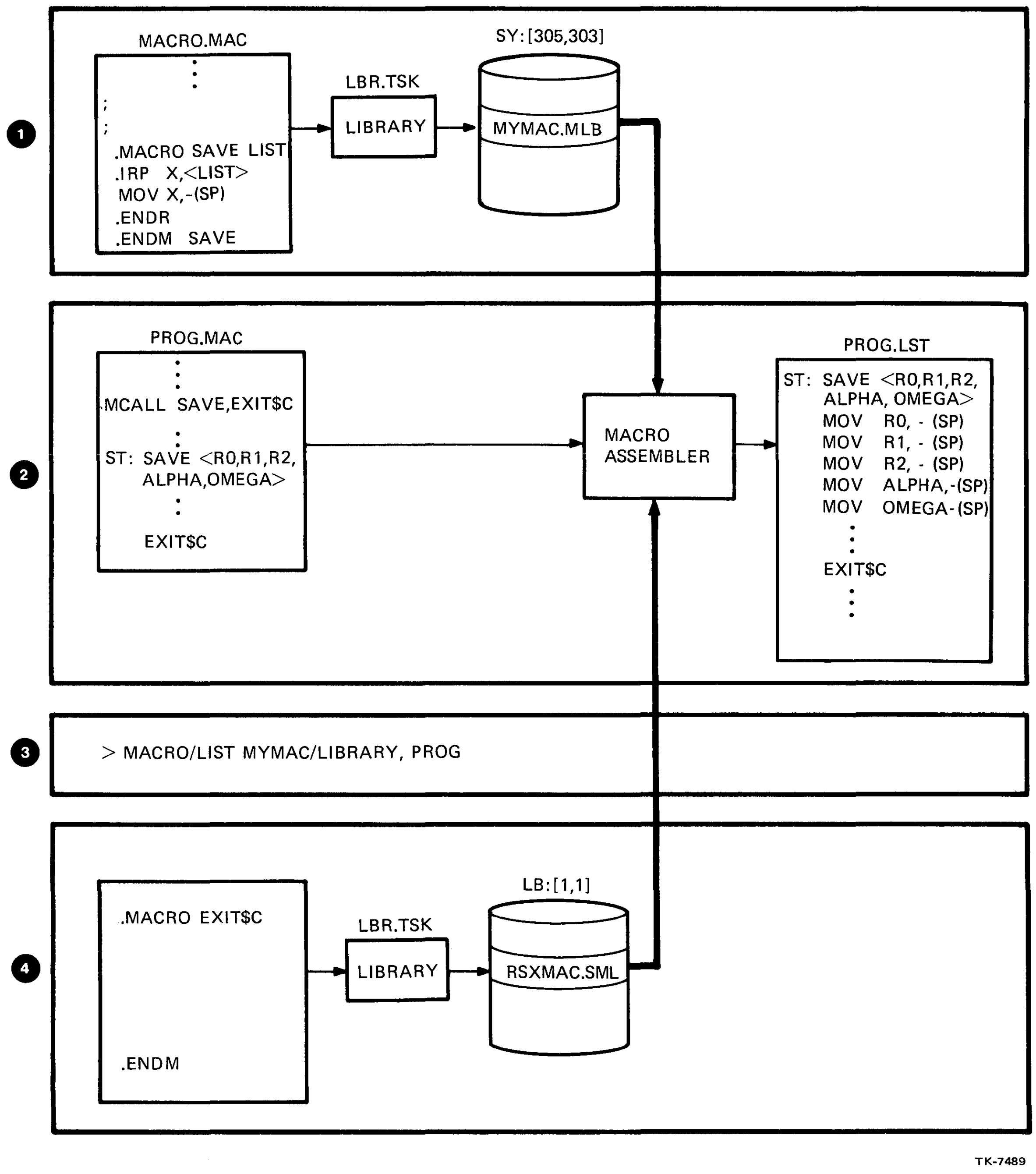


Figure 9-5 Using Macro Libraries

Notes on Figure 9-5

The following comments are keyed to the figure.

1. Using the Librarian, put the Macro source in a library. In this example, the Macro SAVE is added to the User library MYMAC.MLB.

2. Reference the Macro in a source file. In a source program, PROG.MAC, reference is made to the SAVE macro. EXIT$C is also referenced.

3. Assemble the source referencing the library containing the Macro. The Macro Assembler searches MYMAC.MLB for the macro definition for SAVE and finding it expands the macro, making the proper symbol substitutions. The Macro EXIT$C is not found in this library and remains unsatisfied.

4. Previous to the assembly process, the System Macro library, RSXMAC.SML, was created using the Librarian. The Macro EXIT$C is contained in this library. Once the assembler searches any User libraries specified in the MACRO command line, it turns to the System Macro library to resolve any other unsatisfied references. In this case, EXIT$C is found, extracted from the library and included in the source program. The MACRO-11 Assembler searches the User library for unsatisfied references, then automatically searches the System Macro library, LB:[1,1]RSXMAC.SML.

Using Object Libraries

The procedure for using Object libraries is similar to that of Macro libraries, however, their use occurs later in the program development process.

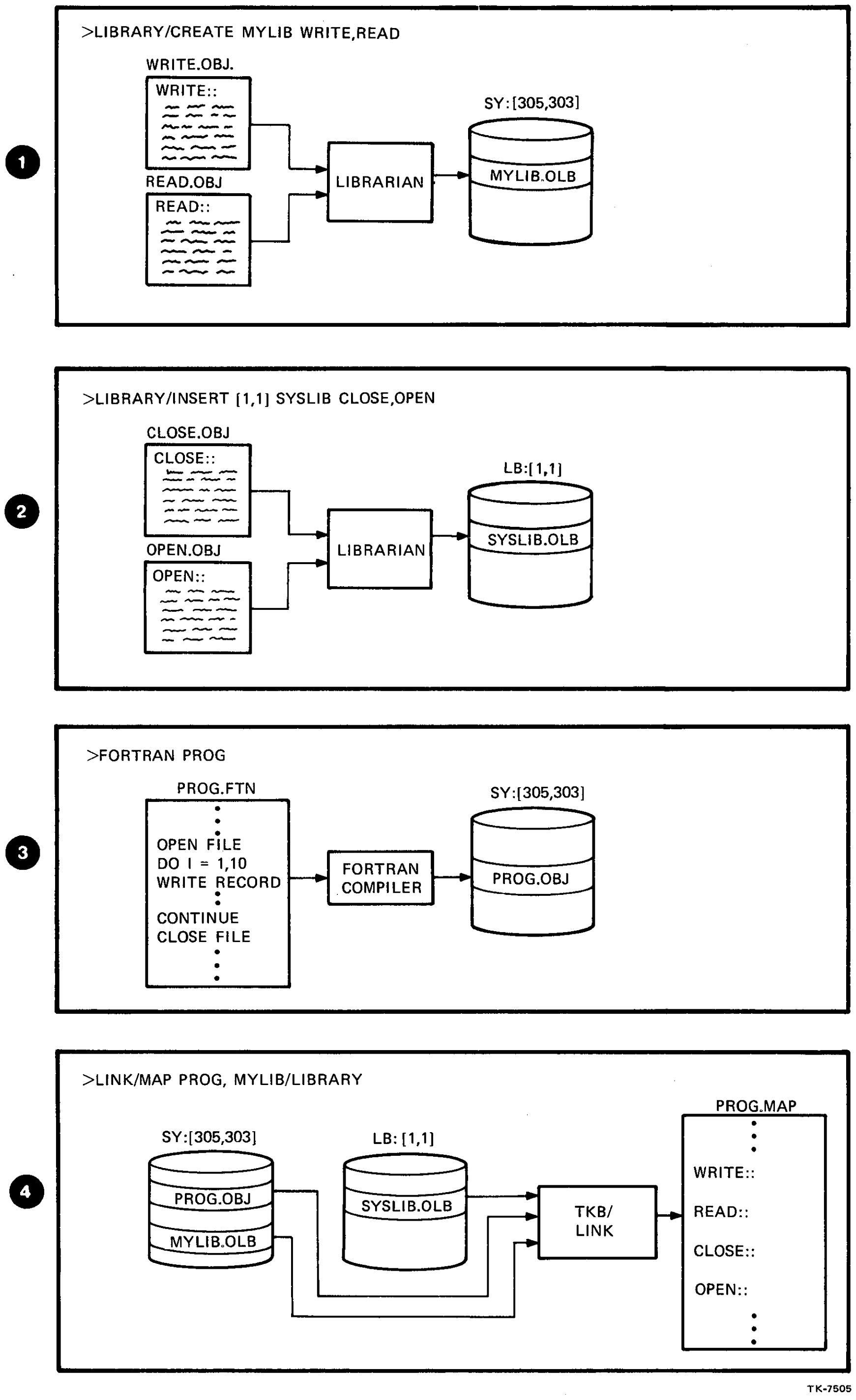


Figure 9-6 Using Object Libraries

Notes on Figure 9-6

The following comments are keyed to the figure.

1. A User Object library called MYLIB.OLB is created using the Librarian. Two object modules (WRITE, READ) contained in two object files (WRITE.OBJ and READ.OBJ) are placed in the library when it is created.

2. The System Object library, SYSLIB.OLB, is modified to include two object modules (CLOSE and OPEN) which are contained in two object files (CLOSE.OBJ and OPEN.OBJ).

3. In a source file, PROG.FTN, reference to three of those modules is made (OPEN, WRITE and CLOSE). The file is compiled. A listing file generated from the compile shows those three references as unsatisfied, as they are not defined within this source file.

4. At link time, the User Object library, MYLIB.OLB, is supplied in the command line. The Task Builder reads this library first, searching for code for OPEN, WRITE and CLOSE. Only the code for WRITE is found, extracted and included in the Task Build from this library. The Task Builder then searches the SYSLIB.OLB for CLOSE and OPEN.

   Be careful in supplying library specifications in the LINK command line. If the User library contains object modules for CLOSE and OPEN, the Task Builder would extract these modules to satisfy the references to those symbols. If the desired code is in the System library, the results then are wrong. For a solution to the situation, read the DCL HELP provided on the /LIBRARY/INCLUDE qualifier to the LINK command, or refer to the /LB switch discussion in the RSX-11M/M-PLUS Task Builder Manual.

Using Universal Libraries

When universal libraries are used as repositories for text files, you create them by using the Librarian and inserting the text modules into them. The module names will be taken from the input file name. You can then use the librarian commands to list the contents, extract or insert new modules. If the library is a collection of indirect command modules, a particular command module can be executed without extracting it from the library using the following command:

```
>@MYLIB.CLB/LB:WHO (MCR command)
```

In this example, the library file, MYLIB.CLB, is used with the /LB switch to specify the indirect command module (WHO.CMD) to be executed.

Universal libraries can also be accessed from a task by building that capability into the task in the following manner:

- Build a Universal library containing the desired modules.
- Develop your application program using $ULA, a System library routine, to access the library.
- Assemble/compile, task-build and run the application program.

For more information regarding $ULA, refer to Appendix B of the IAS/RSX-11M System Library Routines Reference Manual.

Creating/Maintaining Libraries

Table 9-3 lists operations using the Librarian. The DCL command is shown, along with a command example. Table 9-4 shows the equivalent MCR commands to accomplish the same operations.

Table 9-5 lists CREATE command parameters and qualifiers, while Table 9-6 lists Librarian listing qualifiers.

Table 9-3 DCL Library Commands

	Creating a New Library
DCL Command	`>LIBRARY/CREATE[:ARG][/QUALIFIER[S]] LIBSPEC [INFILE[S]]`
DCL Example	`>LIBRARY/CREATE/BLOCKS:200. MYLIB`
Note	See Table 9-5
	Listing the Contents of a File
DCL Command	`>LIBRARY/LIST[:FILESPEC][/QUALIFIER[S]] LIBSPEC`
DCL Example	`>LIBRARY/LIST/FULL MYLIB`
Note	See Table 9-6
	Adding a Module to a Library
DCL Command	`>LIBRARY/INSERT[/QUALIFIER[S]] LIBSPEC FILESPEC[S]`
DCL Example	`>LIBRARY/INSERT UNILIB.ULB LOGIN.CMD,LOGOUT.CMD`
	Replacing a Module in a Library
DCL Command	`>LIBRARY/REPLACE[/QUALIFIER[S]] LIBSPEC FILESPEC[S]`
DCL Example	`>LIBRARY/REPLACE UNLIB.ULB LOGIN.CMD,LOGOUT.CMD`
	Deleting a Module from a Library
DCL Command	`>LIBRARY/DELETE LIBSPEC MODULE[S]`
DCL Example	`>LIBRARY/DELETE MYLIB A,B,C`
	Compressing a Library
DCL Command	`>LIBRARY/COMPRESS[:ARG][/QUALIFIER[S]] OLDLIBSPEC NEWLIBSPEC`
DCL Example	`>LIBRARY/COMPRESS:BLOCKS:200. MYLIB MYLIB2`
Note	See Figure 9-6
	Extracting a Module
DCL Command	`>LIBRARY/EXTRACT/OUTPUT:FILESPEC LIBSPEC MODULE[S]`
DCL Example	`>LIBRARY/EXTRACT/OUTPUT:A MYLIB.MLB A`

Table 9-4 Equivalent MCR Library Commands

Creating a New Library

```
>LBR MYLIB.OLB/CR::::OBJ=OPEN,CLOSE,WRITE,READ
```

Listing the Contents of a File

```
>LBR MYLIB.OLB/LI
>LBR MYLIB.OLB/LE
>LBR MYLIB.OLB/FU
```

Adding a Module to a Library

```
>LBR MYLIB.OLB/IN=GET,PUT
```

Replacing a Module in a Library

```
>LBR MYLIB.OLB/RP=OPEN,CLOSE
```

Deleting a Module from a Library

```
>LBR MYLIB/DE:GET,PUT
```

Compressing a Library

```
>LBR MYLIP/CO:::=MYLIB
```

Extracting a Library Module

```
>LBR CLOSE.OBJ=MYLIB/EX:CLOSE
```

Table 9-5 CREATE Command Parameters and Qualifiers

/CREATE Parameters

Parameter	Default
BLOCKS:N	BLOCKS:1ØØ.
MODULES:N	MODULES:256.
GLOBALS:N	MACRO UNI GLOBALS:Ø OBJ - GLOBALS:512.

/CREATE Qualifiers

Qualifier	Default	Library Type	Default File Ext
/OBJECT	YES	Object Library	.OLB
/MACRO	NO	Macro Library	.MLB
/UNIVERSAL	NO	Universal Library	.ULB

Table 9-6 Librarian Listing Qualifiers

Qualifier	Function
/BRIEF	Lists Space Available and Module Names Default
/NAMES	- Space Available - Module Names - Entry Points
/FULL	- Space Available - Module Names - Full Description of Each Module

LEARNING ACTIVITIES

1. READ the following sections in Chapter 1Ø of the RSX-11M/M-PLUS Utilities Manual:

   - 1Ø.5.1, Compress Switch
   - 1Ø.5.2, Create Switch
   - 1Ø.5.3, Delete Switch
   - 1Ø.5.7, Extract Switch
   - 1Ø.5.8, Insert Switch for Object, Macro Libraries
   - 1Ø.5.9, Insert Switch for Universal Libraries
   - 1Ø.5.1Ø, List Switches
   - 1Ø.5.12, Replace Switch for Object, Macro Libraries
   - 1Ø.5.13, Replace Switch for Universal Libraries

2. DO Written Exercises 5 through 11 for this module.

Examples 9-1 and 9-2 show some frequently used LIBRARY commands.

```
    >
    >
    >
❶   >LIBRARY/LIST LB:[1,1]RSXMAC.SML

    Directory of file RSXMAC.SML;121
    Macro library created by:  LBR V06.00
    Last insert occurred 23-AUG-81 at 13:04:21
    MNT entries allocated: 512; Available: 93
    EPT entries allocated: 0; Available: 0
    File space available: 01193 words
    Recoverable deleted space: 00179 words

    ABRT$
    ABRT$C
    ABRT$S
    AFF$
    ALTP$
    ALTP$C
    ALTP$S
    ALUN$
    ALUN$C
    ALUN$S
    ASTX$
    ASTX$C
    ASTX$S
    ATRG$
    ATRG$C
    ATRG$S
    BDOFF$
    CALL
    CALLR
    >
    >
❷   >LIBRARY/LIST/FULL LB:[1,1]RSXMAC.SML

    Directory of file RSXMAC.SML;121
    Macro library created by:  LBR V06.00
    Last insert occurred 23-AUG-81 at 13:04:21
    MNT entries allocated: 512; Available: 93
    EPT entries allocated: 0; Available: 0
    File space available: 01193 words
    Recoverable deleted space: 00179 words

    ABRT$    Size:00096  Inserted:7-AUG-81

    ABRT$C   Size:00096  Inserted:7-AUG-81

    ABRT$S   Size:00068  Inserted:7-AUG-81

    AFF$     Size:00101  Inserted:7-AUG-81

    ALTP$    Size:00112  Inserted:7-AUG-81

    ALTP$C   Size:00104  Inserted:7-AUG-81

    ALTP$S   Size:00078  Inserted:7-AUG-81
```

Example 9-1 Obtaining Library Directories (Sheet 1 of 2)

❸
```
>LIBRARY/LIST/NAMES LB:[1,1]SYSLIB.OLB

Directory of file SYSLIB.OLB;33
Object module library created by:  LBR V06.00
Last insert occurred 11-AUG-81 at 10:19:25
MNT entries allocated: 768; Available: 524
EPT entries allocated: 2304; Available: 964
File space available: 00675 words


** Module:ALERR

  $ALERR


** Module:ALSCT

  ALSCT


** Module:ALTPRI

  ALTPRI


** Module:ANSPAD


** Module:ARITH

  $DIV    $MUL
```

Example 9-1 Obtaining Library Directories (Sheet 2 of 2)

Notes on Example 9-1

The following comments are keyed to the example.

❶ To list the contents of a library you use the /LIST qualifier on the LIBRARY command. The library, file specification is given as a parameter. The output from issuing this command consists of a short statistical section and then the list of names of the modules contained in the library. This example lists the contents of the System Macro library. The module names are listed in ascending order. A <CTRL/O> was typed before the output was finished, so this is not a complete list.

❷ By using the /FULL qualifier, additional information is provided on the modules. Their size and the date they were inserted into the library are displayed.

❸ To obtain the entry point names in an object module, the /NAMES qualifier is used. The entry point name appears after the module name. In this example, module ARITH has two entry points, $DIV and $MUL.

```
  >
  >
  >
❶ >LIBRARY/CREATE/MACRO MYLIB FILES



❸ >LIBRARY/DELETE MYLIB.MLB MVB$,OFF$
  Modules deleted:
  MVB$
  OFF$

❹ >LIBRARY/REPLACE MYLIB.MLB FILES
  Module 'CALL  ' replaced

  Module 'QIOW$ ' replaced

  Module 'EXIT$C' replaced




❷ >LIBRARY/LIST MYLIB.MLB

  Directory of file MYLIB.MLB;1
  Macro library created by:  LBR V06.00
  Last insert occurred 3-SEP-81 at 14:54:40
  MNT entries allocated: 256; Available: 251
  EPT entries allocated: 0; Available: 0
  File space available: 23877 words

  CALL
  EXIT$C
  MVB$
  OFF$
  QIOW$
```

Example 9-2 Commands to Create a Library and Delete Modules from that Library

Notes on Example 9-2

❶ This is an example of the command you issue to create a library. If you do not specify the libary type, the Librarian assumes an Object library. In this case, the Librarian creates a Macro library with a file name of MYLIB.MLB. FILES is the name of the input object file (FILES.OBJ) containing the modules to be inserted into the library. The size of the resulting library file will be 100 blocks, with space in the MNT for 256 entries. These are the default values for creating a Macro library (See Table 9-5). If these values are not suitable for your library, you must supply the proper values with the /CREATE qualifier.

❷ Listing the contents of the library shows that there are five macro definition modules in the library: CALL, EXITC, MVB, OFF$ and QIOW$.

❸ To delete modules from a library, a list of module names is supplied after the library file specification. When you delete a module from the library, it is only logically deleted from the file. The space in which it resides is not released for use until the file is compressed. Compressing a library file removes deleted modules, and compresses the remaining modules toward the beginning of the file. Free space is then generated at the end of a file for inserting new modules.

❹ When you want to place a newer version of a module in the library, you must use the /REPLACE qualifier. The input file, FILES.OBJ, contains newer versions of CALL, QIOW$ and EXIT$C that will replace the respective modules in the library.

ADVANCED MAINTENANCE OPERATIONS

INTRODUCTION

You have studied some of the most common operations involving files and storage volumes. We will now look at some of the more advanced operations.

Until now, it was assumed that all the files you would access would be on your default volume, or on some other volume defined by your system manager as public (i.e., available to anyone). For reasons of security or convenience, you may instead wish to access your files on a private volume. Or, you may wish to set up a shareable volume that certain chosen users can access. This module shows you how to set up, use and maintain private and shareable volumes.

It was also assumed that you would deal solely with a single RSX-11M and/or RSX-11M-PLUS system. Transferring files to another operating system, or between different RSX-11M/M-PLUS systems, involves access to volumes with a different format than FILES-11. These volumes are called foreign volumes. Accessing foreign volumes is also covered in this module.

OBJECTIVES

1. Prepare and maintain private volumes for file storage.

2. Use the file transfer utility to transfer files between computer systems and convert file formats.

3. Print files and control the print queue.

4. Use the dump utility to inspect the contents of a file.

5. Use the compare utility to segregate the differences between two ASCII files.

RESOURCES

1. RSX-11M/M-PLUS Utilities Manual

2. RSX-11M/M-PLUS Command Language Manual

VOLUME MAINTENANCE

Up until now, you have been using a volume that was initialized, maintained, and backed up by someone else, most likely your system manager. However, the opportunity may arise when you will want to use a private volume to store files for special applications, or for transferring files to other computing systems. To do this, you must know about the utility tasks available for maintaining a volume. In this section you will learn about the volume maintenance utilities (FMT, BAD, BRU, DSC, VFY), a more detailed understanding of the file structure, and device and volume accessibility.

Device Types

There are two types of hardware devices that the operating system must control: record-oriented and file-structured devices.

Record-oriented devices, like line printers, terminals and card readers, process records one at a time. For example, the operating system transfers data to the line printer 132 characters at a time. After the transmission completes, there is no way to access that record again. The printer does not collect each of these records into a file. Storage of these records occurs off-line, on a piece of paper.

On file-structured devices like disk drives, magnetic tape drives, and DECtape drives, the operating system is able to collect records into a file and store them for future use. One structure is used for storing and retrieving files on all file-structured devices.

FILES-11 Volume Structure

FILES-11 is a software system that creates and maintains the file structure on disks, DECtape and floppy disks. It uses a two level directory structure to organize and maintain files on a volume. The first level structure is the Master File Directory (MFD). The second level is the User File Directory (UFD). This structure is maintained by the Ancillary Control Processor task, F11ACP, part of the FILES-11 software system. It performs the following tasks:

- Locates logical blocks on disk
- Allocates storage for files
- Reads and writes file attributes
- Controls access to files
- Performs MOUNT and DISMOUNT operations for FILES-11 volumes

To locate a file on a volume, the system must first check the MFD, ([Ø,Ø]ØØØØØØ.DIR), to see if the UFD specified in the file specification actually exists on the volume. If there is no entry in the MFD for the specified UFD, the system returns an error message. If there <u>is</u> an entry, the system then looks at the directory file for the specified UFD, to see if the requested file actually exists.

Volume Files

A new volume must be prepared before the operating system can access it. One of the preparation steps is to establish the FILES-11 structure by creating the five standard system files that must be present on each volume. You do this with the INITIALIZE command, which creates the five files listed in Figure 10-1 under the special UFD [0,0].

Once these files are present on the volume, you can use the CREATE/DIRECTORY command to create any needed UFD.

Once the UFDs are established, files can be created under each by using the editor, copying files from another volume, or by program execution.

	File Name	Contents
1	INDEXF.SYS	Index File
2	BITMAP.SYS	Storage-allocation file
3	BADBLK.SYS	Bad block file
4	000000.DIR	Master File Directory (MFD)
5	CORIMG.SYS	System Checkpoint File

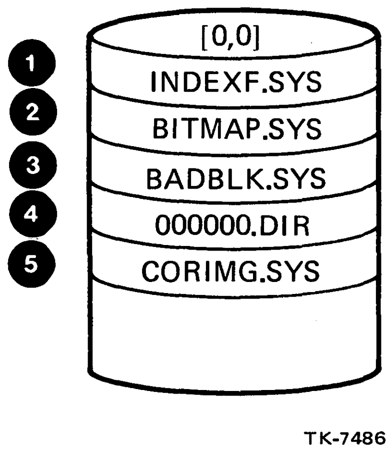


Figure 10-1 FILES-11 Standard System Files Found on Every Volume

Index Files

The first system file in Figure 1Ø-1 is the index file, the most important file on the volume. It is the master key for accessing any file.

Figure 1Ø-3 shows the format of the index file, and contains the following information:

- Volume Information (Home Block)
  - Volume Name
  - Device Type
  - Volume Owner's UIC
  - Volume Protection Code
  - Default File Protection
- File Header Blocks
  - One for each file on volume
  - 256 words long
  - Contains:

    File ID number to indicate which file header to use to locate the file on the volume

    File sequence number to verify correct header

Every file is made up of two parts, as shown in Figure 1Ø-2. The body of the file, which can be stored anywhere on the disk volume, contains the data. A system, called INDEXF.SYS, contains the second part of a file, the file header. Every file on the volume has a header that the system uses to locate the body of the file. Before locating any file on the volume, the INDEXF.SYS file must be read to obtain the necessary information to locate the body of the file.

Figure 1Ø-3 shows the format of the INDEXF.SYS file. Notice that headers exist for the five system files that must be present on every volume, the MFD, all UFDs, and user files present on the volume.

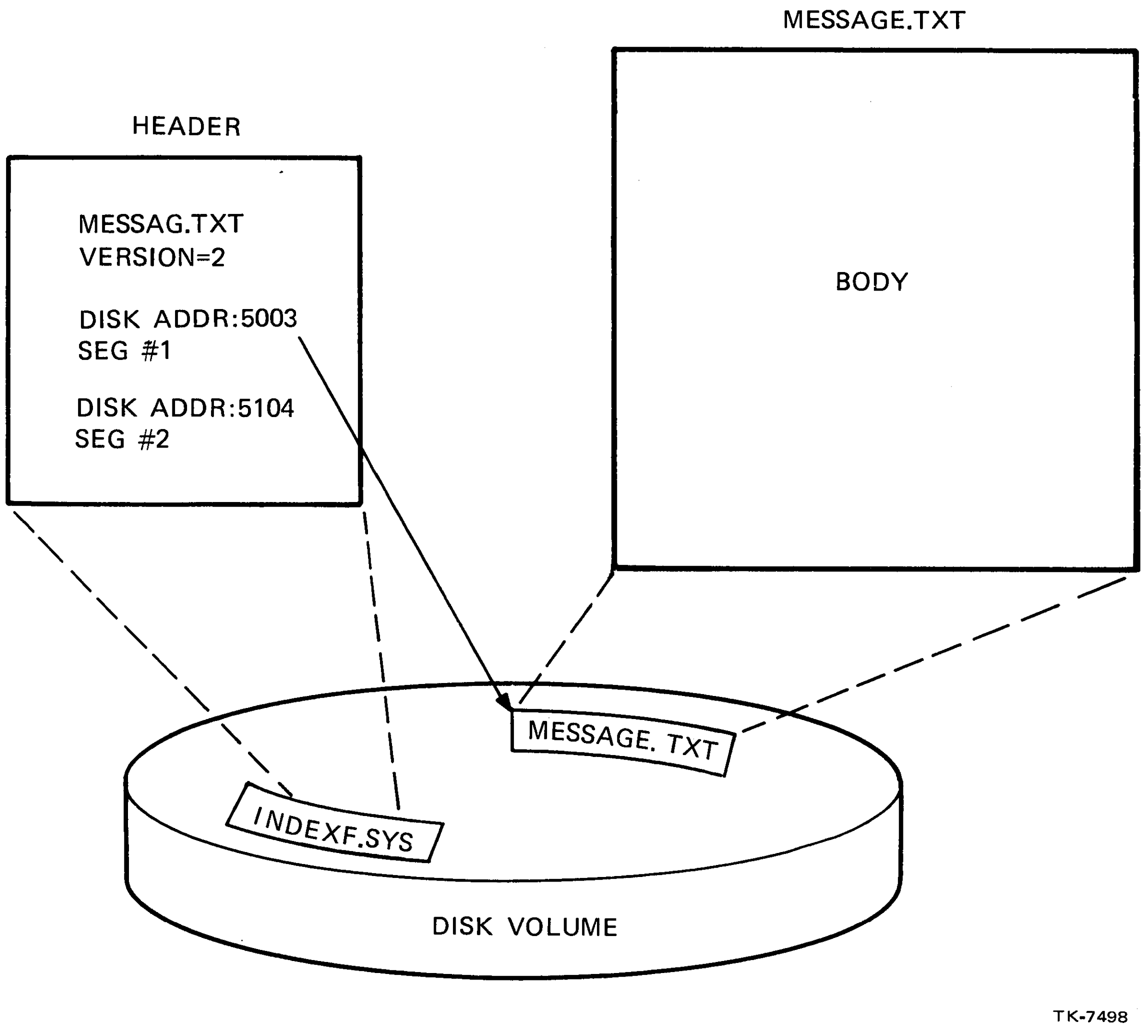


Figure 1Ø-2 The Two Parts of a File

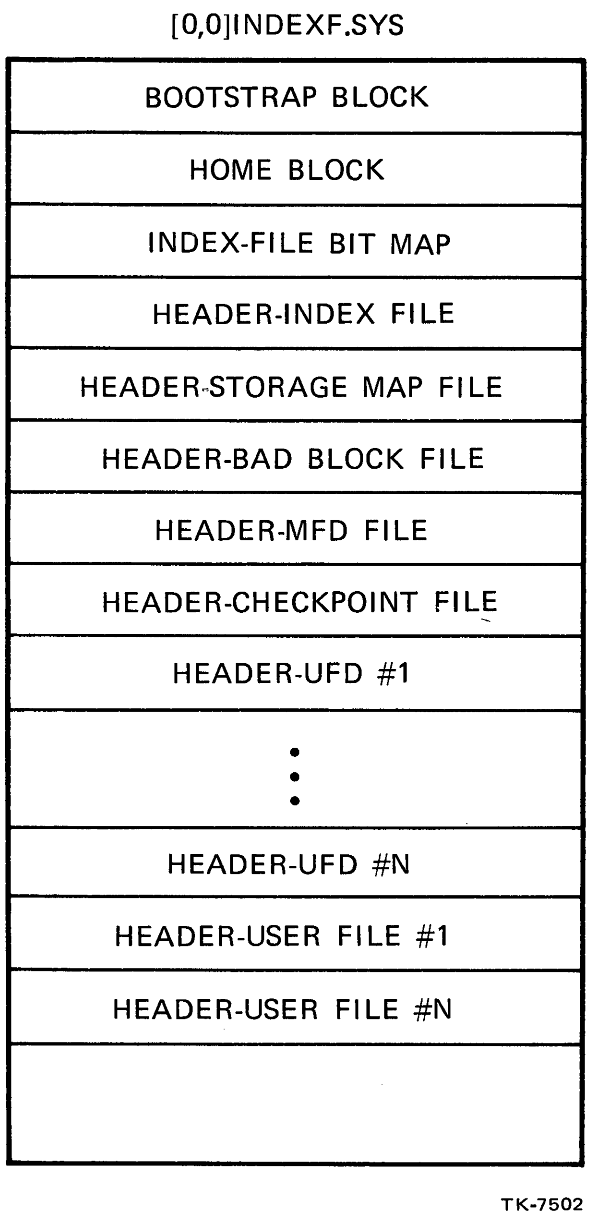


Figure 1Ø-3 Index File Format

Bitmap Files

The second system file, BITMAP.SYS, is the file the operating system uses to control available space on the volume. Each volume is divided into blocks where the operating system can store information. In the bitmap file, there is one bit for each block on the volume. When a block is in use, its representative bit is set to a "1". If it is not in use, the bit is set to "Ø". The operating system checks the BITMAP.SYS file to find available space for creating and extending files, and to mark the bit for blocks that become available.

Bad Block File - All blocks not readable by the operating system are allocated to the file called BADBLK.SYS. The system task, BAD, run when preparing a volume for use, provides the information for this file. Allocating bad blocks to this file makes them unavailable, and decreases the frequency of I/O read and write errors.

Core Image File - This file, CORIMG.SYS, is set up to provide space for system checkpointing. Every volume has the capability of having system checkpoint space where checkpointed tasks can be written. To activate this area, a privileged user must use the SET DEVICE/CHECKPOINT_FILE command to allocate space to this file.

Directory Files - The two types of directory files that are present on a volume contain entries for UFDs or user files. Each entry in the file contains a file name, a file header number, and a file sequence number.

On a single user volume, (Figure 1Ø-4), only one type of directory file is needed, the Master File Directory (MFD).

On multiuser volumes, (Figure 1Ø-5) you must have one MFD and one UFD for each volume user. You use the CREATE/DIRECTORY command or the ACNTS program to establish the UFDs.

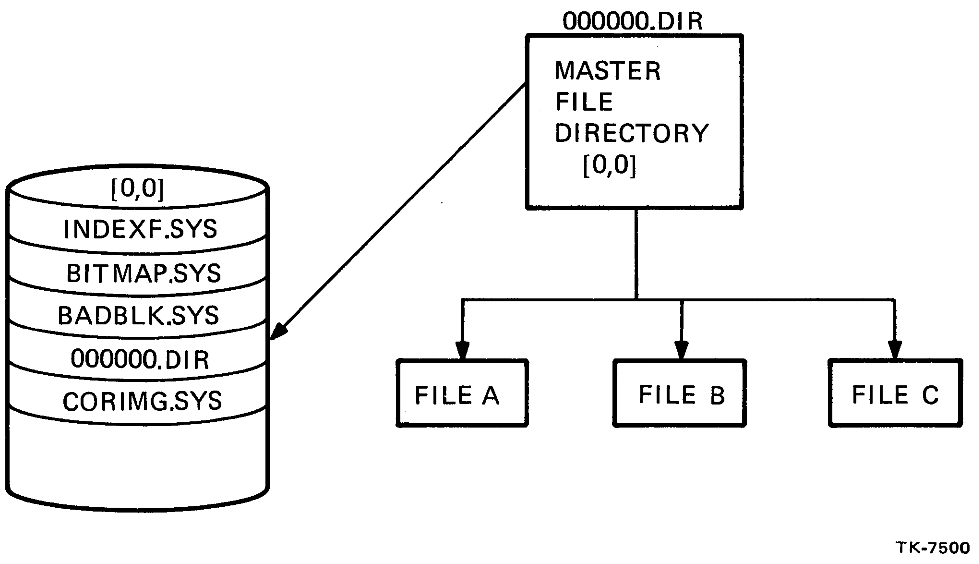


Figure 1Ø-4 Directory Structure for Single-User Volume

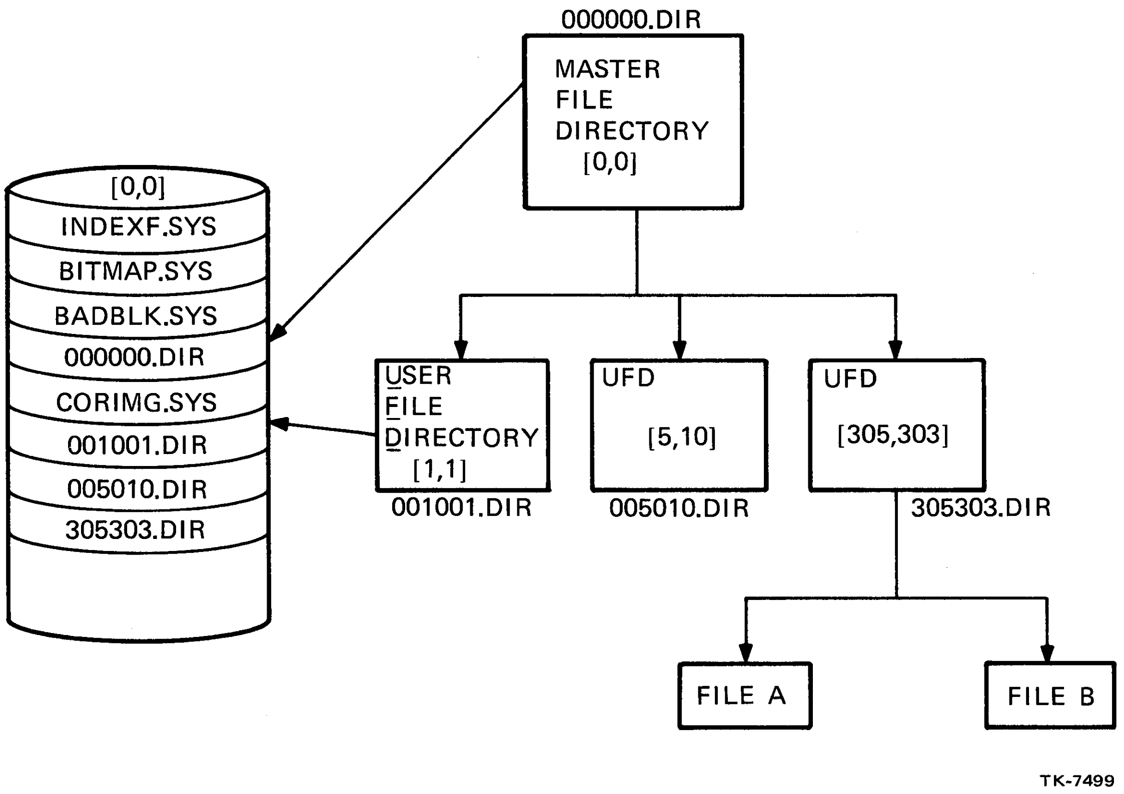


Figure 1Ø-5 Directory Structure for Multiuser Volume

ANSI Magnetic Tape Structure

The structure used for storing data on magnetic tape varies according to the application. Generally, on RSX-11M/M-PLUS systems, the ANSI Magnetic Tape structure is used to write data on the medium. This structure, shown in Figure 1Ø-6, is maintained by the Magnetic Tape Ancilliary Control Processor task (MTACP). Notice that it does not have the MFD, UFD structure. Files are stored sequentially on the tape, with the file header preceding the body of the file. The file header contains the following information:

- file name and extension
- file section number
- file sequence number
- creation date
- record format

Other Tape Formats

Utilities such as BRU, DSC, and FLX each have their own magnetic tape formats for efficient processing. A tape created by a particular utility can only be processed by that utility. It is important to remember how you create a magnetic tape so that information can be retrieved by using the proper utility.

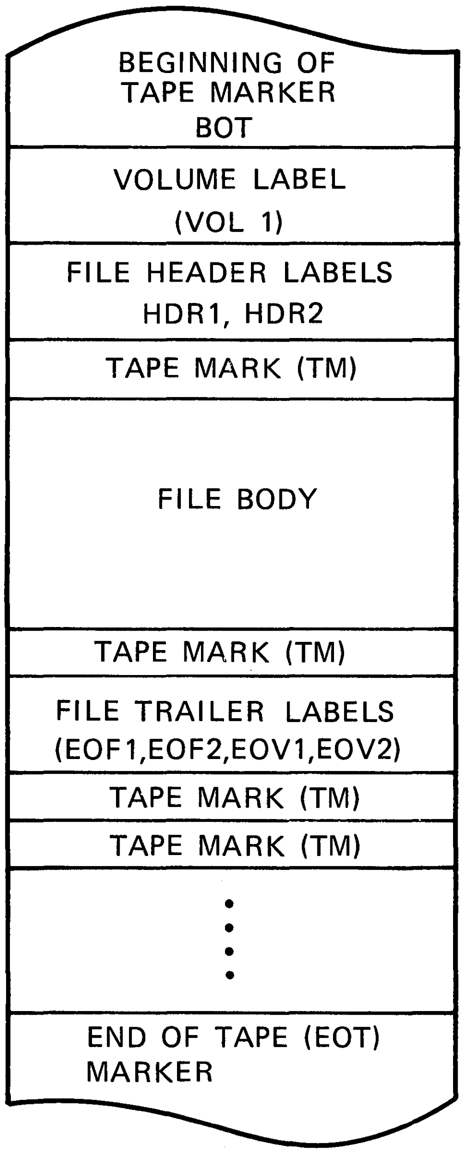


Figure 1Ø-6 ANSI Magnetic Tape Structure

Device and Volume Accessibility

Volumes and the devices on which they reside are available to the user depending upon characteristics established for them. Not all volumes or all devices are available for everyone's use. Accessibility is established in two ways: device ownership and volume accessibility.

Device Ownership

Devices on the system are declared to be public, private or unowned, as in Table 1Ø-1.

A public device is one established for use by all users. The SET DEVICE command, which is privileged, declares a device to be public. When a device is set to public, the volume mounted on the device is automatically mounted as public. This allows any user access to the data stored on that volume.

A private device is one for use by the person who allocates it.

Any device that has not been set public, private, or mounted is known as an unowned device.

Once the device attributes are established, a user can grant access to the volume mounted on the device, as shown in Table 1Ø-2. If the device is public, no further command is neccessary. However, to establish the volume as private or shareable, use the MOUNT command.

Table 1Ø-1 Device Ownership

Domain	Availability	Command
Public	Device accessible to all users; access automatically allowed	SET DEVICE DBØ: PUBLIC (privileged command)
Private	Device accessible only by person who allocated it	ALLOCATE DK1: PRVPCK
Unowned	Not allocated, mounted or set public	

Mounting a Volume

Before a disk or magnetic tape volume can be used on a device, the operating system must know that it is there. Just physically mounting a disk volume on the drive is not enough. You must logically inform the operating system by using the MOUNT command. This command establishes the software connection between the operating system and the volume.

The operating system also must know if the structure of the volume information is in FILES-11 format. If not, you mount it as FOREIGN.

The MOUNT command also establishes who may have volume access. Table 1Ø-2 indicates how to establish a volume as public, private or shareable.

Table 1Ø-2 Volume Accessibility

Domain	Accessibility	Command
Public	Volume accessible to all users	Automatically mounted public when SET DEVICE/PUBLIC command is issued
Private	Volume accessible only to user who allocated device	MOUNT DK1:
Shareable	Volume accessible to those issuing MOUNT command with correct volume label	MOUNT/SHAREABLE DK1: PACK

Preparing a Disk Volume for Use

Physical Formatting

Before you can use a disk pack on the system for the first time, it must be properly initialized. Initialization of the pack is a two step process. The first step is to physically format the disk. The system task, FMT, formats the disk into blocks, the smallest unit of writing area allocated to a file. The task, BAD, is then run to determine which blocks on the disk cannot be written to, or read properly. The following steps accomplish the physical formatting of the disk:

- Allocate disk drive using ALLOCATE/DEVICE command
- Physically load disk pack on drive
- Logically mount pack using MOUNT/FOREIGN command
- Run utility formatter task (FMT)

  ```
  MCR FMT
  ```

- Run bad block task (BAD)

  ```
  MCR BAD
  ```

FILES-11 Initialization

The second step in the initialization process is to set up the FILES-11 volume structure, which includes the creation of the five system files that must be present on the volume. Once the FILES-11 structure is there, you can create UFDs for those who require them. Example 1Ø-1 shows the complete process. Once the following steps are done, the disk volume is ready for use:

- Use INITIALIZE command to set up FILES-11 volume structure
- Logically DISMOUNT the volume and REMOUNT it with volume label name
- Create UFDs with CREATE/DIRECTORY command

Preparing a Magnetic Tape Volume for Use

To prepare a magnetic tape for use requires that an ANSI standard volume label be written on the tape, followed by a dummy file. Tape initialization requires no physical formatting. The following steps prepare a magnetic tape for use. An actual tape initialization process is shown in Example 1Ø-2.

- Allocate magnetic tape unit using ALLOCATE/DEVICE command
- Physically mount the tape on the drive
- Logically mount the tape using MOUNT/FOREIGN command
- Use INITIALIZE command to prepare magnetic tape in ANSI standard format
- Logically DISMOUNT volume and REMOUNT it with volume label name

```
>
>ALLOCATE DK2:
>MOUNT/FOREIGN DK2: USER
>MCR FMT
FMT>DK2:

** WARNING - Data will be lost on DK2: **

Continue? [Y OR N]: Y

Start formatting

Start verification

Operation complete

FMT>^Z
>MCR BAD
BAD>DK2:
BAD -- DK2: Total bad blocks= 0.
BAD>^Z
>INITIALIZE DK2: USER
>DISMOUNT DK2:
DMO -- TT56:   dismounted from DK2:    *** Final dismount initiated ***
>MOUNT DK2: USER
>DIR DK2:[0,0]


Directory DK2:[0,0]
15-SEP-81 11:33

INDEXF.SYS;1         155.        15-SEP-81 11:33
BITMAP.SYS;1         3.          15-SEP-81 11:33
BADBLK.SYS;1         1.          15-SEP-81 11:33
000000.DIR;1         1.          15-SEP-81 11:33
CORIMG.SYS;1         0.          15-SEP-81 11:33

Total of 160./160. blocks in 5. files
>
>
>CREATE/DIRECTORY DK2: [305,303]
>DIR DK2:[305,303]
DIR -- No such file(s)

>LO
DMO -- TT56:   dismounted from DK2:    *** Final dismount initiated ***
Have a Good Morning
15-SEP-81 11:36 TT56: logged off QUASAR
>
```

Example 1Ø-1 Preparing a Disk Volume

```
>
>
>ALLOCATE MM0:
>MOUNT/FOREIGN MM0: USER
>INITIALIZE MM0: USER
>DISMOUNT MM0:
DMO -- TT56:   dismounted from MM0:    *** Final dismount initiated ***
>
```

Example 1Ø-2 Preparing a Magnetic Tape Volume

Backing Up a Volume

For critical files that you cannot afford to lose, it makes sense to routinely backup the files on another volume. Backing up a volume means making a duplicate copy of it, and storing the copy in a protected place. Then, if needed, you can restore the files from the backup volume.

There are a number of ways to backup files, or a complete volume. The simplest way is to use the COPY command to copy files to another volume. This is sufficient when backing up a few files, but is quite cumbersome if backing up a complete volume. BRU and DSC are two utilities that exist to backup volumes. Each has its benefits.

Backup and Restore Utility (BRU)

The Backup and Restore Utility (BRU) is used for FILES-11 volumes. The output volume can be either a disk or magnetic tape. In addition to performing the backup, BRU will initialize the disk and run BAD to locate bad blocks. With this utility you can backup selected files, or a complete volume by file specification, date or time.

BRU writes data to the magnetic tape in its own tape format. Therefore, BRU must be used to restore data from a BRU backup tape.

Table 1Ø-3 lists the various backup command qualifiers.

Command Format

$$>\underbrace{\text{BACKUP}}_{1}/\underbrace{\text{SAVE\_SET}}_{2}\underbrace{\text{:VIPPROGS}}_{3}\ \underbrace{\text{DB1:**}}_{4}\ \underbrace{\text{MM1:}}_{5}$$

1 Command to invoke BRU to back up a disk area

2 Command qualifier

3 Qualifier value

4 Source device and file specification for files to be backed up

5 Destination device

Table 1Ø-3 Backup Command Qualifiers

User Wants To	Command Qualifier to Use
Backup files that are located on a mounted volume	/MOUNTED
Backup files created before a certain date	/CREATED/BEFORE:(1Ø-JUL-81 12:ØØ)
Backup files revised after a certain date	/MODIFIED/AFTER:(Ø1-JAN-81 16:ØØ)
Verify that backup copy is same as source copy	/VERIFY
List backup set names on tape volume	/LIST
List files in a backup set	/SAVE_SET:VIPPROG/DIRECTORY
Give backup volume a name	/SAVE_SET:VIPPROG
Give tape volume a name	/LABEL:TAPE:HOTPGM
Specify input disk volume name	/LABEL:INPUT:DISKIN
Specify output disk volume name	/LABEL:OUTPUT:DSKOUT
Create another backup set on same backup tape	/REWIND/APPEND

Disk Save and Compress Utility (DSC)

The Disk Save and Compress Utility (DSC) is used to duplicate FILES-11 volumes using either disk-to-disk or disk-to-tape file transfers. DSC backs up and restores entire volumes, and in the process compresses files into contiguous blocks, making more space available. DSC also has its own output magnetic tape format, so DSC must be used to restore a tape that it creates.

VERIFY (VFY)

The VERIFY utility is used to maintain the FILES-11 structure. It recaptures blocks marked as used but not belonging to a file. It searches for files in the index file that are not in any directory, as well as validates directories against the files they list. VERIFY requires that the volume to be verified be mounted as a FILES-11 device, and no other activity on the volume takes place. It is generally used after a system crash to ensure integrity of the structure, or if you suspect corruption on the volume.

LEARNING ACTIVITIES

1. READ the following sections in the RSX-11M/M-PLUS Command Language Manual:

   - 5.1.3, Public, Shared, Private and Unowned Devices and Mounted Volumes
   - 5.1.4, How to Prepare a Scratch Disk for Use
   - 5.1.5, How to Prepare a Scratch ANSI Magnetic Tape for Use
   - 5.4, Allocate
   - 5.5, Deallocate
   - 5.6, Mount
   - 5.7, Dismount
   - 5.8, Initialize
   - 5.9, Backup
   - 5.1Ø.2, Set Device

2. READ the following in the RSX-11M/M-PLUS Utilities Manual:

   - Chapter 7, Backup and Restore Utility (BRU)
   - Chapter 8, Disk Save and Compress (DSC)
   - Chapter 5, Disk Volume Formatter (FMT)
   - Chapter 6, Bad Block Locator Utility (BAD)

3. DO Written Exercises 1 through 11 for this module.

FILE MAINTENANCE

Transferring Files Between Computer Systems

File Transfer Program

The File Transfer Program (FLX) transfers files from one volume to another. In addition, when the volume structures differ, FLX will convert the file to the output volume format. The following file conversions and transfers can be accomplished:

- DOS-11 to FILES-11 volumes
- FILES-11 to DOS-11 volumes
- DOS-11 to DOS-11 volumes
- FILES-11 to FILES-11 volumes
- FILES-11 to RT-11 volumes
- RT-11 to RT-11 volumes
- RT-11 to FILES-11 volumes

Figure 1Ø-7 shows the different file structures used by three operating systems that run on PDP-11 hardware. RT-11 uses a contiguous file storage structure. All blocks of a file are stored together on the disk.

RSTS (DOS-11) uses a linked list structure. Each block is stored individually where space permits. A pointer to the next block in the file is stored with each block.

On RSX-11M/M-PLUS, FILES-11 breaks up the file into segments and stores the segments on the volume. A segment consists of one or more blocks. In the header of the file, a record is kept of the number of segments, the number of blocks in each segment, and the address location of the segment on the volume.

In addition to this difference, there is a difference in the way text files, object files, and executable files are formatted on each operating system. For text files on RSX, the record length is determined by a character count, which is stored in the first position of a string of characters that constitute the record. On RSTS and RT, the record length is determined by the <CR> <LF> that is stored at the end of a string of characters.

Each operating system formats and stores files differently. Therefore, the DCL COPY command which does a straight copy, cannot be used to transfer files from a volume initialized on one system, to one initialized on another system.

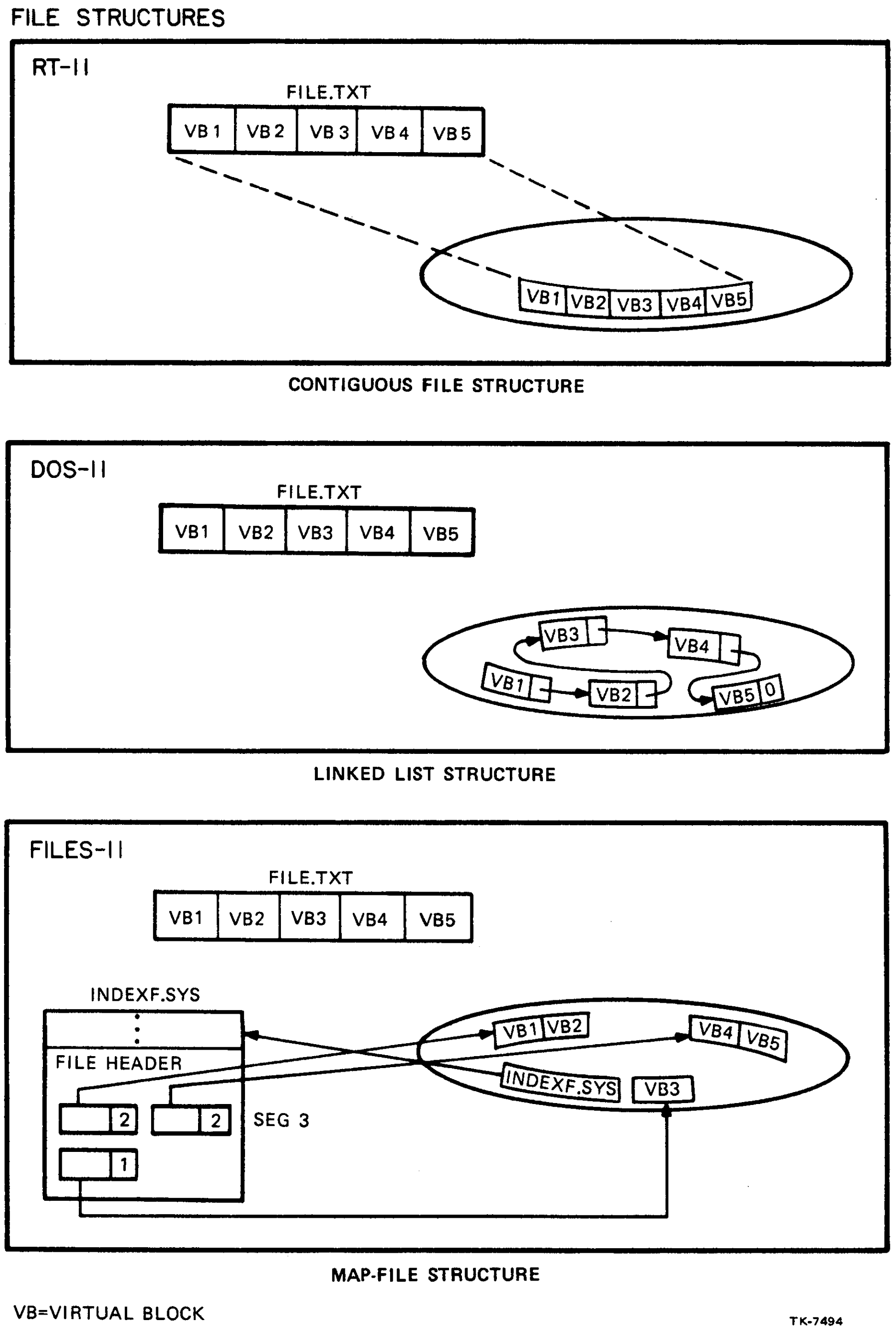


Figure 1Ø-7 PDP-11 File Structures

Transfer Mode Switches

There are three transfer modes (Table 10-5) that you may use:

- Formatted ASCII
- Formatted Binary
- Image Mode

Formated ASCII is used to embed the <CR> <LF> in text files when you are generating RSTS and RT output. Formatted binary is used to correctly format .OBJ, .STB, .BIN and .LDA files for RSTS and RT output. Image mode is a straight copy with no changes for executable files and libraries.

Command Format

$$>\overbrace{\text{FLX}}^{\text{❶}}\text{<RET>}$$

$$\text{FLX>}\underbrace{\text{DK1:}}_{\text{❷}}\underbrace{\text{/RT}}_{\text{❸}}\underbrace{=}_{\text{❹}}\underbrace{\text{DK0:SYS1.MAC}}_{\text{❺}}\underbrace{\text{/RS}}_{\text{❻}}$$

❶ Command to invoke the file transfer program

❷ Output device

❸ A volume format, transfer mode or control switch

❹ Delimiter

❺ Input file and device specification

❻ A volume format, transfer mode, or control switch

Table 1Ø-4 FLX Format Switches

	FLX Volume Format Switches
Switch	Description
/DO	DOS-11 formated volume input volume default
/RS	RSX-11 FILES-11 formatted volume output volume default
/RT	RT-11 formatted volume
	Setting Default Switches
Switch	Input/Output Default File Formats
/RS	Input file = FILES-11 Output file = DOS-11
/DO	Input file = DOS-11 Output file = FILES-11

Table 1Ø-5 Default Transfer Modes

Default Mode	Switch	File Type
Image	/IM:n	.TSK .MLB .SML .OLB .SYS .ULB
Formatted Binary	/FB:n	.OBJ .STB .BIN .LDA
Formatted ASCII	/FA:n	All others

Controlling a Print Queue

When a device is frequently used, like the line printer, a method is needed to manage that device efficiently to ensure that it is readily available for use. A system task, the Queue Manager, is responsible for managing such devices. Line printers, plotters, and hard-copy terminals being used as line printers are the kinds of devices the Queue Manager oversees.

Once a device is under control of the Queue Manager, it can no longer be used in the usual manner. For example, when the line printer is under Queue Manager control, files cannot be sent directly to the line printer. Queue Manager commands must be used to process files.

Figure 10-8 shows how the Queue Manager handles a line printer as a spooled device. The PRINT command indicates a file to be printed on the line printer. This command is passed to the Queue Manager for processing. A file called LB:[1,7]QUEUE.SYS contains an entry to indicate that a file is ready for printing. This part of the process is called spooling. The Queue Manager frequently checks the queue to see if there are any waiting jobs. If it finds one for the line printer, it checks if the line printer is available. If no job is printing, the Queue Manager starts printing the file. This part of the process is called despooling.

To enter a job into the print queue, you use the PRINT command. Examples are shown in Table 10-6. Once the job is in the queue, to alter job characteristics or to cancel a job, you must use the commands in Table 10-7 or 10-8.

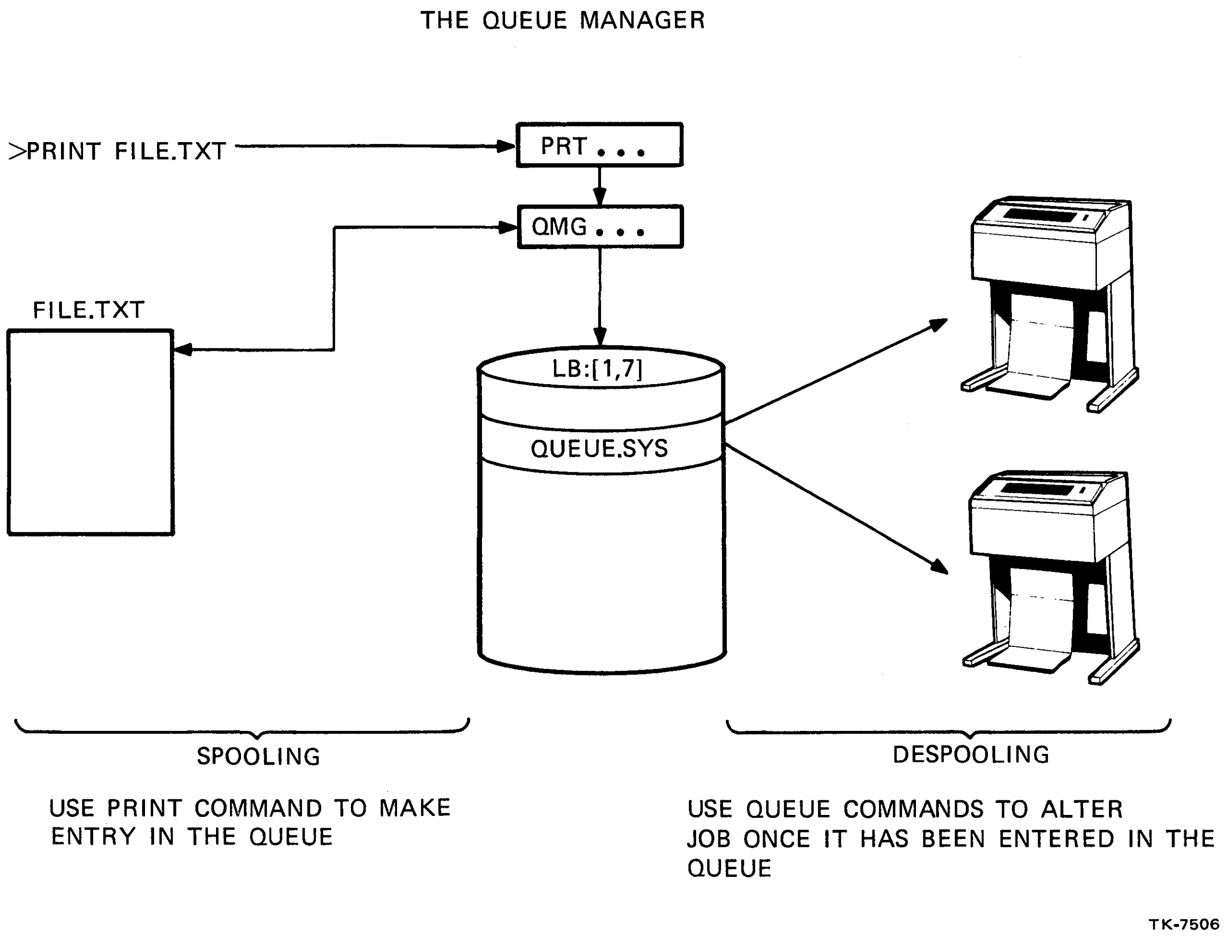

THE QUEUE MANAGER
>PRINT FILE.TXT
PRT . . .
QMG . . .
FILE.TXT
LB:[1,7]
QUEUE.SYS
SPOOLING
USE PRINT COMMAND TO MAKE ENTRY IN THE QUEUE
DESPOOLING
USE QUEUE COMMANDS TO ALTER JOB ONCE IT HAS BEEN ENTERED IN THE QUEUE
TK-7506

Figure 10-8 The Queue Manager

Table 1Ø-6 Print Command Qualifiers

User Wants To	Use Following Commands
Specify number of copies to print	PRINT/COPIES:3 PROG.LST
Specify forms job is to be printed on	PRINT/FORMS:1 PAYROL.TXT
Delete file after printing	PRINT/DELETE PROG.LST, PROG.MAP
Set print job priority in its queue	PRINT/PRIORITY:1ØØ MEMO.TXT
Hold job in queue until released by queue command	PRINT/HOLD LARGE.TXT
Print job after a stated time	PRINT/AFTER:18:3Ø LARGER.TXT
Control number of text lines per page	PRINT/LENGTH:3Ø SHORTFORM.TXT

Table 1Ø-7 DCL Commands to Alter Print Queue

User Wants To	Use Following Commands
See if print job is still in queue	SHOW QUEUE
Delete print job from print queue	DELETE/JOB PRINT JOBNAME
Release job held in print queue	SET QUEUE/JOB/RELEASE PRINT JOBNAME -or- RELEASE/JOB PRINT JOBNAME
Change job priority in print queue	SET QUEUE/JOB/PRIORITY:1ØØ PRINT JOBNAME

Table 1Ø-8 Equivalent MCR Commands to Alter Print Queue

User Wants To	Use Following Commands
See if print job is still in queue	>QUE /LIST
Delete print job from print queue	>QUEUE PRINT:PROSE/DEL
Release job held in print queue	>QUEUE PRINT:PROSE/DEL
Change priority of job in print queue	>QUEUE PRINT:PROSE/MOD/PRIO:1ØØ

Looking at the Contents of a File

File Dump Utility (DMP)

In the course of program development, it is sometimes necessary to inspect the contents of a file. You may want to check the file format to ensure that it is correct. And if it is not correct, you would want to determine the spot where it was not formatted properly. The File Dump Utility (DMP) helps in uncovering such problems.

DMP will dump a file in any one of the formats listed in Table 10-9. You must decide which format is suitable for your use and for the file.

Figure 10-9 depicts how the DMP program can interpret sixteen bits of data. If a word contains the value as shown in the figure, the string of 0s and 1s can be interpreted in many different ways. For example, interpreting the data as an octal word gives it the value of 044124. Interpreting the data as octal bytes, the left byte value is 110, and the right byte value is 124. Interpreting the data as two ASCII characters yields an H in the left byte, and a T in the right byte. Each of these interpretations could be correct depending upon the content and purpose of the file.

The DMP utility has two modes of operation: file mode and device mode. File mode requires that the volume on which the file is located be mounted as a FILES-11 volume. This mode dumps the virtual blocks of a file, and is used more frequently. Device mode is for dumping logical blocks on a volume. For a more detailed description of both file and device mode, read the appropriate sections in Chapter 11 of the RSX-11M/M-PLUS Utilities Manual.

To produce the results shown in Example 10-3, a directory file 203054.DIR was copied from UFD [0,0] into UFD [305,303]. The DMP utility was then used on the file [305,303]203054.DIR to dump the header of that file. The three sections of the file header (header, identification, and map areas) provide useful information for the user, and especially the operating system. The header area provides the file sequence number (1335,3) which is used to locate the file on the volume, the file owner ([305,303]), the file protection word ([REWD,RWED, RWED,R]), and other information.

The identification area supplies the file name, type, revision number, the date and time of creation, and the date and time of last revision. The DCL command, DIRECTORY, uses this information from the header of the UFD file to provide the listing you see at your terminal when you issue the command.

Example 1Ø-4 shows the results of dumping the same directory file in Octal Word format. The numbers in the first column represent the virtual disk addresses for the data that follows. Across the row is the octal representation of each word of data in the file.

Example 1Ø-5 shows the same directory file dumped in Radix-5Ø format; information in the file header is stored in Radix-5Ø format. To use the dump program efficiently, the user must know how the data is stored in the file.

Command Format

$$>\overbrace{\text{RUN \$DMP}}^{❶}$$

$$\text{DMP}>\underbrace{\text{TI:}}_{❷}\underbrace{=}_{❸}\underbrace{\text{305303.DIR}}_{❹}\underbrace{/\text{R5}}_{❺}$$

❶ Command to invoke the Dump program

❷ Output file specification

❸ Delimiter

❹ Input file specification

❺ A dump switch to specify a mode

Table 10-9 DMP Switch Format

DMP Switch	Format
Default Mode	Octal word format
/AS	ASCII mode
/BY	Octal byte format
/DC	Decimal word format
/HD	Dumps file header
/HX	Hexadecimal byte format
/LW	Hexadecimal longword format
/R5	Radix-50 format
/WD	Hexadecimal word format

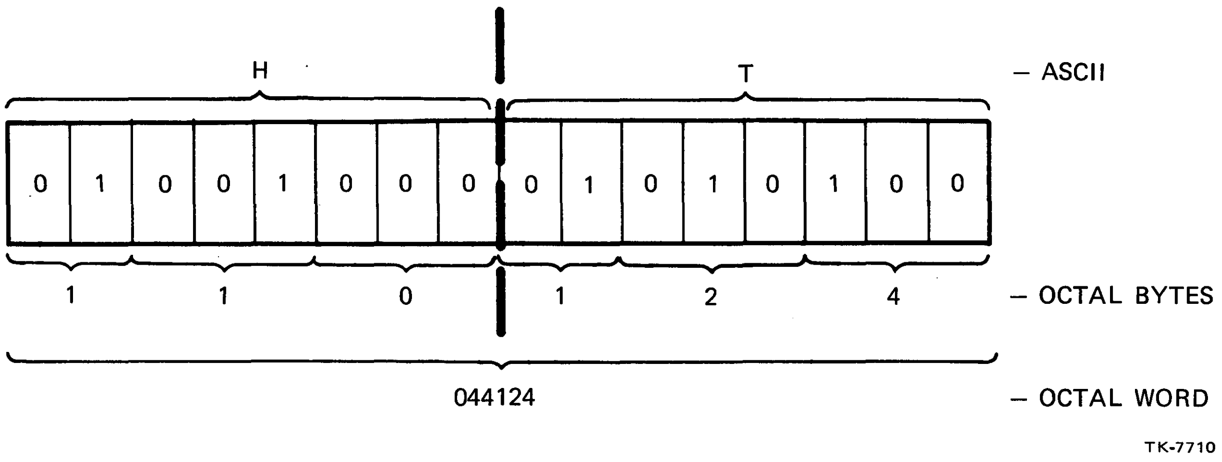


Figure 10-9 How the DMP Program Interprets 16 Bits

```
DUMP OF DB2:[305,303]203054.DIR;1 - FILE ID 1335,3,0
                                        FILE HEADER

HEADER AREA
        H.IDOF          027
        H.MPOF          056
        H.FNUM,
        H.FSEQ          (1335,3)
        H.FLEV          401
        H.FOWN          [305,303]
        H.FPRO          [RWED,RWED,RWED,R]
        H.UCHA          200 =  UC.CON
        H.SCHA          000 =
        H.UFAT
                F.RTYP  001 = R.FIX
                F.RATT  000 =
                F.RSIZ  20 = 16.
                F.HIBK  H:0 L:000002 = 2.
                F.EFBK  H:0 L:000003 = 3.
                F.FFBY  0 = 0.
                (REST)
                000000 000000 000000 000000 000000 000000 000000 000000
                000000
IDENTIFICATION AREA
        I.FNAM,
        I.FTYP,
        I.FVER          203054   .DIR;1
        I.RVNO          1
        I.RVDT          16-SEP-81
        I.RVTI          11:07:52
        I.CRDT          16-SEP-81
        I.CRTI          11:07:52
        I.EXDT            -   -
MAP AREA
        M.ESQN          000
        M.ERVN          000
        M.EFNU,
        M.EFSQ          (0,0)
        M.CTSZ          001
        M.LBSZ          003
        M.USE           002 = 2.
        M.MAX           314 = 204.
        M.RTRV
        SIZE    LBN
        2.      H:000 L:024104 = 10308.
CHECKSUM
        H.CKSM          061747
```

Example 10-3 Directory File Header Dump

```
DUMP OF DB2:[305,303]203054.DIR;1 - FILE ID 1335,3,0
                    VIRTUAL BLOCK 0,000001 - SIZE 512. BYTES

000000      000152 000076 000000 071620 143245 000000 074742 000001
000020      001172 000025 000000 004640 000000 000000 100003 000002
000040      001201 000053 000000 004640 000000 000000 074742 000002
000060      003047 000111 000000 004640 000000 000000 014474 000001
000100      003060 000020 000000 055210 000000 000000 100003 000002
000120      003061 000011 000000 055210 000000 000000 074742 000002
000140      001533 000245 000000 054204 000000 000000 100003 000003
000160      003123 000054 000000 055210 076452 000000 100003 000002
000200      003124 000017 000000 055210 076452 000000 074742 000002
000220      003125 000100 000000 015370 000000 000000 100003 000002
000240      003126 000010 000000 015370 000000 000000 074742 000002
000260      003127 000046 000000 015370 076452 000000 100003 000002
000300      003131 000012 000000 015370 076452 000000 074742 000002
000320      003133 000037 000000 015370 000000 000000 014474 000002
000340      003134 000014 000000 062204 000000 000000 100003 000002
000360      003136 000061 000000 062204 000000 000000 074742 000002
000400      003143 000155 000000 062204 000000 000000 014474 000002
000420      003145 000012 000000 021300 000000 000000 100003 000002
000440      003154 000150 000000 055251 054374 000000 100003 000002
000460      003157 000011 000000 055254 000000 000000 100003 000002
000500      003277 000044 000000 055400 000000 000000 100003 000002
000520      003353 000002 000000 054010 000000 000000 100003 000002
000540      003355 000002 000000 054353 017500 000000 100003 000002
000560      003371 000002 000000 046166 000000 000000 100003 000002
000600      003401 000002 000000 046547 000000 000000 100003 000002
000620      003411 000002 000000 051272 000000 000000 100003 000002
000640      003441 000023 000000 011665 000000 000000 100003 000002
000660      003453 000002 000000 010155 000000 000000 100003 000002
000700      003457 000002 000000 077353 000000 000000 100003 000002
000720      003465 000002 000000 047006 000000 000000 100003 000002
000740      003473 000002 000000 024263 000000 000000 100003 000002
000760      003475 000002 000000 024261 000000 000000 100003 000002
```

Example 1Ø-4 Directory File Dumped in Octal Word Mode

```
DUMP OF DB2:[305,303]203054.DIR;1 - FILE ID 1335,3,0
                    VIRTUAL BLOCK 0,000001 - SIZE 512. BYTES

000000      BZ   AV      RSX 11M     STB    A
000020      04    U      AUX         TSK    B
000040      PA   AC      AUX         STB    B
000060      90   A3      AUX         DAT    A
000100      9X    P      NSP         TSK    B
000120      9Y    I      NSP         STB    B
000140      US   DE      NFT         TSK    C
000160     A S   AD      NSP TAB     TSK    B
000200     A T    O      NSP TAB     STB    B
000220     A U   AX      DLX         TSK    B
000240     A V    H      DLX         STB    B
000260     A W    B      DLX TAB     TSK    B
000300     A Y    J      DLX TAB     STB    B
000320     A $    1      DLX         DAT    B
000340     A .    L      PCL         TSK    B
000360     A 0   AI      PCL         STB    B
000400     A 5   B/      PCL         DAT    B
000420     A 7    J      EVP         TSK    B
000440     AAD   BX      NTI NIT     TSK    B
000460     AAG    I      NTL         TSK    B
000500     ACG    6      NVP         TSK    B
000520     ADK    B      NCP         TSK    B
000540     ADM    B      NIC E       TSK    B
000560     ADY    B      LIN         TSK    B
000600     AD3    B      LOO         TSK    B
000620     AEA    B      MIR         TSK    B
000640     AEY    S      CFE         TSK    B
000660     AE5    B      BYE         TSK    B
000700     AE9    B      TLK         TSK    B
000720     AFE    B      LSN         TSK    B
000740     AFK    B      FTS         TSK    B
000760     AFM    B      FTQ         TSK    B
```

Example 10-5 Directory File Dumped in Radix-50 Mode

LEARNING ACTIVITIES

1. READ the following in the RSX-11M/M-PLUS Utilities Manual:

   - Chapter 11, File Dump Utility (DMP)
   - Chapter 4, File Transfer Program (FLX)

2. READ the following sections in the RSX-11M/M-PLUS Command Language Manual:

   - 2.5, Show Queue
   - 4.4.2, Print

3. DO Written Exercises 12 through 16 for this module.

PROGRAM MAINTENANCE

Comparing the Contents of Two Files

File Compare Utility (CMP)

The File Compare Utility (CMP) compares the contents of two ASCII files to determine their differences. CMP reads the two input files, comparing them line-by-line, and generates a listing showing the difference.

The DCL command to invoke the utility is DIFFERENCES which is shown in the command format below. The /OUTPUT qualifier indicates that the differences should be placed in a file with the name DIF.DIF. If you want the output to come to the terminal, omit the /OUTPUT qualifier. The next two file specifications indicate which files are to be compared.

Example 1Ø-6 shows the contents of two ASCII files. Using these two files as input to the CMP produces the results found in Example 1Ø-7.

Example 1Ø-7 shows the standard output format, indicating the lines that are different between the files.

Example 1Ø-8 is an example of the second output format available. On lines 6, 8, and 13-17, an exclamation point appears to the right of the line number. This character is used to represent a change bar, and indicates which lines in the second input file differ from the first input file.

Command Format

```
>DIFFERENCES/OUTPUT: DIF.DIF SOURCE.MAC;1 SOURCE.MAC;2
```

```
>
>
>
>TYPE COMPARE.TXT
THIS IS A TEST TO SHOW THE RESULTS OF THE COMPARE
FILE UTILITY (CMP).  THIS UTILITY COMPARES TWO
ASCII FILES.  THE FILES ARE COMPARED LINE BY LINE TO
DETERMINE WHETHER PARALLEL RECORDS ARE IDENTICAL.
USING CMP, YOU CAN PERFORM THE FOLLOWING FILE-
COMPARE FUNCTIONS:

        GENERATE A LISTING SHOWING THE DIFFERENCES
        BETWEEN THE TWO FILES.  EACH DIFFERENCE IS
        LISTED AS A PAIR; FIRST, THE LINES FROM
        THE FIRST FILE THAT ARE BEING COMPARED TO
        LINES IN THE SECOND FILE, THEN THE LINES
        FROM THE SECOND FILE.

        GENERATE A LISTING IN THE FORM OF ONE LIST,
        WITH DIFFERENCES MARKED BY CHANGE BARS.

        GENERATE OUTPUT SUITABLE FOR INPUT TO THE SLP
        UTILITY.  THIS OUTPUT CONTAINS THE SLP COMMANDS
        AND INPUT REQUIRED TO MAKE THE FIRST INPUT
        FILE IDENTICAL TO THE SECOND INPUT FILE.

>TYPE CMP.TXT
THIS IS A TEST TO SHOW THE RESULTS OF THE COMPARE
FILE UTILITY (CMP).  THIS UTILITY COMPARES TWO
ASCII FILES.  THE FILES ARE COMPARED LINE BY LINE TO
DETERMINE WHETHER PARALLEL RECORDS ARE IDENTICAL.
USING CMP, YOU CAN PERFORM THE FOLLOWING FILE-
COMPARE FUNCTIONS:

        GENERATE A LISTING SHOWING THE DIFFERENCES
        BETWEEN THE TWO FILES.  EACH DIFFERENCE IS
        LISTED AS A PAIR; FIRST, THE LINES FROM
        THE FIRST FILE THAT ARE BEING COMPARED TO
        LINES IN THE SECOND FILE, THEN THE LINES
        FROM THE SECOND FILE.

        GENERATE A LISTING IN THE FORM OF ONE LIST,
        WITH DIFFERENCES MARKED BY CHANGE BARS.

        GENERATE OUTPUT SUITABLE FOR INPUT TO THE SLP
        UTILITY.  THIS OUTPUT CONTAINS THE SLP COMMANDS
        AND INPUT REQUIRED TO MAKE THE FIRST INPUT
        FILE IDENTICAL TO THE SECOND INPUT FILE.

CMP PROVIDES SWITCHES THAT ALLOW YOU TO CONTROL COMPARE
PROCESSING. USING THESE SWITCHES, YOU CAN CONTROL
COMPARISON OF BLANKS, TABS, FORM-FEEDS, AND COMMENTS.
YOU CAN ALSO CONTROL LINE NUMBERING AND THE NUMBER OF
LINES REQUIRED FOR CMP TO CONSIDER THAT A MATCH IS MADE
BETWEEN LINES IN THE TWO FILES.
>
>
```

Example 1Ø-6 Two Similar ASCII Files

```
>DIFFERENCE/OUTPUT:DIF.TXT COMPARE.TXT CMP.TXT
>TYPE DIF.TXT
*************************************************
  1)  DB0:[305,303]COMPARE.TXT;1
***************
  2)  DB0:[305,303]CMP.TXT;2
   23    CMP PROVIDES SWITCHES THAT ALLOW YOU TO CONTROL COMPARE
   24    PROCESSING. USING THESE SWITCHES, YOU CAN CONTROL
   25    COMPARISON OF BLANKS, TABS, FORM-FEEDS, AND COMMENTS.
   26    YOU CAN ALSO CONTROL LINE NUMBERING AND THE NUMBER OF
   27    LINES REQUIRED FOR CMP TO CONSIDER THAT A MATCH IS MADE
   28    BETWEEN LINES IN THE TWO FILES.

    1 differences found
DIF.TXT/BL/FF=COMPARE.TXT,CMP.TXT
>DIFFERENCE/OUTPUT:DIF.TXT/CHANGE_BAR COMPARE.TXT CMP.TXT
>TYPE DIF.TXT
*************************************************
  1)  DB0:[305,303]COMPARE.TXT;1
***************
  2)  DB0:[305,303]CMP.TXT;2
   23    CMP PROVIDES SWITCHES THAT ALLOW YOU TO CONTROL COMPARE
   24    PROCESSING. USING THESE SWITCHES, YOU CAN CONTROL
   25    COMPARISON OF BLANKS, TABS, FORM-FEEDS, AND COMMENTS.
   26    YOU CAN ALSO CONTROL LINE NUMBERING AND THE NUMBER OF
   27    LINES REQUIRED FOR CMP TO CONSIDER THAT A MATCH IS MADE
   28    BETWEEN LINES IN THE TWO FILES.

    1 differences found
DIF.TXT/BL/FF/VB=COMPARE.TXT,CMP.TXT
>
```

Example 10-7 Compare Program Output Between Two ASCII Files (Standard Format)

```
    1                         .TITLE FIG1
    2             TEXT:       .ASCII  /TRY THIS. DID IT WORK?/
    3             R0=%0
    4             R1=%1
    5             R2=%2
    6    !        DOT='.
    8    !        BUFF:       .BLKB    ^D80
    9                         .EVEN
   11             ST:         MOV      #TEXT,R0
   12                         MOV      #BUFF,R1
   13    !                    MOV #DOT,R2
   14    !        LOOP:       MOVB     (R0)+,(R1)
   15    !                    CMPB     (R1)+,R2
   16    !                    BNE      LOOP
   17    !                    .END

    2 DIFFERENCES FOUND
PROG.DIF/CB=PROG.MAC;2,PROG.MAC;3
```

Example 10-8 Change Bar Format

LEARNING ACTIVITIES

1. READ Chapter 12, File Compare Utility (CMP), in the RSX-11M/M-PLUS Utilities Manual.
2. READ Section 4.4.5, Differences, in the RSX-11M/M-PLUS Command Language Manual.
3. DO Written Exercise 17 for this module.

APPENDIX A BATCH JOBS (M-PLUS ONLY)

BATCH JOB FILE

In addition to the Indirect Command File Processer, the RSX-11M-PLUS Operating System has another means of automating a process. The Batch File Processor is like the Indirect Command File Processor in that it is able to read and then process commands from a file. Because the job is run from a virtual terminal, your terminal is free to do other tasks.

BATCH Command Line

The following command line format shows how to write a batch command. The dollar sign in the first position is necessary to indicate that the line is to be interpreted as a Batch command. If the dollar sign is not present, the line is considered to be data. The first command in the Batch file is JOB (mandatory) and marks the beginning of the job. The last command is EOJ (mandatory) which indicates the end of the job. What comes in between is dependent upon the process you are automating, and will include some or possibly all of the commands listed in Table A-1.

Command Format

$$>\underbrace{\$}_{❶}[\underbrace{\text{START:}}_{❷}][\underbrace{\text{PRINT DELTAX.TXT}}_{❸}]\ \underbrace{-}_{❹}$$

❶ Batch processor symbol indicating that the line following is to be interpreted as a DCL command.

❷ Optional label (1 through 6 characters terminated by a colon) used to mark a position in the command file to which a GOTO statement may refer.

❸ Any DCL or MCR command except LOGIN, LOGOUT, HELLO and BYE.

❹ Character (hyphen) to indicate the command will be continued on the next line.

Table A-1 Batch Commands

User Wants To	Batch Command To Use
Indicate beginning of user batch job (mandatory)	$ JOB
Indicate end of a user batch job (mandatory)	$ EOJ
Include comments in batch log	$!
Invoke an indirect command file from a batch job	$ @
Allocate a drive and logically mount a volume	$ MOUNT
Dismount a volume and deallocate a drive	$ DISMOUNT
Inhibit printing a data block in the log file	$ DATA/NOCOPY
Include data with a dollar sign ($) in the first position of a line within a data block	$ DATA/DOLLARS
Indicate end of data block	$ EOD
Stop batch job	$ STOP
Alter flow of execution of batch job	$ GOTO
Check for return of a given status code of operations throughout command file	$ ON
Check for a given status code following execution of given command in batch job	$ IF

PROCESSING THE BATCH FILE

After you define the process and translate it into a corresponding Batch file, the next step is to submit the batch job for processing. The following command format shows how to do this.

The SUBMIT command places your job in the Batch queue, which the Queue Manager checks periodically for jobs to run.

Depending upon the SUBMIT command qualifiers you use, (Table A-3) and the number of jobs in the Batch queue, your job may either start up immediately, or at a later time. The /AFTER qualifier allows you to specify the time at which you wish the Queue Manager to consider your job for processing.

Once a job is in the queue, you must use the SET QUEUE, RELEASE/JOB, DELETE/JOB or SHOW QUEUE to display, delete or change the characteristics of the job (Table A-4).

The Batch Processor automatically provides a record of the activity generated when processing a job, which is called the log file. Unless you specify otherwise, this log file prints on your system's line printer after processing is complete, and then is deleted.

Example A-1 shows a Batch file and the log file the Batch processor creates when processing the file.

Command Format

$$>\underbrace{\text{SUBMIT}}_{❶}\underbrace{\text{/JOB:BATRUN}}_{❷}\ \underbrace{\text{BATCH.CMD}}_{❸}\underbrace{,}_{❹}\underbrace{\text{PROCESS.CM}}_{❸}$$

❶ Command to queue a job for processing by a batch processor

❷ Command qualifier

❸ File specification of file from which batch commands will be read. Default file type is .CMD.

❹ File specification delimiter

Table A-2 Exit Status Code

Status Code	Meaning
SUCCESS	Results should be as expected
WARNING	Task succeeded but irregularities possible
ERROR	Results unlikely to be as expected
SEVEREERROR	One or more fatal errors or abort encountered

Table A-3 Submit Command Qualifiers

User Wants To	Submit Command Qualifier To Use
Specify queue in which job is to be placed	/QUEUE:BATCH
Print batch log on printer	/[NO]PRINT
Specify time and day to unblock job in queue	/AFTER:(1Ø-MAY-82 17:ØØ:ØØ)
Hold job in queue	/HOLD
Specify job name for the batch job	/NAME:HOTONE
Indicate if a job should be restarted from beginning, if interrupted	/[NO]RESTART
Set queue priority of job	/PRIORITY:1ØØ
Specify deletion of command file after execution	/DELETE

Table A-4 Changing the Batch Queue

User Wants To	DCL Command To Use
Display information regarding jobs in batch queue	SHOW QUEUE BATCH
Display information regarding a particular job in batch queue	SHOW QUEUE BATCH/NAME:jobnam
Delete a job from queue	DELETE/JOB BATCH jobnam
Release a batch job held in a queue	RELEASE/JOB BATCH jobnam
Change priority of a job in batch queue	SET QUEUE BATCH jobnam/PRI:1

Examples A-1 and A-2 show how Batch commands can be organized to automate a process. Example A-1 allocates a disk drive, mounts the volume and performs a full directory on the volume, sending the output to the line printer. Then the volume is dismounted, and the drive deallocated.

Example A-2 runs a task called WONDER. If a severe error occurs, a message is sent to terminal number 1Ø indicating that an error has occurred, and the run will end. If an error occurs, a message is sent to terminal number 1Ø, and the task OLD is run. If a warning occurs, a message is sent to the terminal and the command file terminates.

```
$JOB PRTJOB [305,303]
$!THIS IS A BATCH CONTROL FILE THAT WILL PRINT ALL MY
$!MY MACRO SOURCE FILES.  WHEN I SUBMIT THIS FILE FOR PROCESSING
$!I WILL USE THE /AFTER QUALIFIER OF THE SUBMIT COMMAND TO
$!DELAY THE EXECUTION OF THE BATCH FILE UNTIL AFTER NORMAL
$!WORKING HOURS SO THAT I WON'T TIE UP THE PRINTER.
$PRINT *.MAC
$EOJ
```

```
QMG Batch Job - PRTJOB          BPR  V02       24-JAN-82  11:17       Page 1
Processor BAP0


11:17:08        $JOB PRTJOB [305,303]

        ===================================
        User Job - PRTJOB       Terminal VT2:
                UIC = [305,303]
        ===================================

        TERM
          .     RSX-11M-PLUS V02 BL8     [2,54] System     KERMIT
          .
          .          **********************************************************
          .          *                                                        *
          .          *               Welcome to RSX-11M-PLUS batch            *
          .          *                  Version 2  Base level 8               *
          .          *                                                        *
          .          *               This is file LB:[1,2]BATCH.TXT           *
          .          *                                                        *
          .          **********************************************************
          .
11:17:10        $!THIS IS A BATCH CONTROL FILE THAT WILL PRINT ALL MY
11:17:10        $!MY MACRO SOURCE FILES.  WHEN I SUBMIT THIS FILE FOR PROCESSING
11:17:10        $!I WILL USE THE /AFTER QUALIFIER OF THE SUBMIT COMMAND TO
11:17:10        $!DELAY THE EXECUTION OF THE BATCH FILE UNTIL AFTER NORMAL
11:17:10        $!WORKING HOURS SO THAT I WON'T TIE UP THE PRINTER.
11:17:10        $PRINT *.MAC
        TERM    PRI - Job 666, name "PRTJOB   ", submitted to queue "PRINT "
11:17:18        $EOJ
        TERM    Connect time: 1      minutes
          .     CPU time used: 2     seconds
          .     Task total:    6
```

Example A-1 Batch Control File to Print MACRO Source Files

```
$JOB
$!
$! THIS BATCH JOB WILL GET A FULL DIRECTORY LISTING
$! OF AN RL01 DISK THAT IS ALREADY LOADED AND SPUN UP.
$!
$SET TERMINAL MCR
$ALL DL1:
$MOU DL1:IMREADY
$PIP LP0:=DL1:[*,*]/FU
$DMO DL1:
$DEA DL1:
$EOJ
```

```
$JOB AONE[303,14]
$!
$! IF COMMAND EXAMPLE
$!
$RUN WONDER.TSK
$IF SEVEREERROR THEN GOTO BOMB
$IF ERR THEN GOTO ALAS
$IF WARNING THEN GOTO OKAY
$GOTO REST      !SUCCESS ASSUMED
$BOMB:
$BRO TT10:  SEVERE ERROR.  RUN ENDS
$STOP
$ALAS:
$BRO TT10: ERROR RUNNING OLD.TSK
$RUN OLD.TSK
$GOTO REST
$OKAY:
$BRO TT10:WARNING RECEIVED
$REST:EOJ
```

Example A-2 Sample Batch Command Files

Digital Equipment Corporation • Bedford, MA 01730