digital

VAX/VMS
Guide to Using Command Procedures

Order No. AA-H782A-TE

VAX11

March 1980

This manual presents key concepts and techniques for developing command procedures using the VAX/VMS DIGITAL Command Language (DCL).

VAX/VMS
Guide to Using Command Procedures

Order No. AA-H782A-TE

SUPERSESSION/UPDATE INFORMATION: This is a new document for this release.

OPERATING SYSTEM AND VERSION: VAX/VMS V02

SOFTWARE VERSION: VAX/VMS V02

digital equipment corporation · maynard, massachusetts

First Printing, March 1980

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual presents key concepts and techniques for developing command procedures using the VAX/VMS DIGITAL Command Language (DCL). Many examples, including examples of complete command procedures, are included to demonstrate applications of the concepts and techniques discussed.

INTENDED AUDIENCE

All users of the VAX/VMS operating system can benefit from using command procedures. Command procedures can be constructed both to serve very simple purposes and to perform complex tasks that approximate the capabilities of a high-level programming language. For example, you can construct frequently used command sequences into command procedures and thereby save keystrokes; you can also code sophisticated command sequences that pass parameters, test status values, process files, and perform similar program-like tasks.

STRUCTURE OF THIS DOCUMENT

Each chapter in this manual builds on material presented in earlier chapters. Thus, if command procedure development is new to you, read the associated documents then study this manual sequentially beginning with Chapter 1. If, however, you are an experienced programmer with some knowledge of command procedures, you may want to simply skim the Table of Contents and Index for the specific topics you need.

This manual contains nine chapters and one appendix.

- Chapter 1, "Developing Command Procedures," defines command procedures and describes the steps in command procedure development.

- Chapter 2, "Controlling Command Procedure I/O," describes the system-defined logical name equivalences and how to use them to control input to and output from command procedures.

- Chapter 3, "Using Symbols in Command Procedures," describes how you can define and manipulate command symbols in command procedures.

- Chapter 4, "Symbol Substitution in Command Procedures," defines the mechanism of symbol substitution and describes ways you control symbol substitution in command procedures.

- Chapter 5, "Using Lexical Functions in Command Procedures," shows how to use the DCL lexical functions in command procedures to obtain information about the status of a process and to manipulate character strings.

- Chapter 6, "Controlling Execution Flow in Command Procedures," describes ways to use the DCL commands IF, GOTO, EXIT, and STOP to control the sequence in which command procedure lines are executed.

- Chapter 7, "Controlling Error Conditions and CTRL/Y Interrupts," shows how you can establish error condition routines based on the severity of errors encountered during command procedure execution and handle CTRL/Y interrupts that occur during command procedure execution.

- Chapter 8, "Creating, Reading, and Writing Files," describes how to manipulate sequential files using command procedures.

- Chapter 9, "Controlling Batch Jobs," describes how the system creates batch jobs and how you can control their execution from within command procedures.

- Appendix A, "Annotated Command Procedures," contains several complete command procedures that illustrate the techniques described in Chapters 1 through 9.

ASSOCIATED DOCUMENTS

The _VAX/VMS Guide to Using Command Procedures_ is not a stand-alone document. You should understand the material presented in the following manuals before using this guide:

- _VAX/VMS Primer_. This tutorial guide to the use of the VAX/VMS operating system introduces new users to the DIGITAL Command Language, the use of a text editor, interactive and batch mode operations, files and file specifications, and logical names.

- _VAX/VMS Summary Description and Glossary_. This manual is a technical summary of VAX/VMS concepts and components, including those process concepts of interest to the user of command procedures. The manual also contains a glossary of VAX-11 terms.

- _VAX/VMS Command Language User's Guide_. This manual is the primary reference document for information about DCL. The manual contains complete descriptions of all DCL commands except those most commonly used by system operators, defines the grammar for the DCL command language, defines file specification and usage, and (of particular use to command procedure developers) contains many examples of specific commands. You should use the _VAX/VMS Command Language User's Guide_ as a reference dictionary while studying the material in this manual.

CONVENTIONS USED IN THIS DOCUMENT

This manual uses the following graphic conventions.

ESC RET DEL TAB	These symbols indicate that you press the ESC, RETURN, DELETE, or TAB key on the terminal.
CTRL/Y CTRL/Z	These symbols indicate that you hold down the <CTRL> key while you press a terminal key, for example Y.

`$  show time`
`  05-JUN-1979 11:55:22`
	In examples of interactive dialog, all the lines you type are shown in red letters. Everything the system prints or displays is shown in black letters.

`$  FORTRAN MYFILE`
`$  LINK MYFILE`
`$  RUN MYFILE`
	When the contents of a command procedure are shown, the lines in the file are always shown in uppercase letters.

`$ LOOP:`
`  .`
`  .`
`  .`
`$ GOTO LOOP`
	A vertical ellipsis in an example means that not all the lines in the command procedure are shown; or that not all the data the system would display is shown.

quotation marks apostrophe	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe is used to refer to a single quotation mark (').

CHAPTER 1

DEVELOPING COMMAND PROCEDURES

A DCL command procedure is a file that contains a sequence of DCL commands.

Common uses for command procedures include constructing sequences of commands you frequently use during interactive terminal sessions and defining a batch job stream to submit from a terminal session or a system card reader. As you develop skill in applying command procedures, you will discover many other applications for them. The annotated examples in Appendix A of this manual illustrate only some of the uses of more complex command procedures.

In its simplest form, a command procedure consists of one or more command lines for the DCL command interpreter to execute. For example, a procedure to compile, link, and execute the FORTRAN program ALPHA could contain the lines:

```
$  FORTRAN/LIST ALPHA
$  LINK/MAP ALPHA
$  RUN ALPHA
```

In its most complex form, a command procedure can resemble a program coded in a high-level programming language: it can establish loops and error checking routines; perform arithmetic calculations and input/output operations; manipulate character string data; call other command procedures; pass parameters to other command procedures; and test values set in other command procedures.

This chapter summarizes the steps in command procedure development:

- Creating command procedures
- Formatting and documenting command procedures
- Executing command procedures
- Testing and debugging command procedures
- Maintaining command procedures

The remaining chapters in this guide discuss some of the key concepts and techniques for developing and using command procedures. Included is information on how to:

- Control command procedure input and output
- Pass parameters to and from command procedures
- Use symbols in command procedures

- Control symbol substitution in command procedures
- Use the DCL lexical functions in command procedures
- Direct the flow of command procedure execution
- Handle status returns and CTRL/Y interrupts in command procedures
- Create, read, and write sequential files using command procedures
- Control batch jobs and batch job queues from command procedures

The DCL commands that you use in command procedures are summarized in Table 1-1. Note that many of these commands can be used in contexts other than command procedures, but are particularly relevant to command procedures. You should use the VAX/VMS Command Language User's Guide as a reference for the grammar or use of any command, including those illustrated in this manual.

Table 1-1
Commands Frequently Used in Command Procedures

Name	Function
@filename	Executes a command procedure.
=	Arithmetic assignment; equates a local symbol name to an arithmetic expression or constant.
==	Arithmetic assignment; equates a global symbol name to an arithmetic expression or constant.
:=	String assignment; equates a local symbol name to a character string.
:==	String assignment; equates a global symbol name to a character string.
label:	Defines a label statement.
ASSIGN	Equates a logical name to a physical device name, to a complete file specification, or to another logical name, and places the equivalence name string in the process, group, or system logical name table.
CLOSE	Closes a file that was opened for input or output with the OPEN command and deassigns the logical name specified when the file was opened.
CONTINUE	Resume execution of a command procedure that was interrupted by pressing CTRL/Y or CTRL/C .

(continued on next page)

Table 1-1 (Cont.)
Commands Frequently Used in Command Procedures

Name	Function
DECK	Marks the beginning of an input stream for a command procedure when the first non-blank character in any data record in the input stream is a dollar sign ($).
DELETE/ENTRY	Deletes one or more entries from a printer or batch job queue.
DELETE/SYMBOL	Deletes a symbol definition from a local symbol table or from a global symbol table.
EOD	Marks the end of an input stream for a command procedure.
EOJ	Marks the end of a batch job submitted through a system card reader.
EXIT	Terminates processing of the current command procedure.
GOTO	Transfers control to a labeled statement in a command procedure.
IF...THEN	Tests the value of an expression and executes a command if the test is true.
INQUIRE	Requests interactive assignment of a value for a local or global symbol during the execution of a command procedure.
JOB	Marks the beginning of a batch job submitted through a system card reader.
Lexical Functions	Return information about character strings and attributes of the current process.
ON...THEN	Defines the default courses of action when a command or program executed within a command procedure (1) encounters an error condition or (2) is interrupted by CTRL/Y.
OPEN	Opens a file for reading or writing.
PASSWORD	Specifies the password associated with the user name specified on a JOB card for a batch job submitted through a card reader.
PRINT	Queues one or more files for printing, either on a default system printer or a specified device.
READ	Reads a single record from a specified input file and assigns the contents of the record to a specified logical name.

(continued on next page)

Table 1-1 (Cont.)
Commands Frequently Used in Command Procedures

Name	Function
SET CARD_READER	Defines the default ASCII translation mode for a card reader.
SET CONTROL_Y	Enables interrupts caused by CTRL/Y.
SET NOCONTROL_Y	Disables interrupts caused by CTRL/Y.
SET NOON	Prevents the command interpreter from performing error checking following the execution of commands.
SET NOVERIFY	Prevents command lines in a command procedure from being displayed at a terminal or printed in a batch job log file.
SET ON	Causes the command interpreter to perform error checking following the execution of commands.
SET QUEUE/ENTRY	Changes the current status or attributes of a file that is queued for printing or for batch job execution but not yet processed.
SET VERIFY	Causes command lines in a command procedure to be displayed at a terminal or printed in a batch job log file.
SHOW QUEUE	Displays the current status of entries in the printer and/or batch job queues.
STOP/ABORT	Aborts a job that is currently being printed.
STOP/ENTRY	Deletes an entry from a batch queue while it is running.
STOP/REQUEUE	Stops the printing of the job currently being printed and places that job at the end of the output queue.
SUBMIT	Enters one or more command procedures in the batch job queue.
SYNCHRONIZE	Places the process issuing this command into a wait state until a specified batch job completes execution.
WAIT	Places the current process in a wait state until a specified period of time has elapsed.
WRITE	Writes a record to a specified output file.

1.1 CREATING COMMAND PROCEDURES

There are several ways to create command procedures. Interactive users can create a command procedure by using a VAX/VMS-supported editor such as SOS or EDT or by using the DCL command CREATE. Batch job users can either (1) create a command procedure interactively and submit the command procedure from a terminal or (2) punch a card deck that includes the command procedure and submit the card deck to a system card reader.

The following examples show the creation of a simple command procedure by two different methods: the CREATE command and the SOS editor:

```
$  create fred.comRET
$  RUN ARET
$  RUN BRET
$  RUN CRET
CTRL/Z
$
```

```
$ edit sam.comRET
Input:  DBA1:[HIGGINS]SAM.COM;1
00100     $ RUN ARET
00200     $ RUN BRET
00300     $ RUN CRET
00400     ESC
 E
[DBA1:[HIGGINS]SAM.COM;1]
$
```

You can construct command procedures that contain only data to be read by a command or program; or that contain only qualifiers or parameters for a command; or that contain both. When you specify the Execute Procedure (@) command in any position in a command string, the command interpreter assumes that the at sign (@) character is followed by the name of a file with a file type of COM, and begins reading input from the specified command procedure.

For example, you could create a command procedure that contains a number of qualifiers you frequently use together when you issue a LINK command, as shown below:

```
DEBUG/SYMBOL_TABLE/MAP/FULL/CROSS_REFERENCE
```

If this command procedure is named DEFLINK.COM, you can request these qualifiers on a LINK command line to link an object module. The following example shows how to enter the LINK command to link an object module named SYNAPSE.OBJ:

```
$  LINK SYNAPSE@DEFLINKRET
```

Note that no space precedes the at sign (@) character in this example. If you type a space before the at sign, the command interpreter assumes that the command file contains a file specification for the LINK command. Because the LINK command allows only one file specification, an error would result when this command was parsed.

NOTE

Regardless of the method used to create it, a command procedure is a sequential file containing variable-length records.

1.2 FORMATTING AND DOCUMENTING COMMAND PROCEDURES

Each line in a command procedure represents a line that you want the DCL command interpreter to process. You enter the lines into the command procedure in the order in which you want the system to process them. For example, to create a command procedure file named TESTALL.COM that contains RUN commands to execute the program images named A.EXE, B.EXE, and C.EXE, you could create a file containing the following lines:

```
$  RUN A
$  RUN B
$  RUN C
```

To execute this command procedure from an interactive terminal session, you would use the Execute Procedure (@) command, as follows:

```
$  @TESTALL RET
```

When you create a command procedure, you must begin each line with a dollar sign ($), whether the line starts a command string or is a comment; you can precede or follow the dollar sign with no blank spaces or tabs or with one or more blank spaces or tabs.

The format used for the command strings themselves is the same as the format you would use to enter commands interactively, with the exception that you must begin each command string with a dollar sign.

1.2.1 Continuing Commands on More than One Line

You can continue any command string on more than one line by using the hyphen continuation character, just as you do for interactive command continuation. You must not, however, begin any continuation line with a dollar sign. For example:

```
$  PRINT TEST.OUT -
          /AFTER=18:00 -
          /COPIES=10 -
          /QUEUE=LPB0:
```

The qualifiers for this PRINT command are placed on separate lines in the command procedure for readability. The hyphen continuation character is used to indicate that the command continues after the first command line. The spaces preceding each qualifier are not required. They are included to make the command string more readable.

1.2.2 Documenting Command Procedures

Although no rules govern the precise format of lines in a command procedure, it is good programming practice to make your command procedures self-documenting so that they are easy to read and to maintain. The techniques described in the following sections are useful for clear command procedure documentation.

1.2.2.1 **Use Comments** - Comments are as important in command procedures as they are in source programs and should be used frequently. Whether a comment is a separate line or part of a line in a command procedure, always precede it with an exclamation point (!). For example:

```
$ !      FRED.COM                    VAX/VMS V2.0
$ !
$ !      COMPILES, LINKS, RUNS ALPHA.FOR
$ !
$ FORTRAN/LIST ALPHA                         ! COMPILE
$ LINK/MAP ALPHA                             ! LINK
$ RUN ALPHA                                  ! GO
```

If you must use a literal exclamation point in a command line, enclose it in quotation marks, so the command interpreter will not interpret the exclamation point as a comment delimiter. Note that the exclamation point character can be used in data lines because data lines do not begin with a dollar sign and are not processed by the command interpreter.

If you want to include symbol names or lexical functions in comments, you can do so if you precede each symbol name with an apostrophe, the command interpreter performs symbol substitution in comment lines during its lexical processing. When the command is parsed and executed, however, the command interpreter ignores the exclamation point and all data following the exclamation point.

1.2.2.2 **Spell Out Command and Qualifier Names** - Do not truncate DCL command and qualifier names used in command procedures. Although the grammar rules of DCL allow truncation, it is wise to spell out full command and qualifier names to ensure that ambiguous abbreviations do not occur in the future.

Moreover, a procedure that spells out command names and qualifiers is self-documenting, as the DCL commands and qualifiers are generally named according to the functions they perform. For example, compare the following two lines:

```
$ PRINT ALPHA.LIS/COPIES=2
$ PR ALPHA/C=2
```

The first command line expresses clearly the request to print two copies of the file ALPHA.LIS. The second line is terse and may not be easily interpreted by other users (or remembered by yourself).

1.2.2.3 **Use Indentation** - In longer command procedures, you can improve readability by tabbing command lines to offset them from labels you use. For example:

```
           $ COUNT = 1
$LOOPER:
           $ IF COUNT .GT. 10 THEN GOTO ENDLOOP
           $ DEFINE SWITCH 'COUNT'
           $ RUN ALPHA
           $ COUNT = COUNT + 1
           $ GOTO LOOPER
$ENDLOOP:
```

In the example above, the labels LOOPER and ENDLOOP clearly delimit the portion of the command procedure that performs this particular loop.

The commands IF and GOTO, and techniques for constructing loops in command procedures are described in Chapter 6, "Controlling Execution Flow in Command Procedures."

1.3 EXECUTING COMMAND PROCEDURES

When you log in to the VAX/VMS operating system, the system creates a detached process for you, and assigns to the process the user privileges, execution priority, and resource quotas that determine the process context -- the nature of the images that the process will be allowed to execute.

When you issue the Execute Procedure (@) command from a detached process, the command interpreter returns control to the interactive DCL command level only after the command procedure either has been successfully executed or has been terminated, for example as the result of an error condition. Command procedure execution is serial, just as is the execution of any image within a detached process. Thus, you cannot do any interactive work from the process while the command procedure is being executed.

However, when you issue the SUBMIT command from a detached process, a separate process is created for the batch job defined by the SUBMIT command. As soon as the batch job is queued, but before the job is executed, control is returned to the process that executed the SUBMIT command. The batch process created by the system executes independently from the submitting process, allowing you to do useful work at the terminal. Upon completion, the operating system deletes the batch process.

Thus, there are two operating modes for command procedures, interactive mode and batch mode. You execute a command procedure in interactive mode only when you issue the @ command during an interactive terminal session or from a command procedure. The only other method of initiating a command procedure (issuing the SUBMIT command) results in the creation of a separate process used to run a batch job.

You can execute DCL command procedures in five different ways.

- You can issue the Execute Procedure (@) command during an interactive terminal session. This method is called executing command procedures from interactive mode.

- You can issue the SUBMIT command during an interactive terminal session. This method is called submitting command procedures for batch execution.

- You can place a card deck that contains a command procedure in a system card reader. This method is called submitting batch jobs through the card reader.

- You can issue the Execute Procedure (@) command from a command procedure. This method is called executing nested command procedures.
- You can construct a special command procedure file, called a login command file, that VAX/VMS automatically attempts to locate and execute each time you log in to the operating system.

The following sections illustrate these methods of command procedure execution.

1.3.1 Executing Command Procedures in Interactive Mode

When you execute a command procedure in interactive mode, first enter the Execute Procedure (@) command and then enter the file specification of the command procedure. The command interpreter assumes your current disk and directory defaults and a default file type of COM. For example, to execute the command procedure WEATHER.COM in your default disk and directory, issue this command:

```
    $ @WEATHER(RET)
```

Figure 1-1 illustrates the execution of a command procedure in interactive mode. When you enter the Execute Procedure (@) command, the command interpreter finds, then executes the file TESTALL.COM in your default disk and directory. Each command string in TESTALL.COM is then executed sequentially. When the end-of-file for TESTALL.COM is reached, the command interpreter returns control to the interactive command level and issues the dollar sign prompt at your terminal. You now can resume interactive work.

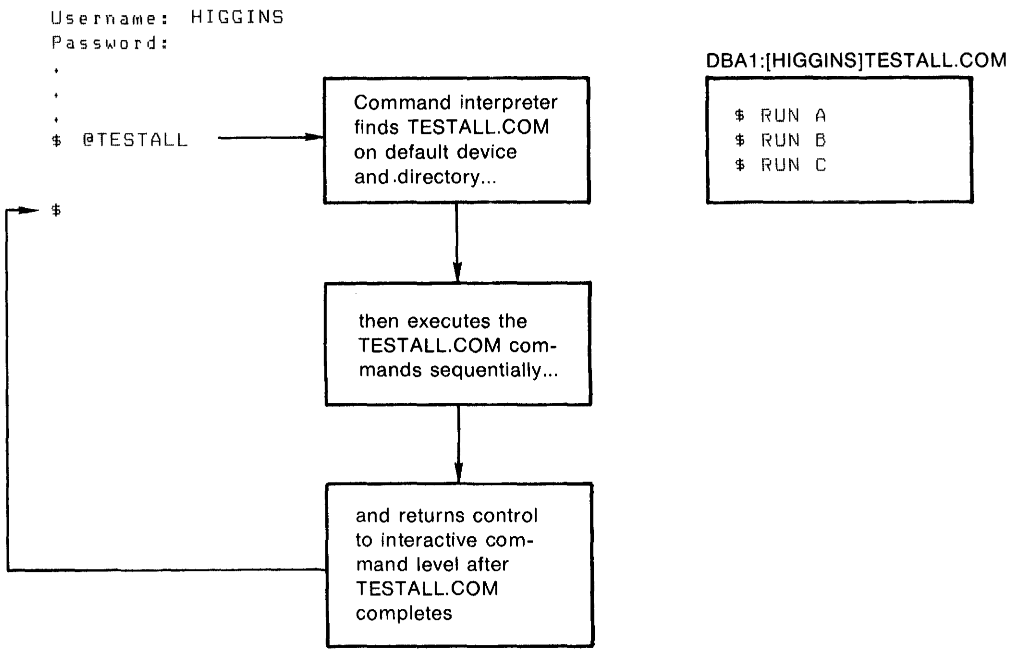

Username: HIGGINS
Password:
.
.
.
$ @TESTALL
$
Command interpreter finds TESTALL.COM on default device and directory...
then executes the TESTALL.COM commands sequentially...
and returns control to interactive command level after TESTALL.COM completes
DBA1:[HIGGINS]TESTALL.COM
$ RUN A
$ RUN B
$ RUN C

Figure 1-1 Executing a Command Procedure in Interactive Mode

If a command procedure is not in your default disk and directory, or does not have the file type COM, give the complete file specification, as shown in the following example:

```
     $  @DBB2:[COMMON]SETUP.FIL(RET)
```

This command executes a command procedure that is located on the disk DBB2 in the directory COMMON. The command procedure file name is SETUP.FIL.

For command procedures that you execute frequently, you can define a symbol name as a synonym for the entire command line. For example:

```
     $  SETUP := @DBB2:[COMMON]SETUP.FIL.(RET)
```

This is an assignment statement that defines the symbolic name SETUP to be equivalent to the string @DBB2:[COMMON]SETUP.FIL. This symbol can be used, for example, as a command name during the current terminal session.

If you wanted to be able to use this symbol every time you logged in to VAX/VMS, you would include this symbol in a global assignment in your login command file. Refer to Section 1.3.5, "Using Login Command Files," for this method of executing a command procedure.

1.3.2 Submitting Command Procedures for Batch Execution

If you use the Execute Procedure (@) command interactively, you cannot enter other commands to do other work while the procedure is executing. If you create and use procedures that require lengthy processing time -- for example, the compilation or assembly of large source programs -- you can submit the procedure for execution as a batch job instead of using the Execute Procedure command. Once the batch job is queued by the operating system, your terminal is free for you to continue interactive work.

The SUBMIT command requests the operating system to enter a command procedure into a batch job queue. The SUBMIT command assumes your current disk and directory defaults and it assumes a default file type of COM for the command procedure. For example, to execute the command procedure TESTALL.COM in your default disk directory, you could issue the command:

```
     $  SUBMIT TESTALL
        Job 210 entered on queue SYS$BATCH

     $
```

In this example, the system displays a message showing that the job has been queued; the message gives you the job number (210) and the name of the system queue on which it entered the job (SYS$BATCH). Batch queues are normally set up and started by the system manager or the system operator; in most cases, one of these queues will be named SYS$BATCH.

Figure 1-2 illustrates how the SUBMIT command is used to queue the file TESTALL.COM as a batch job. Although control is returned to you as soon as the job is queued successfully, TESTALL.COM is not executed until the operating system creates a process for it. The execution of the batch job begins with an automatic login to your account and an execution of your login command file (if you have one). Note, however, that the batch job is a separate process with its own unique process context; for example, it cannot access symbols that you define interactively.

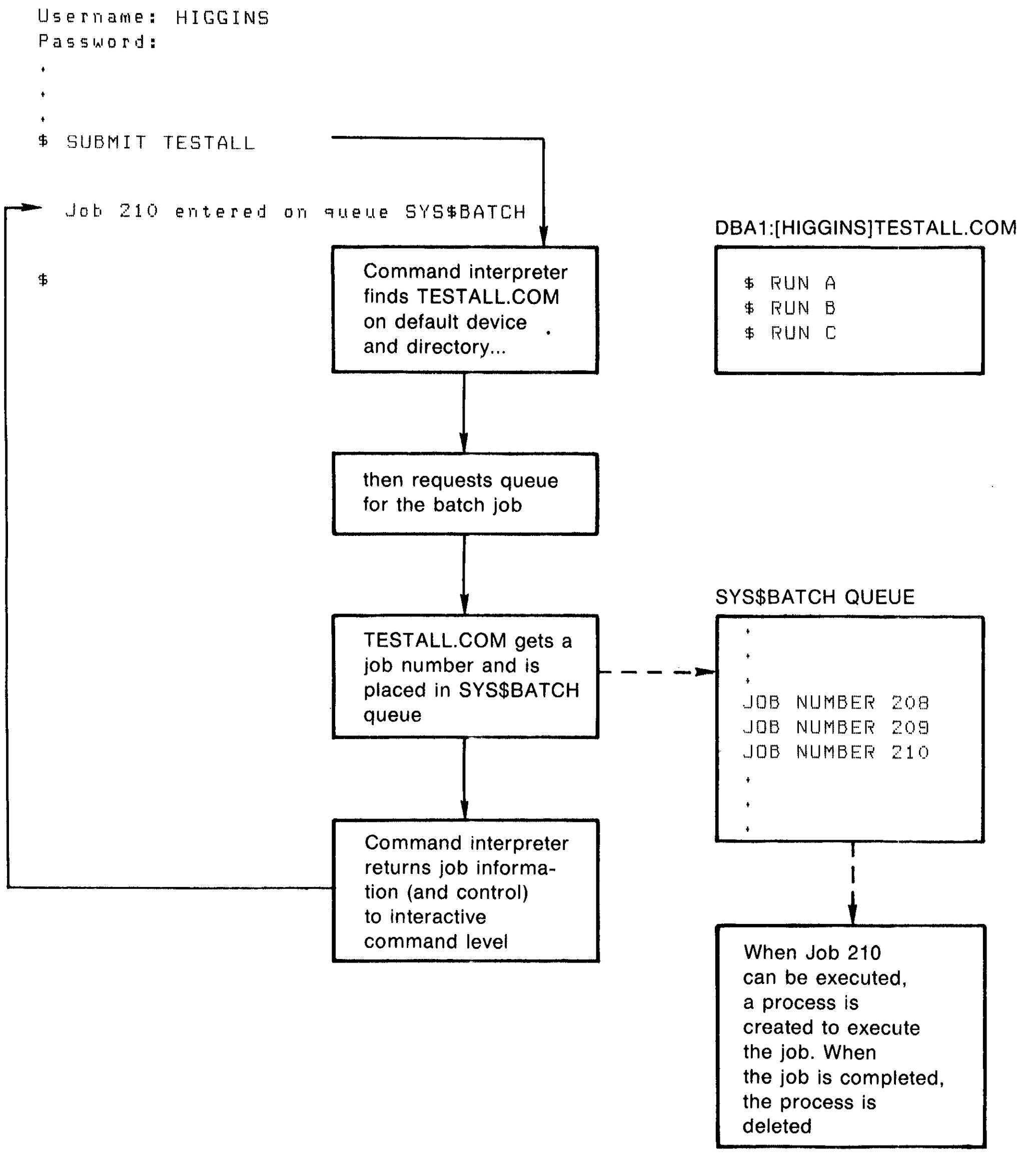

Username: HIGGINS
Password:
$ SUBMIT TESTALL
Job 210 entered on queue SYS$BATCH
$
Command interpreter finds TESTALL.COM on default device and directory...
then requests queue for the batch job
TESTALL.COM gets a job number and is placed in SYS$BATCH queue
Command interpreter returns job information (and control) to interactive command level
DBA1:[HIGGINS]TESTALL.COM
$ RUN A
$ RUN B
$ RUN C
SYS$BATCH QUEUE
JOB NUMBER 208
JOB NUMBER 209
JOB NUMBER 210
When Job 210 can be executed, a process is created to execute the job. When the job is completed, the process is deleted

Figure 1-2 Submitting a Command Procedure to a Batch Job Queue

1.3.3 Submitting Batch Jobs Through the Card Reader

When you submit a batch job through a system card reader, you must precede the card deck containing the command procedure with cards containing JOB and PASSWORD commands. These cards specify your user name and password and, when executed, effect a log in for you. The last card in the deck must contain the EOJ command. The EOJ card, when executed, is equivalent to logging out. Figure 1-3 illustrates a card reader batch job.

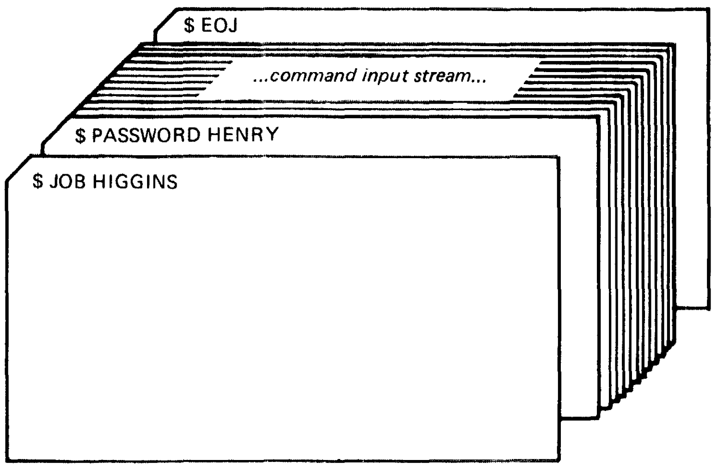


Figure 1-3 A Card Reader Batch Job

Note that you can prevent other users from seeing your password by suppressing printing when you keypunch the PASSWORD card.

When the system reads a job from the card reader, it validates the user name and password specified on the JOB and PASSWORD cards. Then, it copies the entire card deck into a temporary disk file named INPBATCH.COM in your default directory and queues the job for batch execution. Thereafter, processing is the same as for jobs submitted interactively with the SUBMIT command. When the batch job is completed, the operating system deletes the INPBATCH.COM file.

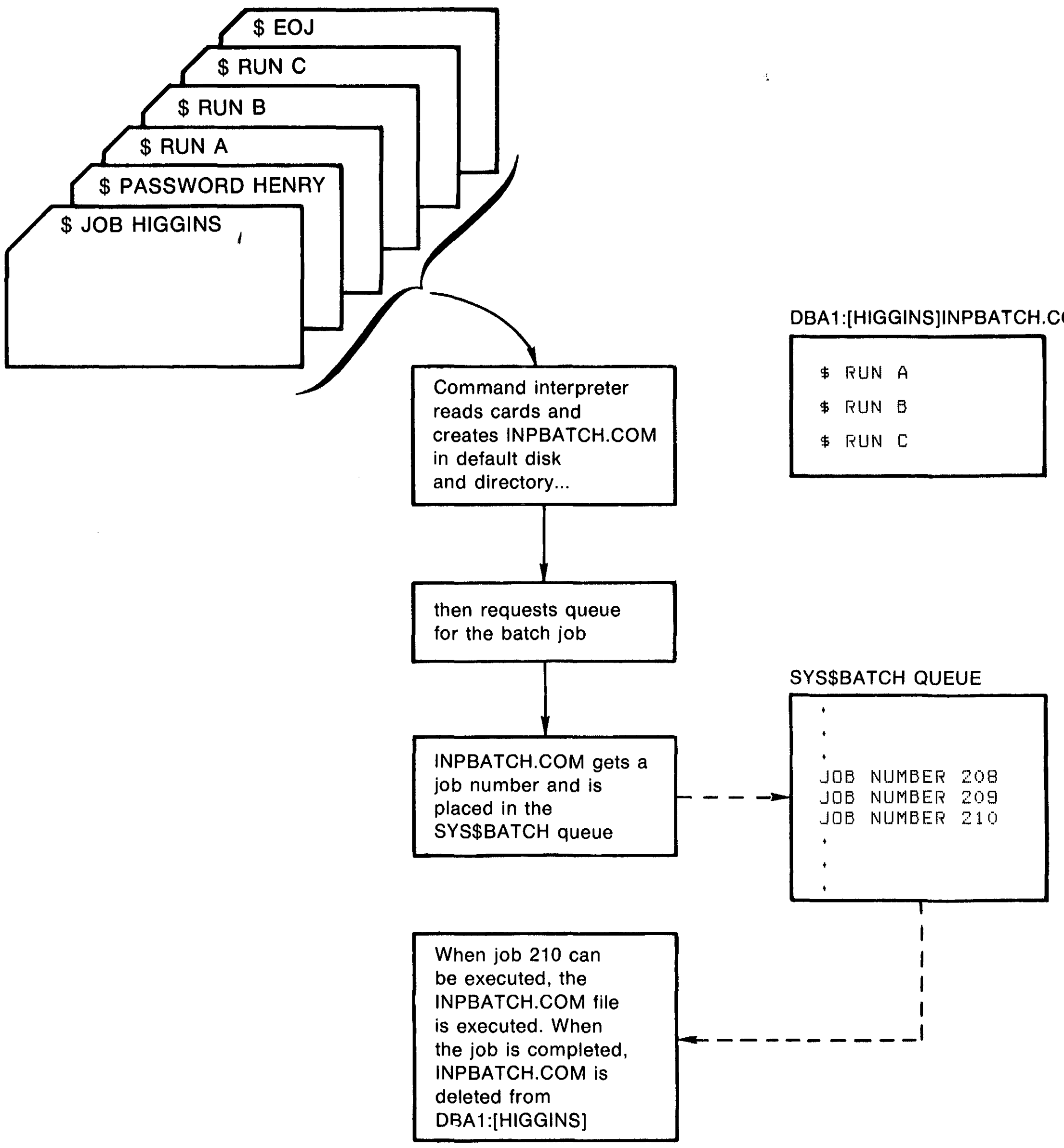
When the system reads input from the card reader, it also recognizes two special types of card:

- Translation mode cards
- EOF cards

Translation mode cards in the batch job's input stream change the current translation mode. The translation mode is based on the device type of the card punch on which the cards were punched. An 026 punch is indicated by an 026 translation mode card (12-2-4-8 overpunch). An 029 card punch is indicated by an 029 translation mode card (12-0-2-4-6-8 overpunch). The default card translation mode can be set with the SET CARD_READER command.

An EOF card (12-11-0-1-6-7-8-9) overpunch or card containing an EOJ command signals the end of the job.

Figure 1-4 illustrates a batch job submitted through a system card reader. The command interpreter reads the cards and creates a file, INBATCH.COM, in the user's default disk and directory. The system queues the job in the SYS$BATCH queue. After the job is executed, the system deletes the file INBATCH.COM from the user's default disk and directory.



$ EOJ
$ RUN C
$ RUN B
$ RUN A
$ PASSWORD HENRY
$ JOB HIGGINS
Command interpreter reads cards and creates INPBATCH.COM in default disk and directory...
then requests queue for the batch job
INPBATCH.COM gets a job number and is placed in the SYS$BATCH queue
When job 210 can be executed, the INPBATCH.COM file is executed. When the job is completed, INPBATCH.COM is deleted from DBA1:[HIGGINS]
DBA1:[HIGGINS]INPBATCH.COM
$ RUN A
$ RUN B
$ RUN C
SYS$BATCH QUEUE
JOB NUMBER 208
JOB NUMBER 209
JOB NUMBER 210

Figure 1-4 Submitting a Batch Job Through a System Card Reader

1.3.4 Executing Nested Command Procedures

Command procedures can be nested. That is, one command procedure can contain an Execute Procedure (@) command to execute another command procedure. In this case, the command interpreter reads input from the second command procedure file until it reaches the end of the file or until that procedure exits; then returns control to the first command procedure.

The maximum number of command procedures you can nest is eight. For more information on nesting command procedures, refer to Section 2.1, "System-Defined Logical Name Equivalences," and Section 6.3, "Nesting Procedures: The Execute Procedure Command."

1.3.5 Using Login Command Files

There is a special type of procedure, called a login command file, or simply login file, that the system automatically attempts to locate and execute each time you log in to the operating system. This file also is executed automatically, if present, at the beginning of every batch job you submit.

1.3.5.1 The LOGIN.COM File

Use the LOGIN.COM file to execute any commands or sequences of commands that you typically execute at the start of each terminal session. For example, if you define synonyms for DCL commands, you can place the global assignment statements for the command name synonyms in your LOGIN.COM file so they will be available every time you log in. The LOGIN.COM file can also contain commands to set up terminal characteristics, assign logical names, run programs, execute command procedures, or display message files. For example, a LOGIN.COM file could contain the following statements:

```
$  QP  :== SHOW QUEUE/DEVICE/FULL
$  QB  :== SHOW QUEUE/BATCH/FULL
$  TIM :== SHOW TIME
$  SET PROTECTION = (GROUP:RE,WORLD)/DEFAULT
$  ASSIGN DBA1:[MALCOLM.PAYROLL] PAY
$  ASSIGN DBA1:[MALCOLM.DOCUMENTS] DOC
$  TYPE SYS$SYSTEM:NOTICE.TXT
```

You can use the Execute Procedure (@) command to test your LOGIN.COM file. You can also set up your LOGIN.COM file so that it executes different commands depending on whether the current process mode is interactive or batch. For an example of how to do this, see Section 5.2.1, "The F$MODE Lexical Function."

When you create your LOGIN.COM file, you must locate it on your default disk and directory and you must name it LOGIN.COM. After that, you use and maintain it just like any other command procedure.

1.3.5.2 A System-Defined Login File

The system manager, who authorizes use of the system, can optionally specify for each user a login file to be executed at the start of a terminal session or batch job. When a system manager has specified a system-defined login file, the commands in that file take precedence over any user-defined login file. However, the system-defined login file can, and usually does, contain the command:

```
$  @LOGIN
```

Then, the commands individual users want to execute can be executed in addition to the commands in the system-defined login file. The relationship between the two login files is illustrated in Figure 1-5. The file specification for the system-defined login file is SYS$MANAGER:SYSLOGIN.COM, the file specification for the user-defined login file is DBA1:[HIGGINS]LOGIN.COM. When user HIGGINS logs in, the system-defined login file executes first. In this case, it contains the command @LOGIN, so the command interpreter locates and executes the user-defined login file on the default disk and directory for user HIGGINS, then returns to the system-defined login file which completes the login process.

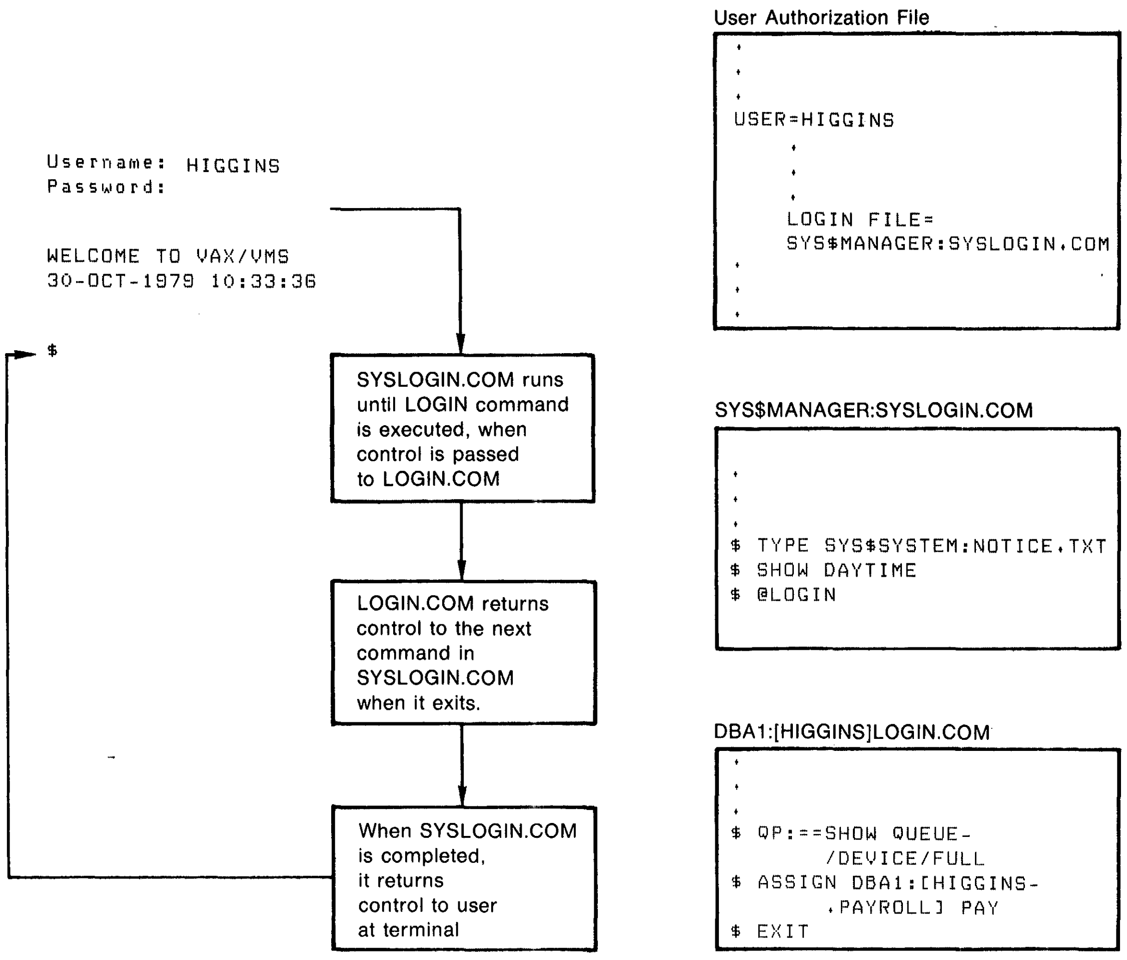


Figure 1-5 Executing System and User Login Files

NOTE

In some installations, a system-defined login file can control an entire terminal session. Such a procedure, which can actually restrict the commands a user is allowed to execute, is illustrated in the sample command procedure FORTUSER.COM in Appendix A.

1.4 TESTING AND DEBUGGING COMMAND PROCEDURES

Typically, command procedures need to be tested, then debugged, before you can use them with complete confidence. You can debug command procedures by controlling the input and output to them and by use of the commands SET VERIFY and SET NOVERIFY. Methods for debugging command procedures are discussed in Section 2.3, "Verifying Command Procedure Execution."

1.5 MAINTAINING COMMAND PROCEDURES

If the command procedures you develop are correctly formatted, carefully documented, and verified, their maintenance is relatively easy. Because new versions of the VAX/VMS operating system may include enhancements to the DCL command language, you should be aware of new commands and any changes to current commands.

Generally, DIGITAL makes changes to the DCL commands (and to the functions of the DCL command interpreter) only to add new features, and to correct errors, but there may be occasions when a new release changes the format or results of a particular command, command parameter, or qualifier. For effective maintenance, study the release notes issued with each VAX/VMS release for the effect, if any, of changes to the DCL command language or the DCL command language processor.

CHAPTER 2

CONTROLLING COMMAND PROCEDURE I/O

This chapter discusses the concepts and techniques you use to control input to and output from command procedures. The topics covered are:

- How the system-defined equivalences for process logical names function at different command levels
- How to use the SET VERIFY command as a debugging aid and as a means of controlling responses and messages from DCL commands and programs
- How to write command procedure output to a disk file
- How to include command or program data in a command procedure
- How to use the ASSIGN command to redefine equivalences for SYS$INPUT and SYS$OUTPUT
- How to display data at your terminal or place data in a batch job's output stream

2.1 SYSTEM-DEFINED LOGICAL NAME EQUIVALENCES

When you log in, the operating system creates a detached process for you and establishes the initial equivalences to the following process logical names:

- SYS$INPUT -- The default command and data input stream for this process. The command interpreter uses SYS$INPUT to read commands and data that are required by commands.
- SYS$OUTPUT -- The default output stream for commands and program images that execute in this process. The command interpreter uses SYS$OUTPUT when it issues prompting and informational messages.
- SYS$ERROR -- The default error message stream for this process. The command interpreter writes error and warning messages to SYS$ERROR.
- SYS$COMMAND -- The initial command input stream for this process. The command interpreter uses SYS$COMMAND to "remember" the original input device.

- SYS$DISK -- The default device for this process. The command interpreter uses this equivalence to fill in the device name portion of file specifications.

- SYS$LOGIN -- The device and directory that are the defaults when you log in.

When you execute a command procedure, the command interpreter provides a new equivalence name for SYS$INPUT, equating it to the command procedure itself. This equivalence overrides the original assignment for the duration of the command procedure.

When command procedures are nested, the command interpreter redefines the equivalence name for SYS$INPUT, equating it to the file from which the current command procedure is read. That is, the SYS$INPUT equivalence is changed as the current command level changes.

The logical names SYS$ERROR, SYS$DISK, and SYS$COMMAND do not change. SYS$COMMAND is always equated to the initial command level: if you execute a command procedure interactively, SYS$COMMAND is always equated to your terminal; if you submit a batch job, SYS$COMMAND is always equated to the initial batch input file. The equivalence for SYS$OUTPUT does not change unless the /OUTPUT qualifier is specified when you enter the Execute Procedure (@) command.

In other words, the initial command input level, command level 0, is the level at which, by default, SYS$INPUT and SYS$COMMAND are the same.

Figure 2-1 illustrates logical name assignments at various command levels.

2.2 VERIFYING COMMAND PROCEDURE EXECUTION

By default, the output from a command procedure executed interactively is displayed on the terminal. This output includes:

- Responses and messages from DCL commands

- All data messages displayed by programs that write to SYS$OUTPUT and SYS$ERROR

If you also want to see the DCL commands and comment lines displayed at the terminal, you can use the SET VERIFY command. Issue SET VERIFY either within the command procedure or at the interactive command level; the command affects all command procedures you subsequently execute during the terminal session.

For example, to display lines in a particular command procedure, you could place the SET VERIFY command at the beginning of the procedure and place the SET NOVERIFY command at the end of the procedure, as follows:

```
$ SET VERIFY
$ RUN TESTA
$ RUN TESTB
$ SET NOVERIFY
```

The SET NOVERIFY command at the end of this procedure restores the default setting for interactive command procedure execution.

Note that verifying a command procedure's execution is the principal debugging tool for detecting errors in a command procedure. If SET NOVERIFY is in effect and an error occurs, it may be difficult to determine which command caused the error. With SET VERIFY in effect, it is much easier to determine the cause of the error.

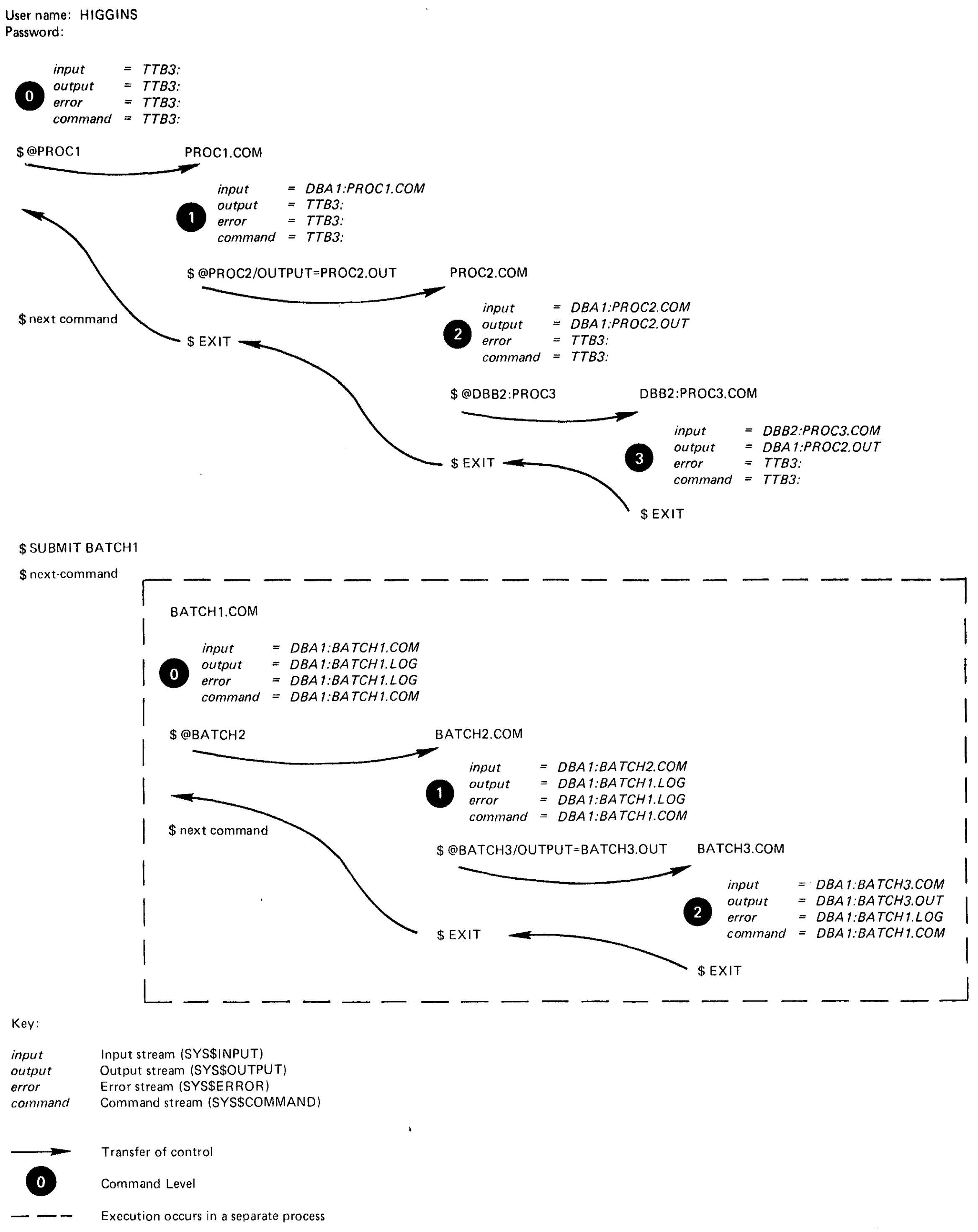


Figure 2-1 Logical Name Assignment at Different Command Levels

2.2.1 Verification in Batch Jobs

In a batch job, the verification is set on by default; all DCL commands and comment lines are written to the batch job log file with the command responses and messages.

You can use the SET NOVERIFY command in a batch job to suppress verification. For example, if a procedure loops around a command or set of commands, you might want to suppress verification while the loop executes and restore it afterward.

You can also use the SET NOVERIFY command at the beginning of your LOGIN.COM file. Otherwise, the contents of this file will appear at the beginning of every batch job log. However, you must include a SET VERIFY command at the end of your LOGIN.COM file if you want the batch job log file to exclude the login file while containing verification information of the batch job.

2.2.2 Changing Verification Settings

The SET VERIFY and SET NOVERIFY commands are executed within the command interpreter and therefore can be issued while a command procedure is executing without affecting the command or program image currently executing.

For example, if you interactively execute a command procedure that does not contain the SET VERIFY command and you decide, after execution begins, that you want to see the DCL command lines displayed, you can interrupt the procedure by pressing CTRL/Y as shown below:

```
$ @MASTER RET
CTRL/Y

^Y

$ SET VERIFY RET
$ CONTINUE RET
$ !         The next step in this procedure is to concatenate all
$ !         related files into a single master file before print-
.
.
.
```

In the above example the Execute Procedure (@) command runs the procedure MASTER.COM. Then, CTRL/Y is pressed to interrupt the procedure's execution, causing the command interpreter to prompt for command input. When the SET VERIFY command is entered, the default verification setting is changed so that the lines in the procedure will be displayed on the terminal. The CONTINUE command is issued to resume execution of the command procedure. The next few lines displayed, in this case, are comment lines in the procedure.

You can write a command procedure that tests the current verification setting, changes it if necessary, and restores the original setting before the command procedure completes execution. This technique requires an understanding of the lexical function F$VERIFY (see Section 5.2.2).

2.3 CONTROLLING INTERACTIVE OUTPUT

When you use the Execute Procedure (@) command interactively, the system equates the logical device SYS$INPUT with the command procedure, while SYS$OUTPUT and SYS$ERROR remain assigned to your terminal. In this way, output resulting from command or program execution and system messages are displayed on your terminal.

If you want a permanent record of the output from the execution of a command procedure, you can use the /OUTPUT qualifier of the Execute Procedure (@) command. The /OUTPUT qualifier redefines the equivalence name for SYS$OUTPUT from the default (the terminal) to a disk file. For example:

```
$ @TESTALL/OUTPUT=TESTALL.LOG<RET>
```

When you issue this command, all the data that is normally displayed on your terminal as TESTALL.COM is executed is instead written to the disk file named TESTALL.LOG. To determine the outcome of the command procedure, you can use the TYPE command to display the file or the PRINT command to print it. For example:

```
$ TYPE TESTALL.LOG<RET>
```

Note that most DCL commands and VAX/VMS utilities write warning and error messages to both SYS$OUTPUT and SYS$ERROR. Therefore, when you use the /OUTPUT qualifier to redefine SYS$OUTPUT, you will still see all error and warning messages that occur during the execution of the command procedure, even though all data will be written only to SYS$OUTPUT.

Table 2-1 summarizes the output from a command procedure, based on the output device and whether verification is on or off.

Table 2-1
Summary of Command Procedure Output

Output Device	VERIFY	NOVERIFY
SYS$OUTPUT (terminal)	Terminal displays: All DCL command lines and comment lines	Terminal displays: All messages to SYS$OUTPUT and SYS$ERROR
Log File (/OUTPUT qualifier specified)	Terminal displays: All messages to SYS$ERROR Log File contains: All DCL command lines and comment lines All messages to SYS$OUTPUT	Terminal displays: All messages to SYS$ERROR Log File contains: All messages to SYS$OUTPUT

2.4 INCLUDING COMMAND AND PROGRAM DATA IN COMMAND PROCEDURES

In a command procedure, you can issue a command that requires input data or run a program that requires input data. By default, the input data will be read from SYS$INPUT, the command input stream. SYS$INPUT is the command procedure.

For example, when you issue the CREATE command, the system reads input lines for a file from the command input stream. When you issue the CREATE command interactively, the command input stream (SYS$INPUT) is your terminal, and you indicate the end of the input data by pressing CTRL/Z.

When you include the CREATE command in a command procedure, the input stream is the command procedure itself. You include the input data lines for the file in the command procedure, immediately following the CREATE command line. The following example illustrates a command procedure that creates a file named WEATHER.DAT, includes the data lines that make up WEATHER.DAT, and issues the RUN command for a program that reads the WEATHER.DAT file.

```
$ CREATE WEATHER.DAT
JAN        39      3
FEB        42      1
MAR        50      7
.
.
.
DEC        46      25
$ RUN WEATHER.FOR
```

In this example, the end of the input data for the file WEATHER.DAT is indicated by the RUN command line. The end of input data for any command or program that is reading input data from a command procedure is indicated by a line that begins with the dollar sign ($) character, or by the physical end-of-file of the command procedure.

To include data lines that begin with dollar signs in the input stream, you must define the input data in a way that prevents the command interpreter from attempting to execute the data as a command. To delimit such an input stream, you use the DECK and EOD (End of Deck) commands. These commands are particularly useful for batch users who submit all work through the system card reader. For example, if you use the CREATE command to write a command procedure into a disk file, you would use the DECK and EOD commands as shown in Figure 2-2, which illustrates a batch job that creates and executes the command procedure WEATHER.COM.

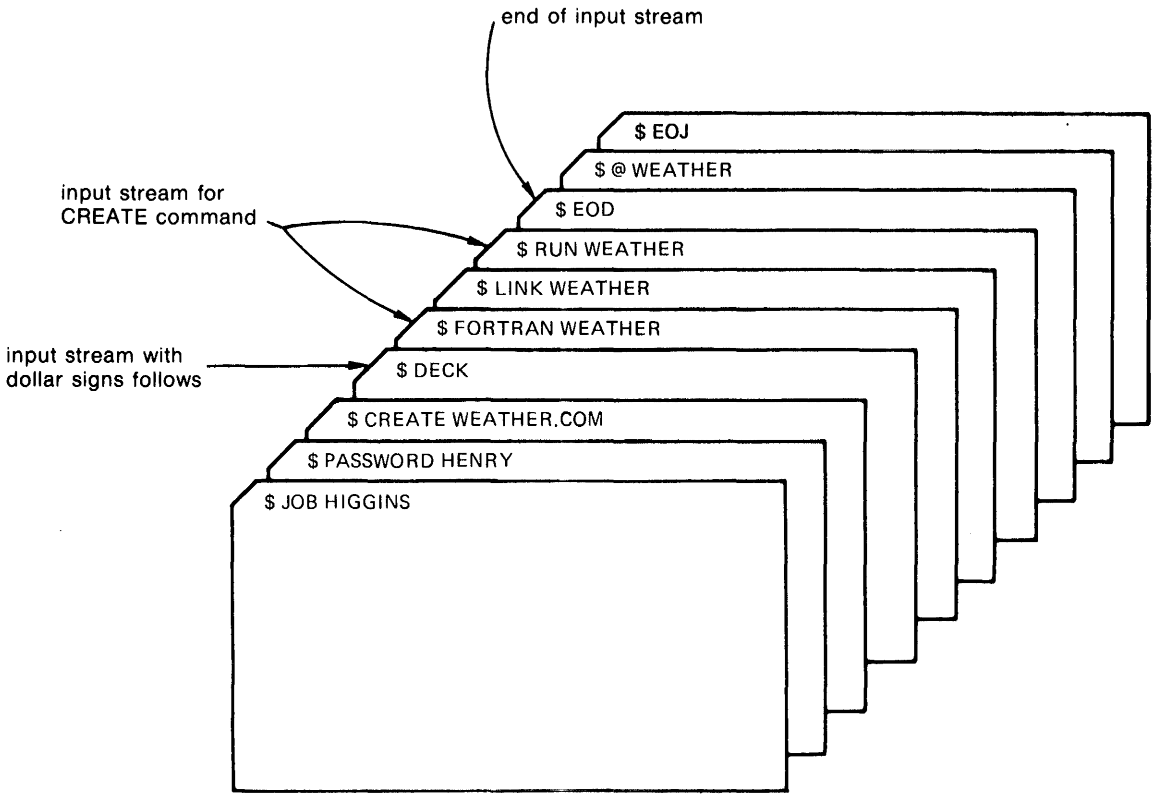


Figure 2-2 An Input Data Stream with Dollar Signs

You can also place input statements for a compiler into a command procedure's input stream by specifying the name of the data file as SYS$INPUT. The compiler will read its input from the command procedure. The following example illustrates a command procedure that contains a FORTRAN command followed by source statements:

```
$ FORTRAN/LIST SYS$INPUT:TESTER
C  THIS IS A TEST PROGRAM
   A = 1
   B = 2
   STOP
   END
$ PRINT TESTER.LIS
```

In this example, the file specification given to the FORTRAN command includes the device specification SYS$INPUT. Thus, the compiler reads the statements following the FORTRAN command (up to the next line that begins with a dollar sign) instead of looking in your default device and directory for a source program named TESTER.FOR. When the compilation is completed, two output files are created: TESTER.OBJ and TESTER.LIS. The PRINT command is then executed to print the output listing file.

2.5 REDEFINING SYS$INPUT AND SYS$OUTPUT

The techniques shown in the preceding section are particularly useful in batch applications. In interactive applications, they can be used for iterative testing of programs under development. For example, consider the following command procedure:

```
$ FORTRAN AVERAGE
$ LINK AVERAGE
$ RUN AVERAGE
33
66
99
9999
```

In this example, the FORTRAN, LINK, and RUN commands compile, link, and execute an interactive program that normally reads its input from the terminal. In this case, the data is read from the command procedure so the procedure can be used to test the source program each time it is revised.

You can use this technique whenever you can provide a program with a nonvarying set of input data. However, you may want to run a program from a command procedure and supply the program with input from the terminal. For example, if you want to run the program AVERAGE and supply its input data from the terminal, you must redefine the input stream. To do so, you include an ASSIGN command in the command procedure before the RUN command:

```
$ ASSIGN SYS$COMMAND:  SYS$INPUT:
$ RUN AVERAGE
.
.
.
```

This ASSIGN command redefines the input stream and equates it to the initial command stream (SYS$COMMAND). When the program AVERAGE is executed it reads input from the terminal rather than from the command procedure.

When an ASSIGN command in a command procedure equates SYS$INPUT to SYS$COMMAND, all subsequent programs (other than the command interpreter itself) that read input from SYS$INPUT will actually read the input from SYS$COMMAND (that is, from the terminal). For example:

```
$ ASSIGN SYS$COMMAND: SYS$INPUT:
$ EDIT AVERAGE.FOR
$ FORTRAN AVERAGE
$ LINK AVERAGE
$ RUN AVERAGE
```

In this example, both the EDIT and RUN commands invoke interactive programs that normally read from SYS$INPUT. In this procedure, however, both of the programs run by these commands will read input from the current SYS$COMMAND device, the terminal. When the editing session is completed, the next command in the procedure is executed. At the end of the procedure, the command interpreter restores the defaults associated with the initial command level in the terminal session.

Note that changing the assignments for the logical names SYS$INPUT and SYS$COMMAND does not affect the device from which the command interpreter reads its input: such devices are known to the command interpreter from the time you log in. In the case of a batch job, the devices are known from the beginning of the job.

2.5.1 User Mode Assignments

When you use an ASSIGN command in a command procedure to change the equivalence of a logical name for a process-permanent file (such as SYS$INPUT), you cannot use the DEASSIGN command to cancel the equivalence. The logical name can be reassigned only by the command interpreter, and then only when a command level change occurs.

This command-interpreter restriction would prevent use of a command procedure like the following:

```
$ ASSIGN SYS$COMMAND: SYS$INPUT:
$ EDIT AVERAGE.FOR
$ FORTRAN AVERAGE
$ LINK AVERAGE
$ RUN AVERAGE
33   }
66   }
99   } this input data will be ignored
9999 }
```

In this example, the ASSIGN command changes SYS$INPUT so that the editor can be run from the command procedure. Later, the input data for the program AVERAGE follows the RUN command in the procedure. The ASSIGN command, however, has redefined the input stream and this assignment (the terminal) is still in effect. When the RUN command executes the AVERAGE program, AVERAGE will attempt to read its data from the terminal instead of reading from the command procedure: the actual input data will be ignored.

To prevent this problem, use the /USER_MODE qualifier in the ASSIGN command. A user-mode logical name assignment exists only for the execution of one program image, in this example, for the duration of the editing session:

```
$ ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT:
$ EDIT AVERAGE.FOR
$ FORTRAN AVERAGE
$ LINK AVERAGE
$ RUN AVERAGE
33
66
99
9999
```

When the editing session is over, the command interpreter automatically cancels the logical name assignment for SYS$INPUT and restores the default for the current command level. Then, when the AVERAGE program reads input data from SYS$INPUT, it reads the data that is in the command input stream.

NOTE

For another example of running the editor from a command procedure file, see the sample procedure EDITALL.COM in Appendix A.

2.5.2 Suppressing Output

Many commands or programs that you execute produce output and display this output (by default) on SYS$OUTPUT. When you execute a command procedure, you may want to suppress this output or direct it to another file. You can do this by redefining SYS$OUTPUT. For example:

```
$ ASSIGN/USER_MODE STATISTIC.SRT SYS$OUTPUT:
$ SORT/KEY=(POSITION:1,SIZE:40) INFILE.DAT OUTFILE.DAT
```

In the above example, statistics that the SORT command normally displays are redirected to the file STATISTIC.SRT. The ASSIGN command specifies the /USER_MODE qualifier so that when execution of the SORT image is completed, the default equivalence will be reestablished. You can use this technique to suppress the output from any DCL command that displays output data on SYS$OUTPUT.

NOTE

This technique is used in the sample command procedure LISTER.COM in Appendix A.

2.6 DISPLAYING OUTPUT DATA

There are many different ways to display data on your terminal or in the output stream for a batch job during the execution of a command procedure. One method, discussed in Section 3.7, is to use the INQUIRE command. Four other methods, using the TYPE, CREATE, COPY, and WRITE commands are illustrated in Figure 2-3. The first part of the figure shows a file, OUTPUT.COM, created by the CREATE command. The second part of the figure shows the resulting display when the Execute Procedure command (@OUTPUT) is executed from the terminal.

The primary difference between the commands that read data from the input stream (such as TYPE, COPY, and CREATE) and the WRITE command is that the command interpreter does not process input data lines. It does, however, process data in a WRITE command string. Thus, a WRITE command can contain symbol names for data (variable values or character strings) and the symbol names will be replaced with their current values before the line is written.

The next three chapters contain detailed information on creating and using symbols in command procedures. The WRITE command is discussed in Chapter 8.

```
$ CREATE OUTPUT.COM
$ !
$ TYPE SYS$INPUT:
        THESE LINES ARE IN THE INPUT STREAM.
        THE TYPE COMMAND DISPLAYS THEM IN THE OUTPUT STREAM.
$ CREATE SYS$OUTPUT:
        .
        .
        THESE LINES (AS WELL AS ANY BLANK LINES PRECEDING AND
        FOLLOWING) ARE IN THE INPUT STREAM.

        THE CREATE COMMAND CREATES A FILE IN THE OUTPUT STREAM.
        .
        .
$ COPY SYS$INPUT: SYS$OUTPUT
        THE COPY COMMAND COPIES DATA FROM THE INPUT STREAM
        INTO THE OUTPUT STREAM.

        NOTE THAT FOR EACH OF THESE COMMANDS (CREATE AND COPY)
        A LINE BEGINNING WITH A DOLLAR SIGN INDICATES THE END
        OF THE INPUT DATA.

$ WRITE SYS$OUTPUT "THE WRITE COMMAND WRITES A SINGLE DATA LINE."
$ WRITE SYS$OUTPUT "LINES WRITTEN BY THE WRITE COMMAND ARE,"
$ WRITE SYS$OUTPUT "HOWEVER, PROCESSED BY THE COMMAND INTERPRETER."
$ !
^Z
$ @OUTPUT

        THESE LINES ARE IN THE INPUT STREAM.
        THE TYPE COMMAND DISPLAYS THEM IN THE OUTPUT STREAM.
        .
        .
        THESE LINES (AS WELL AS ANY BLANK LINES PRECEDING AND
        FOLLOWING) ARE IN THE INPUT STREAM.

        THE CREATE COMMAND CREATES A FILE IN THE OUTPUT STREAM.
        .
        .
        THE COPY COMMAND COPIES DATA FROM THE INPUT STREAM
        INTO THE OUTPUT STREAM.

        NOTE THAT FOR EACH OF THESE COMMANDS (CREATE AND COPY)
        A LINE BEGINNING WITH A DOLLAR SIGN INDICATES THE END
        OF THE INPUT DATA.

THE WRITE COMMAND WRITES A SINGLE DATA LINE.
LINES WRITTEN BY THE WRITE COMMAND ARE,
HOWEVER, PROCESSED BY THE COMMAND INTERPRETER.
$
```

Figure 2-3 Displaying Data in the Output Stream

CHAPTER 3

USING SYMBOLS IN COMMAND PROCEDURES

A command symbol is a character string name that has a value. In VAX/VMS command procedures, you can define symbols as constants or variables and manipulate them in much the same way that you manipulate variables in a programming language. In fact, the symbolic capabilities of the command interpreter, together with commands such as IF and GOTO, make the DCL command language very much like a programming language.

For example, you can define a symbol to represent a character string as shown below:

```
$ FILE := ALPHA
```

This command, called an assignment statement, gives the symbol name specified on the left (FILE) the value ALPHA. Subsequently, the file ALPHA can be referred to symbolically by referring to the symbol name FILE. For example:

```
$ FORTRAN 'FILE'
```

The apostrophes surrounding the symbol name FILE are substitution operators; they tell the command interpreter that the word they surround is a symbol name. The command interpreter substitutes the value ALPHA for symbol FILE before parsing the FORTRAN command.

This chapter describes the syntax of symbol names and gives examples of defining symbol values. Chapter 4, "Symbol Substitution in Command Procedures," provides detailed information on how the command interpreter substitutes values for symbols during command processing; and Chapter 5, "Using Lexical Functions in Command Procedures," introduces the command language's lexical functions.

3.1 SYMBOL NAMES

An assignment statement equates a symbol name with a character string or arithmetic value. You can use assignment statements in command procedures to perform string substitution and manipulation, arithmetic operations, and logical comparisons.

The format of an assignment statement indicates the data type of the value you are giving a symbol name. The valid data types are character and numeric. The rules for forming a symbol name are:

- Begin a symbol name with an alphabetic letter (A through Z), an underscore (_), or a dollar sign ($). All lowercase letters you enter are translated to uppercase by the command interpreter.
- Use from 1 to 255 characters, including any of the characters listed above.

You can define symbol names and use them as variable data in a command procedure by:

- Equating symbol names to constant values or to other variable symbol names with assignment statements (described in Sections 3.2 through 3.5)
- Passing parameters to a command procedure when you invoke it, or to a batch job when you submit it to a queue (described in Section 3.6)
- Using the INQUIRE command to prompt for a symbol's value during the execution of a command procedure (described in Section 3.7)
- Using the READ command to read a character string from an input file or device and assigning the character string value read to a symbol name (described in Chapter 8)

You can delete symbol names from local and global symbol tables. How to do so is described in Section 3.8.

3.2 EQUATING SYMBOLS TO CHARACTER STRINGS

The format of an assignment statement that equates a character string value to a symbol name is:

```
symbol-name := character-string-value
```

Some examples of assignment statements, as they would appear in a command procedure, are:

```
$ NAME := MYFILE.DAT
$ TEMP := TEMPORARY FILE CREATED
$ EXCLAMATION := "Happy Day!"
$ OUTPUT_MESSAGE := "Beginning..."
$ $GLOBAL := GLOBALNAME
```

Note the use of two dollar sign characters in the last assignment statement above. The first dollar sign is the required character that begins command procedure lines; the second dollar sign is the first character of the symbolic name. Issued interactively, the command string would be:

```
$ $$GLOBAL := GLOBALNAME
```

The additional dollar sign is required in interactive mode because the command interpreter always accepts an optional dollar sign preceding any command string.

3.2.1 Special Characters in Symbol Values

A character string value can contain any alphanumeric or special characters; however, you must enclose it in quotation marks if it contains leading space or tab characters, multiple space or tab characters, lowercase letters, or any characters that are not valid in a symbol name.

When it sçans a command string, the command interpreter deletes all leading and trailing space and tab characters and compresses multiple space or tab characters to a single character. You must place quotation marks around a string containing required spaces or tabs to ensure that these characters will not be removed.

To specify a string that contains literal quotation marks, enclose the entire string in quotation marks and use a double set of quotation marks where you want the literal quotation marks to appear. For example:

```
     $ HELLO := "PATTI SAYS ""HI"""
```

You can continue a symbol assignment on more than one line, for example:

```
     $ LONG_NAME := THIS_IS_A_VERY_LONG_SYMBOL-
     NAME_VALUE_CONTINUED_MORE_THAN_ONE_LINE
```

Another case is a long string:

```
     $ ABC:= THIS THE -
             STRING THAT MUST BE -
             CONTINUED.
```

Note, however, if you are continuing a string that must be enclosed in quotation marks, you must use quotation marks around each portion of the string. For example:

```
     $ ABC := "This is "-
     "the string"
```

However, the resulting value will contain a literal quotation mark, that is, the symbol ABC in this example will have the value:

```
     This is "the string
```

You can specify a null string either by using a double set of quotation marks with no intervening characters or by specifying no string. For example, the following statements are equivalent:

```
     $ NULLSTRING := ""
     $ NULLSTRING :=
```

You can omit a trailing quotation mark on the end of a line. For example, the following assignment statements are equivalent:

```
     $ NAME := "Juniper"
     $ NAME := "Juniper
```

For clarity, however, the trailing quotation mark is recommended.

3.2.2 Replacing Substrings in Character String Symbol Values

A special format of the character string assignment statement allows you to replace data within a defined substring of a value. This format is:

```
     symbol-name[offset,size]:= character-string-value
```

The offset is the position of the substring relative to the first character in the string, and the size is the length of the substring.

The square brackets are required notation, and no spaces are permitted between the right bracket and the colon. You can specify literal numeric values for offset and size, or symbol names equated to numeric values. Literal values are assumed to be decimal. These values can be in the range of 0 through 254.

This type of assignment statement evaluates the current value of symbol-name and then replaces a specified string of characters with the specified character string value. For example:

```
     $ A := ABCDEF
     $ A[0,3]:=DEF
```

The first assignment statement above gives the symbol name A the value ABCDEF. The second assignment statement specifies that the value DEF replaces three characters in the value of A, beginning at an offset of 0 from the beginning of the string. The result is that the value of A becomes DEFDEF.

The symbol name you specify can be undefined initially. The assignment statement creates the symbol name and provides leading or trailing spaces in the symbol value if necessary. For example:

```
     $ B[4,3]:= GHI
```

If the symbol named B does not have a value when this assignment statement is executed, the resulting value of B is "   GHI," that is, B has four leading spaces before the characters GHI. You can use this format to create a blank line of any number of characters, for example:

```
     $ LINE[0,80]:= " "
```

This assignment statement gives the symbol named LINE a value of 80 blank spaces. The following example shows how you can use this syntax of an assignment statement to align data in columns for output:

```
     $ RECORD[0,20]:="Programmer"
     $ RECORD[25,15]:="File Name"
```

These two assignment statements construct a value for the symbol RECORD. The first statement fills in the first 20 columns of the value; the second statement fills in columns 26 through 40. Columns 20 through 24 contain blanks.

NOTE

The sample procedure LISTER.COM in Appendix A illustrates further uses of replacing character strings in assignment statements.

Figure 3-1 illustrates some applications of string substitutions using offsets. In the figure, substitutions change the current value of the symbol FILENAME from its initial assignment, MYFILE.DAT, to TRTEST.DAT;1. Then, the current value of the symbol COMMAND is combined with the string TRTEST.DAT;1 to produce a new assignment for COMMAND.

Interactive Assignment	Resulting Symbol Value	Comments
`$FILENAME:=MYFILE.DAT`	MYFILE.DAT	The result is the initial value of symbol FILENAME
. . .		
`$FILENAME[0,2]:=TR`	TRFILE.DAT	Two characters starting at offset 0 are overlaid
. . .		
`$FILENAME[2,4]:=TESTING.LIS`	TRTEST.DAT	When the string value is longer than the character count the value is truncated to the count
. . .		
`$FILENAME[10,2]:=;1`	TRTEST.DAT;1	When the length of the string is equal to the value of the offset, the string is appended to the current value
. . .		
`$COMMAND:=TYPE`	TYPE	This is the initial value of the symbol command
. . .		
`$COMMAND[5,13]:='FILENAME'`	TYPE TRTEST.DAT;1	Appends a space and the current value of FILENAME to the TYPE command verb.

Figure 3-1 Replacing Character Strings in Assignment Statements

3.3 EQUATING SYMBOLS TO NUMERIC AND LOGICAL EXPRESSIONS

The format of an assignment statement that equates a symbol name to a numeric value or expression is:

```
     symbol-name = expression
```

Some examples are:

```
     $ COUNT = 1
     $ VALUE = %X1C
     $ SUM = 1 + 7 - 4/3 + 10
```

An expression can be any literal numeric value or an arithmetic or logical expression. Literal numeric values are assumed to be decimal. You can specify a value in another radix by using the radix operator (%X for hexadecimal, %D for decimal, or %O for octal) as shown in the second example above. When you define a value in either hexadecimal or octal, the command interpreter converts the value to a decimal integer.

When the command interpreter evaluates an expression, it assigns the expression a value based on the result of the operations specified in the expression:

- If the expression contains logical operators, arithmetic comparison operators, or string comparison operators, the expression is considered true if it results in an odd numeric value; the expression is considered false if it results in an even numeric value.

- If the expression contains arithmetic operators, the result is the value of the arithmetic operations.

The following sections show how to specify expressions using assignment statements. Note that the rules for specifying and using expressions in assignment statements also apply to specifying expressions in the IF command, and in all contexts in which the command interpreter automatically performs expression evaluation. The IF command is described in Chapter 6, "Execution Flow in Command Procedures."

For clarity, the examples in this chapter show literal numeric and character string values in expressions. Additional examples of expressions are shown throughout this manual; these examples will show how to use symbols as variables or constants in expressions.

3.3.1 **Operators**

Table 3-1 lists the valid operators you can use in forming expressions and defines the order of precedence of evaluation. Logical and comparison operators must be preceded by a period (.) with no intervening blanks. The operator must be terminated with a period. You can type any number of blanks or tabs between operators and operands. For example, the following expressions are equivalent:

```
A.EQS.B
A .EQS.  B
```

Each operator (except .NOT. and the unary plus or minus signs) must have operands on each side.

When you specify more than one operation in an expression, the operations are performed in the order of precedence listed in Table 3-1, where 1 is the lowest precedence and 6 is the highest precedence. For example, multiplication is performed before addition. Use parentheses to override the order in which operators are evaluated: expressions within parentheses are evaluated first.

Operations of the same precedence are performed from left to right, as they appear in the command.

Table 3-1
Summary of Operators in Expressions

Type	Operator	Precedence[1]	Operation
Logical Operators	.OR.	1	Logical OR
	.AND.	2	Logical AND
	.NOT.	3	Logical complement
Arithmetic Comparison Operators	.EQ.	4	Arithmetic equal to
	.GE.	4	Arithmetic greater than or equal to
	.GT.	4	Arithmetic greater than
	.LE.	4	Arithmetic less than or equal to
	.LT.	4	Arithmetic less than
	.NE.	4	Arithmetic not equal to
String Comparison Operators	.EQS.	4	String equal to
	.GES.	4	String greater than or equal to
	.GTS.	4	String greater than
	.LES.	4	String less than or equal to
	.LTS.	4	String less than
	.NES.	4	String not equal to
Arithmetic Operators	+	5	Arithmetic sum
	-	5	Arithmetic difference
	*	6	Arithmetic product
	/	6	Arithmetic division (integer quotient)

1. Lowest precedence is 1; highest precedence is 6.

3.3.2 Logical Operations

Use logical operators to perform logical functions on arithmetic values or to construct complicated expressions. Some examples are listed below:

Expression	**Value of Symbol**
A = 3 .OR. 5	A = 7
B = 3 .AND. 5	B = 1
C = .NOT.3	C = -4
D = 3 + 4 .AND. 2 + 4	D = 6

Operands for logical operations must be literal numeric values, symbol names equated to numeric values, or expressions.

Note that logical operators can be used in an arithmetic sense as well. For example:

```
$ A = %X1000 .OR. %X0001
```

This expression performs a logical OR operation on two values. The resulting value of the symbol A is %X1001, or 4097. Note that one of the two values in the OR expression (%X0001) is logically true; the other value (%X1000) is logically false. The resulting value of A (%1001) is logically true. An arithmetic OR always yields a logical as well as an arithmetic result.

3.3.3 Arithmetic Comparisons

Use arithmetic comparison operators to compare numeric values. If the result of an arithmetic comparison is true, the expression has a value of 1; if the result of the comparison is false, the expression has a value of 0. Some examples are listed below:

Expression	Value of Expression
1.LE.2	1 (true)
1.GT.2	0 (false)
1 + 3 .EQ. 2 + 5	0 (false)
"TRUE".EQ.1	1 (true)
"FALSE"	0 (false)

Operands in arithmetic comparisons can be literal numeric values; symbol names equated to numeric values; expressions that yield numeric values; or character strings enclosed in quotation marks that begin with the uppercase letters T, Y, F, or N. (A character string beginning with the uppercase letters T or Y has a value of 1; a character string beginning with the uppercase letters F or N has a value of 0.)

3.3.4 String Comparisons

Use string comparison operators to compare alphanumeric character strings. Character string comparison is based on the binary values of the ASCII characters in the string. The ASCII characters and their hexadecimal values are listed in Table 3-2. The following rules apply to character string comparisons:

- The comparison is on a character-by-character basis: the comparison terminates as soon as two characters do not match.
- If one string is longer than the other, the shorter string is padded on the right with spaces (an ASCII value of %X20) before the comparison is made. Note that a space has a lower numeric value than any of the alphabetic or numeric characters.
- Lowercase letters have higher numeric values than uppercase letters.

If the result of a comparison is true, the expression is given a value of 1; if the comparison is false, the expression is given a value of 0. Some examples are listed below:

Expression	Value of Expression
"MAYBE".LTS."maybe"	1 (true)
"ABCD".LTS."EFG"	1 (true)
"YES".GTS."YESS"	0 (false)
"AAB".GTS. "AAA"	1 (true)

Operands in string comparisons can be literal strings enclosed in quotation marks, symbol names equated to character strings, or literal numeric values. (Literal numeric values are compared using the binary value of their ASCII character string representations.)

If you do not enclose a literal character string in quotation marks, the command interpreter assumes the string is a symbol name and issues an error message if the symbol is not defined.

Table 3-2
ASCII Character Set and Hexadecimal Values

HEX Code	ASCII Char.	HEX Code	ASCII Char.	HEX Code	ASCII Char.	HEX Code	ASCII Char.
00	NUL	20	SP	40	@	60	\
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

3.3.5 Arithmetic Operations

Use arithmetic operators to perform calculations on numeric integer values. In arithmetic operations, all nondecimal values (specified by radix operators) are converted to binary values before the operation is performed. After the operation, the result is converted to decimal. All arithmetic is integer arithmetic; that is, all fractional values are truncated. Some examples are listed below:

Expression	**Result**
A = 5 + 10 / 2	A = 10
B = 5 * 3 - 4 * 6 / 2	B = 3
C = 5 * (6 - 4) - 8 / (2 - 1)	C = 2
D = %X50	D = 80
E = %X10 + 5	E = 21
F = 6 / 4	F = 1

Operands in arithmetic operations can be literal numeric values, symbol names equated to numeric values, or expressions that yield numeric values.

NOTE

The sample procedure CONVERT.COM in Appendix A illustrates arithmetic assignment statements that perform calculations.

3.3.6 **Arithmetic Overlays**

One format of an arithmetic assignment statement can be used to perform binary overlays in the current value of a symbol name. This format is:

```
     $ symbol-name[bit-position,size]= numeric-expression
```

The bit-position is the location relative to bit 0 at which the overlay is to occur, and size is the number of bits to be overlaid. The square brackets are required notation, and no spaces are allowed between the right bracket and the equal sign. The bit-position and size can be either literal numeric values or symbol names equated to numeric values. Literal values are assumed to be decimal.

This type of assignment statement evaluates the current value of the symbol name and then replaces the specified number of bits with the value on the right-hand side of the assignment statement.

This form of an assignment statement can store a maximum of 32 bits at a time. You can use this statement to equate a symbol name to a binary value, for example:

```
     $ BELL[0,32]=%X07
```

This statement gives the symbol named BELL a value equivalent to a hexadecimal 7, the ASCII code for the bell character (CTRL/G) on a terminal.

NOTE

The arithmetic overlay technique is used in the sample procedure WAKEUP.COM in Appendix A. The sample procedure CALC.COM also shows the use of this syntax to give a value to a symbol before the F$CVUI (convert unsigned integer) lexical function is used to extract and convert bit fields within the value.

3.4 CHANGING THE CONTEXT OF A SYMBOL

After a symbol is defined, it can be interpreted as character or numeric data, depending on the context in which it is used:

- It can be used in an arithmetic context, for example, in addition, subtraction, multiplication, or division.
- It can be used as a character string in an expression or it can be concatenated with another string.
- It can be used as a logical value and tested for truth or falsity.

For example, suppose a symbol, COUNT, is assigned the value 4 in an arithmetic assignment statement:

```
$ COUNT = 4
```

Then the value of COUNT can be used in other assignment statements such as the examples below:

`$ TOTAL = COUNT + 1`	An arithmetic assignment statement that adds the value of COUNT to the value 1 and equates the result to the symbol TOTAL, which now equals 5.
`$ SYMBOL := P'COUNT'`	A string assignment statement that appends the character string value of COUNT to the character P. SYMBOL now equals P4.
`$ RESULT=TEMP.OR.COUNT`	A logical OR operation on the symbols TEMP and COUNT. If either value is true the symbol RESULT will have a true value assigned to it.

If you define a null character string value for a symbol, that symbol has a value of 0 when it is used in an arithmetic context. For example:

```
$ A :=
$ B = 2
$ C = A + B
```

After these statements are executed, the symbol C has a value of 2.

3.5 SYMBOL TABLES

The command interpreter maintains symbol names and their associated values in two types of symbol table:

- A local symbol table that contains symbols associated with each active command level
- The global symbol table that contains symbols accessible at all command levels

Symbol tables are of particular importance in the understanding of symbol substitution. Symbol substitution is described in full in Chapter 4.

The following sections describe how to define local and global symbols.

3.5.1 Local Symbols

The command interpreter maintains a symbol table for each active command level. These tables are called local symbol tables, and the symbols they contain can be accessed only from the current command level or from a lower command level. For example symbols defined at command level 0 can be accessed by command level 1, but command level 0 cannot access symbols defined at command level 1.

Use a single equal sign (=) in an assignment statement to define a local symbol. For example:

```
$ COUNT = 1
$ OUTDAT := "Beginning tests...."
```

A local symbol exists as long as the command level at which it was defined remains active, unless the symbol is specifically deleted. For example, if you define the symbol COUNT interactively (at command level 0), any command procedure you subsequently execute (until you log out) can refer to the symbol COUNT and obtain its current value. As another example, the command procedure A.COM contains:

```
$ TOTAL = 1
$ @B
```

The procedure B.COM contains the line:

```
$ NEWTOTAL = TOTAL + 1
```

When B is executed, the symbol name TOTAL is accessible and can be referenced or replaced, because the command level at which TOTAL was defined is still active.

If B.COM defines a value for TOTAL, that definition establishes a new value for TOTAL while B is executed. When execution of B is completed and control returns to procedure A, the value of the symbol TOTAL in A is unchanged.

Local symbols are deleted as soon as the command procedure that defined them exits. In the above example, the symbol NEWTOTAL defined in the procedure B.COM is deleted when execution of B.COM is completed.

In addition to the local symbols that you create, the local symbol table for each command level contains eight special symbols named P1, P2, and so on to P8. These symbols represent values, or parameters, that can be passed to a procedure. The techniques for passing parameters to command procedures are described in Section 3.6.

3.5.2 Global Symbols

In addition to the local symbol tables, the command interpreter maintains a global symbol table. A global symbol exists for the duration of the process, unless specifically deleted, and is recognized at any command level. To define a global symbol, use two equal signs (==) in the assignment statement. For example:

```
$ RESULT == 50
$ FILENAME :== MYFILE.DAT
```

These assignment statements define the global symbols named RESULT and FILENAME.

Global symbols are frequently used to define command synonyms. Normally, you would place all the synonyms in your LOGIN.COM file, so these definitions are available for every terminal session. These synonyms must be defined as global symbols; otherwise, they would be deleted as soon as the procedure LOGIN.COM was executed.

In addition to the global symbols that you create, the global symbol table contains two special symbols whose values are set by the command interpreter. These symbols, named $STATUS and $SEVERITY, contain values indicating the success or failure of the most recently executed image. For information on these symbols and how to use them in command procedures, see Chapter 7, "Controlling Error conditions and CTRL/Y Interrupts."

3.5.3 Order of Search of Symbol Tables

When the command interpreter performs symbol substitution, it searches symbol tables in the following order:

1. The local symbol table for the current command level
2. Local symbol tables for each previous command level, searching backwards from the current level
3. The global symbol table

You can use the SHOW SYMBOL command to display the current value of any symbol. The SHOW SYMBOL command uses the same order of search to locate symbol definitions, that is, it searches the local symbol tables and then the global symbol table to locate a specified symbol name.

3.6 PASSING PARAMETERS TO COMMAND PROCEDURES

When you develop and write command procedures, a primary concern is the ability to act on different data, or parameters, each time you execute the procedure. The command interpreter provides a direct method for specifying, at execution or submission time, values to correspond to symbols within the procedure.

For example, the command procedure named RUNTEST contains the lines:

```
$ ASSIGN 'P1' INFILE
$ ASSIGN 'P2' OUTFILE
$ RUN SORTER
```

The program SORTER.EXE reads a file using the logical name INFILE and writes an output file using the logical name OUTFILE. To assign equivalences to these logical names, values must be provided for P1 and P2 when the procedure is executed.

Note, however, that P1 and P2 are special symbol names; the command interpreter defines eight of these special symbols for use as parameters within command procedures. These local symbols are named P1, P2, and so on to P8. The command interpreter gives them null values by default if you do not specify values for them.

Unspecified parameters used in character string contexts are treated as null strings; in arithmetic contexts, they are given a value of 0. For example, a procedure called ADD.COM can contain the lines:

```
$ TOTAL = P1 + P2 + P3 + P4 + P5 + P6 + P7 + P8
$ SHOW SYMBOL TOTAL
```

When this procedure is invoked, it can be invoked with up to eight numeric values specified for parameters. All unspecified parameters default to 0, which does not affect the result of the addition.

3.6.1 Specifying Parameters for the Execute Procedure Command

When you execute a procedure with the Execute Procedure (@) command, you enter the values for P1, P2, and so on, as command parameters, as follows:

```
$ @RUNTEST INSORT.DAT OUTSORT.DAT
```

This command string gives the symbol named P1 a value of INSORT.DAT and the symbol P2 a value of OUTSORT.DAT. The values for the parameters are assigned according to the order in which you specify them, that is, the first parameter you enter is P1, the second is P2. In this example, P3 through P8 are equated to null strings because no values are specified for them.

You can equate any parameter to a null string by using a set of double quotation marks as a place holder in the command string. For example:

```
$ @RUNTEST "" OUTSORT.DAT
```

This command string sets the parameter P1 to a null string and gives P2 a value of OUTSORT.DAT.

3.6.2 Delimiting Parameters

When you specify parameters for a command procedure that you execute with the Execute Procedure command, spaces in the command string delimit parameters. For example, the following command passes the three parameters, A, B, and C, to the procedure TESTFILE.COM:

```
$ @TESTFILE A B C
```

To pass a parameter that contains lowercase letters, special characters, or embedded blanks, enclose the entire parameter in quotation marks. For example:

```
$ @TESTFILE "lowercase parameter"
```

When the procedure TESTFILE.COM is executed, the parameter P1 is equated to the string:

```
lowercase parameter
```

When you specify parameters, you can specify a string with embedded quotation marks. In this case, the quotation marks are preserved in the string. For example:

```
$ @TESTFILE abc"def"ghi
```

In this example, the parameter P1 is equated to the string:

```
ABC"def"GHI
```

The characters that are not within quotation marks are converted to uppercase, but the string in quotation marks, including the quotation marks, is left intact.

3.6.3 Passing Parameters to Batch Jobs

To pass parameters to a batch job with the SUBMIT command, use the /PARAMETERS qualifiers, as follows:

```
$ SUBMIT TESTFILE/PARAMETERS=AVERAGE
```

This SUBMIT command passes the string AVERAGE as the parameter P1 for the procedure TESTFILE.COM.

Commas delimit parameters within a list for the SUBMIT command. When you specify more than one parameter, separate them with commas and enclose them in parentheses as in the following example.

```
$ SUBMIT RUNTEST/PARAMETERS=(INSORT.DAT,OUTSORT.DAT)
```

Within the parameter list for the SUBMIT command, you can equate a parameter to a null string by using a set of double quotation marks as a place holder. For example:

```
$ SUBMIT RUNTEST/PARAMETERS=("",OUTSORT.DAT)
```

This SUBMIT command equates the parameter P1 to a null string and gives P2 a value of OUTSORT.DAT.

Note that you can submit more than one file for batch execution in a single SUBMIT command. If you specify parameters with the /PARAMETERS qualifier when you submit a list of command procedures, however, the parameters you specify are equated to P1, P2, and so on in each file you specify. For example:

```
$ SUBMIT TESTA,TESTB/PARAMETER=10
```

When the procedure TESTA is executed in the batch job, the symbol named P1 has the value of 10. When execution of TESTA is completed the job executes TESTB; in TESTB also, the symbol P1 has the value of 10, unless the procedure TESTA redefines the value of P1.

You can also use the /PARAMETERS qualifier on a JOB command when you submit batch jobs through the system card reader. The syntax for specifying parameters on the JOB card is identical with the syntax of specification on the SUBMIT command. For example, a batch job could refer to symbols P1 and P2. When you place cards in the card reader, the JOB card could be continued onto a card that specified different values for these parameters for different runs of the procedure. The JOB command might appear as shown on the two cards illustrated in Figure 3-2.

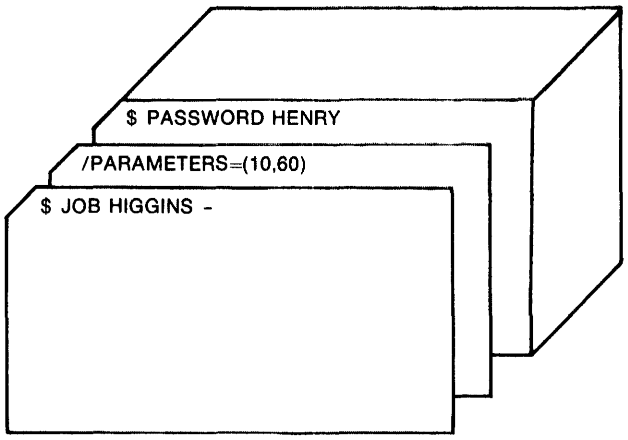


Figure 3-2 Using a /PARAMETERS Qualifier Card

3.6.4 Redefining Parameters

The symbol names P1 through P8, although defined by default if not specified, are not reserved to the command interpreter. You can, in your command procedures, define values for these symbol names or redefine them, as needed. For example:

```
     $ P1 = P1 - 1
```

This assignment statement assumes that P1 has a numeric value and decreases the current value by one.

Another example is:

```
     $ IF P1 .EQS. "" THEN INQUIRE P1 "Input file name"
```

This command checks whether a value was specified for P1; if not, the INQUIRE command requests interactive assignment of a value for P1. The IF command is described in Chapter 6. The INQUIRE command is described in the following section.

3.7 THE INQUIRE COMMAND

When you execute a command procedure interactively, you can use the INQUIRE command to define a value for a symbol while the command procedure is executed. When the INQUIRE command is executed from the command procedure, the command interpreter issues a prompting message to SYS$COMMAND, that is, the terminal; the text of the message is taken from the INQUIRE command line, as shown in the example below.

The procedure RUNTEST contains the lines:

```
     $ INQUIRE IN INPUT FILE
     $ INQUIRE OUT OUTPUT FILE
     $ ASSIGN 'IN' INFILE
     $ ASSIGN 'OUT' OUTFILE
     $ RUN SORTER
```

When you execute this procedure, the terminal interaction might appear as follows (if SET NOVERIFY is in effect):

```
$ @RUNTEST(RET)
  INPUT FILE: DB1:INSORT.DAT(RET)
  OUTPUT FILE: DB2:OUTSORT.DAT(RET)
$
```

When these INQUIRE commands are executed, the prompting messages INPUT FILE: and OUTPUT FILE: are displayed, and you must enter values for the symbols IN and OUT before the command procedure continues. The prompt strings INPUT FILE and OUTPUT FILE are optional parameters for the INQUIRE command; if you do not specify them, the command uses the symbol names IN and OUT to prompt for values.

By default, INQUIRE command appends a colon (:) and a space to the prompt string; you do not need to include them when you specify the prompt string to the INQUIRE command. If you do not want the colon and space to be appended to the prompt string, use the NOPUNCTUATION qualifier, as described in the VAX/VMS Command Language User's Guide. Note that you can request a lowercase prompting message or a prompting message that contains special characters by enclosing the prompt string in quotation marks, as shown below:

```
$ INQUIRE FILE "Enter name of file to edit"
```

When this INQUIRE command is executed, the following prompting message is displayed on SYS$COMMAND (normally, the terminal):

```
Enter name of file to edit:
```

The symbol name FILE and the value you enter in response to this prompt are placed in the local symbol table for the current command level.

The INQUIRE command also accepts entries for the global symbol table. To define a global symbol name with the INQUIRE command, use the /GLOBAL qualifier. For example:

```
$ INQUIRE/GLOBAL FILE "Enter name of file to edit"
```

When you respond to this prompt, the symbol name FILE is entered in the global symbol table with whatever value you enter.

When you do not enter any data in response to an INQUIRE command, the specified symbol name is given a null value. For example:

```
$ INQUIRE FILE "File"
$ IF FILE .EQS. "" THEN EXIT
```

In the above example, the INQUIRE command is followed by a test to determine whether a value was entered. If not, the procedure exits.

NOTE

The sample procedures EDITALL.COM and CALC.COM in Appendix A illustrate the use of this technique.

3.8 DELETING SYMBOLS

The DELETE/SYMBOL command deletes symbols. You can delete one or all local symbols from the local symbol table for the current command level or one or all global symbols from the global symbol table. For example, the following command deletes the local symbol named TOTAL:

```
$ DELETE/SYMBOL TOTAL
```

The /GLOBAL qualifier indicates that a global symbol is to be deleted. For example:

```
$ DELETE/SYMBOL/GLOBAL/ALL
```

This command deletes all global symbols.

Because the command interpreter automatically deletes local symbol tables when a command procedure exits, you do not normally need to delete symbols. However, if you have defined many global symbols for command synonym definitions or if you execute a procedure that requires many local symbols or many symbols with character string values, you may run out of symbol table space.

The maximum number of symbols that can be defined at any one time depends on:

- The amount of space available to the command interpreter to contain local and global symbol tables and labels for the current process. This space is approximately 12 pages (6144 bytes) for each process.
- The size of the symbol names and their values. The command interpreter incurs 14 bytes of overhead for every symbol definition and it allocates space for symbol definitions in 8-byte increments. For example, 24 bytes are required to maintain the average symbol name and its value, which together would consist of from 5 to 12 characters (bytes).

When the command interpreter runs out of space, it issues the following warning message:

```
%DCL-W-SYMOVF, no room for symbol definitions
```

Then, it takes whatever action is currently defined for warning conditions.

If a command procedure that you are developing exhausts symbol table space, try to recreate the procedure using nesting (as described in Section 6.3), so that inactive symbols will be deleted when the procedure that defines them exits. Or, you can delete all global symbol table definitions for command synonyms before you execute the command procedure (as described above).

CHAPTER 4

SYMBOL SUBSTITUTION IN COMMAND PROCEDURES

While it processes a command string, the command interpreter performs symbol substitution by replacing symbol names in the command string with their current values. To use symbols in commands and command procedures, you will need to understand the mechanics of symbol substitution discussed in this chapter:

- How the command interpreter handles command synonyms (Section 4.1)
- How to use the substitution operators, the apostrophe (') and ampersand (&) characters (Sections 4.2 and 4.4)
- When the command interpreter performs automatic substitution (Section 4.3)
- How to use repetitive and recursive substitution (Section 4.5)
- What happens to symbols that remain undefined during command-interpreter processing (Section 4.6)
- How to verify that symbol substitution takes place (Section 4.7)

4.1 COMMAND SYNONYM SUBSTITUTION

When the command interpreter processes a command string, it examines the first token in the command string to determine whether it is a symbol name. A token is nonblank character string that is terminated with a blank or a special character -- a special character, in this context, is any character that is not valid in a symbol name.

If the token represents a defined symbol, the command interpreter replaces the symbol name with its current value. Then, it executes the command string.

For example, the following assignment statement defines the symbol PDEL as a command synonym:

```
$ PDEL := DELETE SYS$PRINT/ENTRY=
```

Then PDEL is used as the first token in a command string:

```
$ PDEL 181
```

The command interpreter replaces PDEL with its current value and executes the command string:

```
     DELETE SYS$PRINT/ENTRY=181
```

In this example, the command synonym PDEL is delimited with a blank character. Note that depending on the command, other characters can serve as delimiters. In the following example, the left parenthesis properly delimits the symbol name PDEL because the parentheses are valid delimiters:

```
     $ PDEL(181,182,183)
```

This command deletes three files in the queue SYS$PRINT.

4.2 USING APOSTROPHES AS SUBSTITUTION OPERATORS

You must use an apostrophe (') to request the command interpreter to replace a symbol name with its current value when you use a symbol name in place of a command parameter or qualifier. For example:

```
    $ TYPE 'FILENAME'
```

In this example, the string FILENAME is a symbol name used as a parameter for the TYPE command; the apostrophes surrounding the string indicate to the command interpreter that FILENAME is a symbol name and not a literal string.

When you want to assign the value of one symbol to another symbol, you must also use an apostrophe on the right-hand side of a string assignment statement:

```
     $ OLDSTRING := 'FILENAME'
```

Otherwise, the command interpreter would equate the symbol named OLDSTRING to the literal string FILENAME, rather than using the current value of FILENAME.

4.2.1 Substitution Within Strings Enclosed in Quotation Marks

Within character strings enclosed in quotation marks, you can request symbol substitution by preceding a symbol name with two apostrophes. For example:

```
     $ PROMPT_STRING := "Creating file ''FILENAME'.TST"
```

If the current value of the symbol named FILENAME is WIDGET, the symbol name PROMPT_STRING is given the value:

```
     Creating file WIDGET.TST
```

Only a single apostrophe is required to delimit the end of the symbol name, as the above example illustrates.

Note that you can use this construct to maintain a symbol's value in lowercase characters when you equate the current value of the symbol to another symbol, as shown below:

```
     $ A := "this is the line"
     $ B := 'A'
```

In this example, A is given a value containing lowercase letters; the quotation marks are not part of the value. To give a value to the symbol B, the command interpreter replaces the symbol name A with its current value; however, as it scans this line, the command interpreter also converts lowercase data to uppercase. Thus, B has the value:

```
     THIS IS THE LINE
```

To retain this character string as it was initially defined, define B as follows:

```
     $ B := "''A'"
```

In this case, the command interpreter replaces the symbol name A with its current value; then, as it continues scanning the line, the quotation marks ensure that the character string is not converted to uppercase.

4.2.2 Concatenating Symbol Values

You can concatenate two or more symbol names in a command string as shown in the following example:

```
     $ NAME := MYFILE
     $ TYPE := .TST
     $ PRINT 'NAME''TYPE'
```

If this example is executed, the PRINT command queues a copy of MYFILE.TST.

Note that you must properly delimit symbol names by placing apostrophes around each symbol name in the command string. Note also that no blanks are included in the string 'NAME''TYPE'. When these symbols are concatenated, the resulting value cannot have any embedded blanks; thus, no blanks can occur between the symbol names.

4.3 AUTOMATIC SUBSTITUTION

The command interpreter assumes, in certain contexts, that a string of characters beginning with an alphabetic character is a symbol name. In these contexts, substitution is automatic and you need not delimit symbol names with apostrophes. In fact, if you use apostrophes, the results are quite different because recursive substitution will occur (see Section 4.5, "Repetitive and Recursive Substitution").

Symbol substitution is automatically performed in:

- Arithmetic assignment statements
- Tokens enclosed in brackets on the left-hand side of assignment statements
- Lexical function processing (see Chapter 5, "Using Lexical Functions in Command Procedures")
- IF commands (see Chapter 6, "Execution Flow in Command Procedures")

- WRITE commands (see Chapter 8, "Creating, Reading, and Writing Files")
- The DEPOSIT and EXAMINE commands (The DEPOSIT and EXAMINE commands provide an interactive debugging capability at the command level; for descriptions of these commands, see the VAX/VMS Command Language User's Guide)

It is important to note that, in any of these contexts, the command interpreter assumes that any string of characters beginning with an alphabetic letter is a symbol name and that any string of characters beginning with an arabic numeral or with the radix operator (%) is a literal numeric value.

For example, when you use an arithmetic assignment statement, the expression on the right-hand side of the statement is evaluated automatically. For example:

```
$ TOTAL = COUNT + 1
```

No apostrophes are needed to request substitution for the symbol COUNT in this arithmetic assignment statement because the command interpreter automatically substitutes values for symbols as it executes arithmetic assignments.

Similarly, in an IF command:

```
$ IF A .EQ. B THEN GOTO NEXT
```

In the above example, the IF command assumes that both A and B are symbol names and uses their current values to test their equality. No apostrophes are necessary.

4.4 USING AMPERSANDS AS SUBSTITUTION OPERATORS

In addition to the normal substitution operator, the apostrophe (') described above, the command interpreter recognizes a special operator, the ampersand (&). In many usages, the two operators are functionally equivalent. For example, the following two commands would have the same result if the string FILENAME is currently equated to a character string value:

```
$ TYPE 'FILENAME'
$ TYPE &FILENAME
```

The difference between these two commands is that in the first command, the command interpreter replaces the string FILENAME with its current value while it is scanning the command input, and in the second command, the command interpreter replaces the string FILENAME with its current value while it is analyzing, or parsing, the command.

The following examples show how the results can vary depending on whether you use an apostrophe (') or an ampersand (&) as the substitution operator:

```
$ B := MYFILE.DAT
$ A := 'B'
$ B := NEWFILE.TMP
$ TYPE 'A'
```

In this example, the first assignment statement equates the value MYFILE.DAT to the symbol name B. Then, as the second assignment statement is scanned, the command interpreter substitutes the current value of B (MYFILE.DAT) for the symbol name A. The third assignment statement redefines the symbol name B, which takes the value NEWFILE.TMP. The symbol name A, however, is still equated to the value MYFILE.DAT, so the TYPE command, when executed, displays the file MYFILE.DAT.

In the next example, however, substitution occurs differently:

```
$ B := MYFILE.DAT
$ A := &B
$ B := NEWFILE.TMP
$ TYPE 'A'
```

In this example, the second assignment statement equates the symbol name A to the current value of B (MYFILE.DAT) as the line is scanned, but substitution is not made. Thus, when the current value of B is redefined in the third assignment statement, the new current value of B (NEWFILE.TMP) is equated to A. The TYPE command, when executed, displays the file NEWFILE.TMP.

The use of an ampersand (&) as a substitution operator is syntactically similar to the use of an apostrophe ('), with the following exceptions:

- You cannot use an ampersand within a character string; that is, an ampersand must follow a delimiter (any blank or special character).
- You cannot use ampersands to request substitution within character strings enclosed in quotation marks.
- You cannot use ampersands to concatenate two or more symbol names.
- You cannot terminate a symbol name with an ampersand.

Ampersands are most effective as substitution operators when they are used with apostrophes to provide recursive substitution, as described in the next section.

4.5 REPETITIVE AND RECURSIVE SUBSTITUTION

Substitution is either repetitive or recursive when substitution for a symbol or token in a command string occurs more than once during the processing of a single command string. Specifically, repetitive substitution results when more than one type of substitution occurs in a single command string. Recursive substitution occurs when the command interpreter examines the value substituted to see if the value itself is a symbol. This happens automatically when you use an apostrophe as a substitution operator.

By understanding the order in which the command interpreter performs different types of symbol substitution, you can control how substitution occurs in your command procedures.

4.5.1 Steps in Symbol Substitution

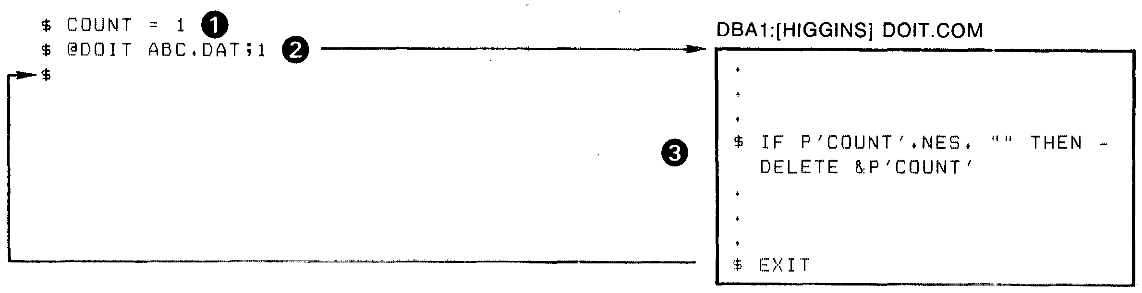
The command interpreter performs symbol substitution in three phases of command processing. These phases are:

1. Command input scanning. During this phase, also called the lexical input phase, the command interpreter reads the command input and replaces all tokens that are preceded with apostrophes (or double apostrophes, for strings within quotation marks).

2. Command parsing. During this phase, the command interpreter analyzes the command string; it (1) determines whether the first token is a command synonym and, if it is, replaces it with its current value and (2) performs all substitution requested with ampersands.

3. Expression evaluation. During this phase, symbols are replaced by the command interpreter during the actual execution of a command, for example, the IF command. This substitution is not, by default, recursive.

Figure 4-1 illustrates a command procedure, DOIT.COM, that contains a command string on which substitution is performed three times, each time during a different phase of command processing.



```
$ COUNT = 1 ❶
$ @DOIT ABC.DAT;1 ❷
$
```

DBA1:[HIGGINS] DOIT.COM

```
.
.
.
$ IF P'COUNT'.NES. "" THEN -
  DELETE &P'COUNT'
.
.
.
$ EXIT
```

❸

❶ The symbol name count is assigned the value 1.

❷ The command procedure DOIT.COM is invoked; and is passed a parameter, the file ABC.DAT;1.

❸ The IF command is processed by the command interpreter in three phases.

First, command input scanning: all substitution requested by the use of apostrophes is performed; the result is:

```
IF P1 .NES. "" THEN DELETE &P1
```

Second, command parsing: all substitution requested by the use of ampersands is performed; the result is:

```
IF P1 .NES. "" THEN DELETE ABC.DAT;1
```

Third, command execution: all character strings used as expressions are evaluated and substitution is performed on these strings. The command line executed is:

```
IF ABC.DAT;1 .NES. "" THEN DELETE ABC.DAT;1
```

Figure 4-1 Example of the Three Phases of Symbol Substitution

Note that the command interpreter does not scan, and therefore does not perform substitution on, any lines in a command procedure that are read as input data by commands or programs executed within the procedure. .For example:

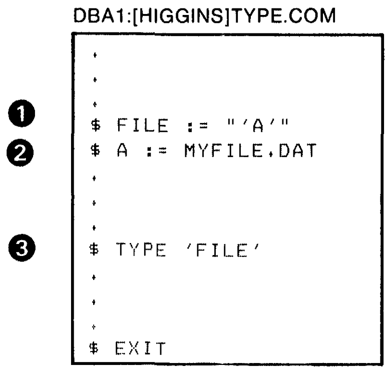
```
$ RUN AVERAGE
55
55
9999
```

The program AVERAGE reads from SYS$INPUT, that is, the command input stream. The literal data lines 55, 55, and 9999 are never read by the command interpreter. Thus, in this context you cannot use symbol names.

4.5.2 Recursive Substitution Using Apostrophes

When you use an apostrophe to request symbol substitution, the command interpreter performs recursive substitution from left to right in the command string. This means that for each token on the line, the string resulting from the substitution is scanned again to determine whether the string contains any apostrophes. If there are apostrophes, the command interpreter performs substitution and again examines the resulting string for apostrophes.

Figure 4-2 illustrates a simple case of recursive substitution. Note that the command interpreter repeats the substitution as many times as necessary to complete the substitution of a value for the token; there is no practical limit to the layers of symbol definition.



DBA1:[HIGGINS]TYPE.COM

```
  .
  .
  .
❶ $ FILE := "'A'"
❷ $ A := MYFILE.DAT
  .
  .
  .
❸ $ TYPE 'FILE'
  .
  .
  .
  $ EXIT
```

❶ The symbol name FILE is equated to the value 'A' while the quotation marks prevent the command interpreter from substituting a value for 'A' — in fact, there is no value for A yet defined.

❷ A is equated to the file MYFILE.DAT

❸ When the command interpreter scans this TYPE command, it substitutes the current value of 'FILE', resulting in

```
TYPE 'A'
```

Since the current value contains apostrophes, the command interpreter scans the line again (substitution recurs) and substitutes the value of A . The command string executed is:

```
TYPE MYFILE.DAT
```

Figure 4-2 Example of Recursive Substitution

Substitution using apostrophes is not, however, recursive when values are substituted for symbols within strings that are enclosed in quotation marks. For example:

```
$ SYMBOL := NAME
$ A := "'SYMBOL'"
$ B := 'A'
```

After the last assignment statement in this example is executed, the resulting value of the symbol B is NAME. This result is achieved in the following steps:

- The symbol name A is replaced with its current value:

  'SYMBOL'

- Because this value has apostrophes in it, the command interpreter replaces the value SYMBOL with its current value:

  NAME

- Because this value has no apostrophes, it is the final value given to the symbol name B.

Note, however, what happens if you define B as follows:

```
$ B := "''A'"
```

In this case, B has the value 'SYMBOL'. The symbol name A is replaced only once because substitution is never recursive within character strings enclosed in quotation marks.

4.5.3 Recursive Substitution Using Command Synonyms

The command interpreter performs recursive substitution automatically only when an apostrophe is in the command string. In some cases, you may want to nest command synonym definitions, as the following lines suggest:

```
$ COMMAND := "TYPE A.B"
  .
  .
  .
$ EXEC := "'COMMAND'"
  .
  .
  .
$ EXEC
```

In this example, when the command synonym EXEC is processed, the command interpreter performs substitution only once. The resulting string is 'COMMAND'; the command interpreter issues an error message because it cannot detect a command on the line. To correctly use the command synonym EXEC, you must precede it with an apostrophe, as shown below:

```
$ 'EXEC
```

Figure 4-3 shows another example of using an apostrophe with a command synonym to force recursive substitution. The example shows the results of substitution first, without using an apostrophe and then, the results of substitution when an apostrophe is specified.

NOTE

The procedure in Figure 4-3 is similar to the GETPARMS.COM procedure in Appendix A.

```
.
.
.
$ GETPARMS:=="@GETPARMS 'P1' 'P2' 'P3' 'P4' 'P5' 'P5' 'P6' 'P7' 'P8' " ❶
$ @TESTPARM A B C D ❷
  P8 = 'P8'  ─┐
  P7 = 'P7'   │
  P6 = 'P6'   │
  P5 = 'P5'   ├ ❺
  P4 = 'P4'   │
  P3 = 'P3'   │
  P2 = 'P2'   │
  P1 = 'P1'  ─┘
  P8 =       ─┐
  P7 =        │
  P6 =        │
  P5 =        ├ ❽
  P4 = D      │
  P3 = C      │
  P2 = B      │
  P1 = A     ─┘
$
```

DBA1:[HIGGINS]TESTPARM.COM

```
❸  $ GETPARMS
❻  $ 'GETPARMS
   $ EXIT
```

DBA1:[HIGGINS]GETPARMS.COM

```
❹ ❼  $ SHOW SYMBOLS/ALL
     $ EXIT
```

❶ Global symbol GETPARMS is defined. Quotation marks prevent the command interpreter from substituting values for P1-P8 when it assigns a value to GETPARMS.

❷ TESTPARM.COM is invoked, and passed four parameters.

❸ The command interpreter substitutes the current value of GETPARMS during command parsing because it is the first token in a command string. The command synonym is executed; there is no recursion.

❹ GETPARMS.COM is invoked. The parameters passed to it have not been replaced because these symbol names contain apostrophes. The result of the SHOW SYMBOL command is displayed at ❺.

❻ TESTPARM.COM resumes execution here. Because an apostrophe precedes the command string, the command interpreter is forced to perform recursive substitution before it executes the command.

❼ GETPARMS.COM is invoked. The parameters are substituted and the results displayed at ❽.

Figure 4-3 Recursive Substitution Using a Command Synonym

4.5.4 Recursive Substitution Using Ampersands

An ampersand as a substitution operator is most effective when you want substitution to occur from right to left on a token. A common use was shown in Figure 4-1: to use the same command string to process multiple parameters (or symbol names assigned incremental values), you must use an ampersand so that recursive substitution occurs in the correct order. For example, the command string in Figure 4-1 uses the following syntax:

```
$ DELETE &P'COUNT'
```

This causes the command interpreter to first replace the symbol COUNT with its value and then, during parsing, to replace the symbol P1 with its value. Note what would happen if the token were specified as follows:

```
$ DELETE 'P''COUNT'
```

In this example, the command interpreter, during initial scanning of the command, would perform normal left-to-right substitution. It would attempt to replace the separate tokens P and COUNT. Because P is not a defined symbol, only COUNT would be replaced. The DELETE command string would be:

```
DELETE 1
```

The action the command interpreter takes when a symbol is undefined depends on the context of the command. For additional details, see Section 4.6, "Undefined Symbols."

For an example of using recursive substitution to process parameters passed to a command procedure, see the procedure in Section 6.1.2, "Using Symbols in IF Commands."

4.5.5 Recursive Substitution in Expressions

When the command interpreter analyzes an expression in a command, any symbols specified in the expression are replaced only once; recursion is not automatic. You can, however, force recursion by using an apostrophe or an ampersand in the expression. When you design a procedure to force recursion in this way, however, you must remember:

- The command interpreter performs all substitution requested by apostrophes and ampersands before the command string is executed.
- Commands that automatically perform symbol substitution do so after the command string has been processed by the command interpreter.

The following example illustrates recursive substitution in an IF command:

```
$ IF P'COUNT' .EQS. "" THEN GOTO END
```

When the command interpreter scans this input line, it replaces the symbol name COUNT with its current value. If the current value of COUNT is 1, the command string, after scanning, is:

```
IF P1 .EQS. "" THEN GOTO END
```

Because this string has no apostrophes, the command interpreter does not perform any more substitution: however, when the IF command executes, it automatically evaluates the symbol name P1 and replaces it with its current value.

Note, however, that if the token resulting from substitution by the command interpreter is not a valid symbol name, the command will fail because a symbol is undefined. For example:

```
$ FILENAME :=A.B
$ IF 'FILENAME' .NES. "" THEN TYPE 'FILENAME'
```

When the command interpreter processes this command string, it replaces the symbol name FILENAME with its current value. After substitution, the command string is:

```
IF A.B .NES. "" THEN TYPE A.B
```

Because A.B is not a valid symbol, an error occurs. For this IF command to be correctly processed, you must omit the apostrophes, as shown below:

```
$ IF FILENAME .NES. "" THEN TYPE 'FILENAME'
```

Apostrophes are required in the TYPE command string because the command interpreter does not automatically replace symbols in TYPE commands.

NOTE

For an example of using an apostrophe in an arithmetic assignment statement to force recursion, see the sample procedure CALC.COM in Appendix A.

4.6 UNDEFINED SYMBOLS

If a symbol is not defined when it is used in a command string, the command interpreter either issues an error message or replaces the symbol with a null string or a 0, depending on the context. The rules are:

- During command input scanning and during command parsing, the command interpreter replaces all undefined symbols that are preceded with apostrophes or ampersands with null strings or zeros.
- During expression evaluation, the command interpreter issues a warning message and does not complete command processing.

These rules are most easily illustrated by comparing string assignment statements with arithmetic assignments statements. In a string assignment statement, the value on the right-hand side is assumed to be literal character data. You must use an apostrophe to request substitution to occur before a symbol name is assigned a value. For example:

```
$ TYPE := .TST
$ FILE := MYFILE'TYPE'
$ PRINT 'FILE'
```

In this example, the symbol name is replaced with its current value while the command input is read; the assignment statement gives the symbol FILE a value of MYFILE.TST. If a symbol name does not have a value, the command interpreter, by default, replaces the symbol name in the command string with a null string. In the above example, if TYPE is not defined, the command interpreter gives the symbol FILE a value of MYFILE.

Note that, within the context of character strings, a null string can be a meaningful construct. In the above example, the absence of a file type in the file specification for the PRINT command causes the PRINT command to use the default file type of LIS.

In an arithmetic assignment, however, the value of the right-hand side is evaluated as an expression, which must have a value. For logical and comparison operations, the resulting value is either a 0 or a 1; for arithmetic operations, the resulting value is always arithmetic. For example:

```
     $ A = 1
     $ B = 2
     $ C = A + B
```

In this case, the symbols A and B must have values or the expression that assigns a value to C is meaningless. If A or B is not defined, the command interpreter issues a warning message and does not give a value to C. Note that if either A or B is defined as a null string, the command interpreter assumes it has a value of 0; then, the expression is valid.

4.7 VERIFICATION OF SYMBOL SUBSTITUTION

The SET VERIFY and SET NOVERIFY commands control whether the command interpreter displays lines in a command procedure as it executes them. When verification is in effect, the command interpreter displays each command line after it has completed initial scanning and before the command is parsed and executed. Thus, you see displayed the results of symbol substitution performed during scanning, but not the results of symbol substitution performed during command parsing and execution. For example:

```
     $ SET VERIFY
     $ COUNT = 1
     $ IF P'COUNT' .NES. "" THEN GOTO &P'COUNT'
```

When this procedure is executed interactively, the following lines are displayed on the terminal:

```
     $ COUNT = 1
     $ IF P1 .NES. "" THEN GOTO &P1
```

The SET VERIFY command is not displayed unless verification is already in effect.

CHAPTER 5

USING LEXICAL FUNCTIONS IN COMMAND PROCEDURES

The command language includes constructs, called lexical functions, that return information about the current process and about character strings. The functions are called lexical functions because the command interpreter evaluates them during the command input scanning (or lexical processing) phase.

You can use lexical functions in any context in which you normally use symbols, expressions, or literal values. In command procedures, you can use lexical functions to translate logical names, perform character string manipulations, and determine the current processing mode of the procedure.

5.1 THE FORMAT OF LEXICAL FUNCTIONS

The general format of a lexical function is:

```
     'F$function-name([args,...])'
```

'F$

Indicates that what follows is a lexical function. The substitution operator (') is required so that the command interpreter will evaluate the function during command input scanning. Otherwise the command interpreter will assume that the function name is a user-defined symbol name (and not a lexical function). Note that the command interpreter evaluates lexical functions in comment lines during input scanning; in forward GOTO commands the command interpreter evaluates lexical functions while it searches for the specific label.

function-name

Specifies the function to be evaluated. All function names are keywords. You can truncate function names to any unique truncation.

( )

Enclose function arguments, if any. The parentheses are required for all functions, including functions that do not accept any arguments.

args,...

Specify arguments for the function, if any.

When you use a lexical function in any context in which symbol substitution automatically occurs, the substitution is recursive.

All arguments specified for lexical functions that begin with alphabetic characters are assumed to be symbol names; therefore, substitution is automatic.

You can specify arguments using literal numeric or character string data or symbols. If a symbol is undefined, the command interpreter replaces it with a null string.

You cannot specify one lexical function as an argument for another lexical function, nor can you use double apostrophes to request substitution of a symbol value within a quoted string that is an argument for a lexical function.

Table 5-1 summarizes the lexical functions, their formats, and the information returned by each. The remainder of this chapter describes lexical functions in more detail and gives examples of their use.

Table 5-1
Summary of Lexical Functions

Function	Value Returned
F$CVSI(bit-position,width,integer)	Signed value extracted from the specified integer, converted to an ASCII literal
F$CVUI(bit-position,width,integer)	Unsigned value extracted from the specified integer, converted to an ASCII literal
F$DIRECTORY()	Current default directory name string, including brackets
F$EXTRACT(offset,length,string)	Substring beginning at specified offset for specified length of indicated string
F$LENGTH(string)	Length of specified string
F$LOCATE(substring,string)	Relative offset of specified substring within string indicated; or, the length of the string if the substring is not found
F$LOGICAL(logical-name)	Equivalence name of specified logical name (first match found in ordered search of process, group, and system logical name tables); or, a null string if no match is found

(continued on next page)

Table 5-1 (Cont.)
Summary of Lexical Functions

Function	Value Returned
F$MESSAGE(code)	Message text associated with the specified numeric status code value
F$MODE()	One of the character strings INTERACTIVE or BATCH
F$PROCESS()	Current process name string
F$TIME()	Current date and time of day, in the format dd-mmm-yyyy hh:mm:ss.cc
F$USER()	Current user identification code (UIC), in the format [g,m]
F$VERIFY(value)	If no argument is used: a numeric value of 1 if verification is set on; a numeric value of 0 if verification is set off If an argument is used: the same value as F$VERIFY (); in addition, the state of the argument's low-order bit turns verification on (if state is 1) or off (if state is 0)

5.2 INFORMATIONAL FUNCTIONS

The command language provides the following informational functions:

- F$MODE returns a character string that shows the mode in which the process is currently executing. That is, F$MODE returns the string "INTERACTIVE" or "BATCH".
- F$VERIFY returns a numeric value indicating whether the verification setting is currently on or off, and may turn verification on or off.
- F$DIRECTORY returns the current default directory name string
- F$PROCESS returns the character string name of the process
- F$USER returns the current user identification code (UIC) of the process
- F$LOGICAL returns the equivalence name string of a specified logical name.

- F$TIME returns the current date and time
- F$MESSAGE returns a character string representing the message text associated with a specific system status value

Each of these functions is described in greater detail below.

5.2.1 The F$MODE Lexical Function

The F$MODE function is useful in command procedures that must act differently when executed in batch mode than when executed in interactive mode. The F$MODE function has no arguments.

For example, a line in a command procedure can use the F$MODE function to test whether the procedure is being executed during an interactive terminal session or within a batch job:

```
$ IF "''F$MODE()'" .EQS. "BATCH" THEN GOTO BATDEF
$ INTDEF:
    .
    .
    .
$ EXIT
$ BATDEF:
    .
    .
    .
```

The IF command in the above example compares the character string returned by F$MODE with the character string BATCH; if they are equal, control branches to the label BATDEF. Otherwise, the statements following the label INTDEF are executed and the procedure exits before the statements at BATDEF. In other words, this procedure has two sets of initialization commands: one for interactive mode and one for batch mode.

This example illustrates an important point about lexical functions that return character string values: if you use a lexical function in an expression, you must remember that substitution will be repetitive. If you intend the result of the function to be used as a literal character string value, as above, you must enclose the function in quotation marks.

Note what would happen if you had entered the IF command as follows:

```
$ IF 'F$MODE()' .EQS. "BATCH" THEN GOTO BATDEF
```

The command interpreter would replace the function as it lexically scans the command line. Assuming interactive mode, the result would be:

```
$ IF INTERACTIVE .EQS. "BATCH" THEN GOTO BATDEF
```

The command interpreter would then attempt to replace INTERACTIVE with its current value, as the string is interpreted to be a symbol name (it begins with an alphabetic character and is not enclosed in quotation marks). If there were no symbol defined for the name INTERACTIVE, an error would occur. Thus, 'F$MODE()' needs quotation marks and, because it is quoted, it needs two apostrophes (as shown above and as described in Section 4.2.1).

5.2.2 The F$VERIFY Lexical Function

If you use the F$VERIFY function with no argument, it returns a value of 0 or 1, based on whether verification of command procedures is off (0) or on (1). You can use this function to test or to save the current setting.

For example, a command procedure can save the current setting before changing it and then later restore the setting:

```
$ SAVE_VERIFY = 'F$VERIFY()'
$ SET NOVERIFY
    .
    .
    .
$ IF SAVE_VERIFY THEN SET VERIFY
```

The assignment statement saves the current verification setting before the SET NOVERIFY command sets verification off. Later, the value of SAVE_VERIFY is tested; if it has a value of 1, verification was previously on; if so, the SET VERIFY command is executed and verification is restored. Otherwise, verification was initially off and remains off.

In this arithmetic assignment statement, apostrophes surround the lexical function even though you do not normally use apostrophes on the right-hand side of an arithmetic assignment statement. This is because apostrophes are required when you specify a lexical function; the function is evaluated when the command input is processed and not when the command itself is executed.

If you use the F$VERIFY function with an argument, the function still returns the current verification setting. However, the command interpreter then examines the state of the low-order bit in the argument and turns verification off if the value is 0, or on if the value is 1.

For example, you could construct a procedure that will not display (or print) commands, regardless of what the initial state of verification is:

```
$ VERIFY = 'F$VERIFY(0)'
.
.
.
$ IF VERIFY .EQ.1 THEN SET VERIFY
```

This procedure uses the assignment statement to set verification off when the assignment is scanned, then restores the previous setting at the end of the procedure.

Note that the argument can be a character string enclosed in quotation marks, if the string begins with uppercase T, Y, F, or N. In these cases, the argument is resolved as an operand in an arithmetic comparison; the resulting value is 1 for strings beginning with T or Y, and 0 for strings beginning with F or N. Thus, the assignment statement above could be expressed as:

```
$ VERIFY = 'F$VERIFY("NO")'
```

5.2.3 The F$DIRECTORY Lexical Function

The F$DIRECTORY function returns the current default directory name string, including square brackets ([]). If you use the SET DEFAULT command and specify angle brackets (<>) in a directory specification, the F$DIRECTORY function returns angle brackets in the directory string.

The F$DIRECTORY function has no arguments.

The following example shows how to use the F$DIRECTORY function to save the current default directory in a command procedure and later restore it:

```
     $ SAVE_DIR :='F$DIRECTORY()'
     $ SET DEFAULT [MALCOLM.TESTFILES]
        .
        .
        .
     $ SET DEFAULT 'SAVE_DIR'
```

In this example, the assignment statement equates the current directory to the symbol SAVE_DIR. Then, the SET DEFAULT command establishes a new default directory. Later, the symbol SAVE_DIR is used in the SET DEFAULT command that restores the original default directory.

5.2.4 The F$PROCESS Lexical Function

The F$PROCESS lexical function returns the current process name string. The F$PROCESS function has no arguments.

By default, an interactive user has a process name string that is the same as the login user name. A batch job is given a process name in the format _JOBxxx where xxx is the job number assigned to the job.

NOTE

For an example of the F$PROCESS lexical function, see the sample procedure ENDED.COM in Appendix A.

5.2.5 The F$USER Lexical Function

The F$USER lexical function returns the current user identification code (UIC), including brackets ([]). The F$USER function has no arguments.

The following example shows how a directory with a name in UIC format is saved, then restored in a command procedure.

```
     $ SAVE_UIC := 'F$USER()'
     $ SET UIC [1,1]
        .
        .
        .
     $ SET UIC 'SAVE_UIC'
```

Using F$USER ensures that the directory corresponds to the current UIC when it is restored. Directories in UIC format ensure the compatibility of Files-11 Structure Level 1 disks between VAX/VMS and RSX-11M systems.

5.2.6 The F$LOGICAL Lexical Function

The F$LOGICAL function translates a logical name and returns the equivalence name string. The translation is not recursive, that is, the resulting string is not checked to determine whether it is a logical name. The function uses the normal search order to locate the logical name: it searches the process, group, and system logical name tables, in that order, and returns the equivalence name for the first match found.

The format of the F$LOGICAL function is:

```
'F$LOGICAL(logical-name)'
```

The logical-name is either the literal logical name to be translated (enclosed in quotation marks) or a symbol name whose value is the logical name to be translated.

You can use the F$LOGICAL function to save the current equivalence of a logical name and later restore it. The following example shows the use of the F$LOGICAL function to determine the name of the current terminal device and the creation of a group logical name table entry based on the equivalence string:

```
$ DEFINE/GROUP TERMINAL 'F$LOGICAL("TT")'
```

This example illustrates another important point about lexical functions: all arguments specified for lexical functions are automatically replaced. This means that arguments that begin with alphabetic letters are assumed to be symbol names and the command interpreter will attempt to replace them. If the symbol is undefined, the command interpreter will replace it with a null string. Thus, if the argument is not a symbol name, you must enclose it in quotation marks.

The following example combines the F$DIRECTORY and F$LOGICAL lexical functions:

```
$ SAVE_DIR := 'F$LOGICAL("SYS$DISK")''F$DIRECTORY()'
```

This assignment statement concatenates the results of two lexical functions. The symbol SAVE_DIR consists of a full device and directory name string.

If there is no current assignment for a specified logical name, the function returns a null string. Thus, to test for an unassigned name, you could use a command similar to the following:

```
$ IF "''F$LOGICAL("INFILE")'" .EQS.  "" THEN GOTO ASSIGN
```

The lexical function is enclosed in quotation marks to ensure that it is evaluated as a literal and not a symbol.

The next example shows how you can test for an unassigned logical name by anticipating a null string returned from the F$LOGICAL function:

```
$ IF "''F$LOGICAL("OUTFILE")'" .NES. "" THEN   -
   DEASSIGN OUTFILE
$ ASSIGN 'P1' OUTFILE
```

The IF command in the above example tests whether the logical name OUTFILE is currently assigned. If so, OUTFILE is deassigned. The next command assigns an equivalence to the logical name OUTFILE. Thus, the ASSIGN command will complete without issuing the normal success message indicating that the name is already assigned.

NOTE

The sample procedures BWAKE.COM and ENDED.COM in Appendix A contain examples of the F$LOGICAL lexical function.

5.2.7 The F$TIME Lexical Function

The F$TIME lexical function returns the current date and time string. The F$TIME function has no arguments.

The time string returned has the following fixed, 23-character format:

```
dd-mmm-yyyy hh:mm:ss.cc
```

When the current day of the month is any of the values 1 through 9, the first character in the returned string is a blank character; thus, the time portion of the string is always in character position 13, that is, at an offset of 12 characters from the beginning of the string.

You can use this function to time-stamp files that you create with command procedures. For example:

```
$ TIME_STAMP := 'F$TIME()'
$ WRITE OUTFILE TIME_STAMP
```

In this example OUTFILE is the name of a file that is opened for writing. The WRITE command is described in detail in Chapter 8, "Reading and Writing Files."

For another example of the F$TIME function, see Section 5.3.3, "The F$EXTRACT Lexical Function."

NOTE

The sample procedure CONVERT.COM in Appendix A shows how to use the time string returned by F$TIME to calculate a delta time value.

5.2.8 The F$MESSAGE Lexical Function

The F$MESSAGE lexical function returns the message text, if any, associated with a specific numeric value.

The format of the F$MESSAGE function is:

```
'F$MESSAGE(status-code)'
```

The status-code is either a literal numeric value or a symbol name equated to a numeric value.

For example, the status code %X1C is associated with the message EXQUOTA. To obtain the text of this message, use the F$MESSAGE function as shown below:

```
$ ERROR_TEXT := 'F$MESSAGE(%X1C)'
```

After this assignment statement is made, the symbol ERROR_TEXT has the value:

```
%SYSTEM-F-EXQUOTA, EXCEEDED QUOTA
```

Note that the value for the symbol consists of all uppercase letters. Normally, system messages are displayed in lowercase letters; in fact, the message text portion of a message is maintained by the system in lowercase and returned by the function in lowercase. However, the processing of lexical functions occurs at the same time that the command interpreter translates lowercase letters to uppercase and compresses multiple blanks and tab characters to single blank characters.

To preserve the text of a message in lowercase characters, enclose the function in quotation marks and use two apostrophes to request substitution. For example:

```
$ A := "''F$MESSAGE(%X1C)'"
```

This assignment statement gives the symbol A the value:

```
%SYSTEM-F-EXQUOTA, exceeded quota
```

Note that although each message in the system message file has a numeric value or range of values associated with it, there are many possible numeric values that do not have corresponding messages. For more information on completion status values and messages, see Chapter 7, "Controlling Error Conditions and CTRL/Y Interrupts."

NOTE

The sample procedure ENDED.COM in Appendix A illustrates a batch job that uses F$MESSAGE to display the message associated with the completion status of the job.

5.3 STRING MANIPULATION FUNCTIONS

A string can be either a literal character string (one enclosed in quotation marks) or a symbol name that has been equated to a string value. The terms associated with string manipulation are "substring" and "offset":

- A substring is any contiguous set of characters within a string.
- An offset is the relative position of a character or a substring in a string with respect to the beginning of the string. The first character in a string is always offset position 0 from the beginning of the string (which always begins at the left-most character in the string).

The following lexical functions allow you to manipulate character strings:

- F$LENGTH returns the length of a specified string as a numeric value
- F$LOCATE returns the offset within a string of a specified character or character substring as a numeric value
- F$EXTRACT returns a substring from within a specified character string as a string value

5.3.1 The F$LENGTH Lexical Function

The F$LENGTH lexical function returns the length of a specified string. The format of the F$LENGTH function is:

```
'F$LENGTH(string)'
```

The string is either a literal character string (enclosed in quotation marks) or a symbol name equated to a string.

For example:

```
$ MESSAGE := "''F$MESSAGE(%X1C)'"
$ STRING_LENGTH = 'F$LENGTH(MESSAGE)'
```

After these assignment statements, the symbol MESSAGE has the value:

```
%SYSTEM-F-EXQUOTA, exceeded quota
```

The symbol STRING_LENGTH has a value equal to the number of characters in the value of the symbol named MESSAGE, that is, 33.

NOTE

For additional examples of F$LENGTH, see the sample procedures CONVERT.COM and FORTUSER.COM in Appendix A.

5.3.2 The F$LOCATE Lexical Function

The F$LOCATE lexical function locates a character or character substring within a string and returns its offset within the string. If the character or character substring is not found, the function returns the length of the string that was searched.

The format of the F$LOCATE function is:

```
'F$LOCATE(substring,string)'
```

The substring is the string of characters that you want to locate within the string and string is the string in which the characters are to be found. Specify substring and string using either literal character strings (enclosed in quotation marks) or symbol names equated to character strings.

For example:

```
$ FILE_SPEC := MYFILE.DAT;1
$ NAME_LENGTH = 'F$LOCATE(".",FILE_SPEC)'
```

The F$LOCATE function in the above example returns the position of the period in the string with respect to the beginning of the string. Thus NAME_LENGTH equals the length of the file name in the file specification MYFILE.DAT, that is, 6.

Note in the above example that the character to be located, the period, is specified literally and is therefore enclosed in quotation marks. The symbol FILE_SPEC is automatically replaced by its current value during the processing of the function.

A common technique to determine whether a character or character string is in a string is to compare the result of a locate function with the length of the string, as shown in the following example:

```
$ INQUIRE TIME "Enter time"
$ IF 'F$LOCATE(":",TIME)' .EQ. 'F$LENGTH(TIME)' THEN -
    GOTO ERROR
```

In this example, the INQUIRE command prompts for a time value. The IF command checks for the presence of a colon in the string entered in response to the prompt. If the value returned by the F$LOCATE function equals the value returned by the F$LENGTH function, the colon is not present.

You can also use the F$LENGTH function with an overlay expression in an assignment statement. The following example shows how to append a character string at the end of an existing string:

```
$ NAME := MYFILE
$ NAME['F$LENGTH(NAME)',4]:= .DAT
```

In the above example, the syntax of the assignment statement indicates that a substring in the string is to be replaced (this syntax is described in Section 3.2.2, "Replacing Substrings in Character String Symbol Values"). In this example, the value returned by the F$LENGTH function is used as a relative offset from the beginning of a string; thus it results in the string .DAT being placed in the character position beyond the last character in the string. The symbol NAME now has the value MYFILE.DAT.

5.3.3 The F$EXTRACT Lexical Function

The F$EXTRACT function returns a substring from a character string value. When you use this function, you specify the location within the string that marks the beginning of the substring and the number of characters you want to extract. The format is:

```
     'F$EXTRACT(offset,length,string)'
```

The offset is the position, relative to the beginning of the string, that marks the beginning of the substring you want to extract; the length is the number of characters you want to extract; and the string is the string from which the substring is to be extracted.

Specify offset and length using literal numeric values or symbol names equated to numeric values. Specify string using either a literal character string (enclosed in quotation marks) or a symbol name equated to a character string.

The following example shows a procedure that displays a different message depending on whether the current daytime is morning or afternoon. It first obtains the current time of day by using the F$TIME function. Then, it extracts the hours from the date and time string returned by F$TIME:

```
     $ TIME := "''F$TIME()'"
     $ IF 'F$EXTRACT(12,2,TIME)' .GE. 12 THEN GOTO AFTERNOON
     $ MORNING:
     $ WRITE SYS$OUTPUT "Good morning!"
     $ EXIT
     $ AFTERNOON:
     $ WRITE SYS$OUTPUT "Good afternoon!"
     $ EXIT
```

The string returned by F$TIME always contains the hours field beginning at an offset of 12 characters from the beginning of the string. The function is enclosed in quotation marks to ensure that when the string returned is assigned to the symbol named TIME, a leading blank, if there is one, will not be truncated (the date field contains a leading blank on the first day through the ninth day of each month).

The F$EXTRACT function extracts two characters from the string, beginning at this offset and compares the value extracted with the value 12.

Note, in the above example, that although the function extracts a character string, the value extracted is tested as a numeric value: the context in which the symbol is used determines the type of data it represents.

Frequently, manipulation of a character string value requires that you locate a particular character within a string and then extract a substring beginning or ending at that location. Because you cannot use a lexical function as an argument for a lexical function, you must use two assignment statements to achieve the desired results, as shown below:

```
     $ DOT = 'F$LOCATE(".",P1)'
     $ FILENAME := 'F$EXTRACT(0,DOT,P1)'
```

In this example, the symbol name DOT is given the numeric value representing the position of a period in the character string value of P1. Then, this symbol name is used as an argument in the F$EXTRACT function to specify the number of characters to extract from the string. If a procedure is invoked with the parameter MYFILE.DAT, these statements result in the symbol FILENAME being given the value MYFILE.

Note that the F$LOCATE function in the above example assumes that the file specification does not contain a node name or a directory specification including a subdirectory name. Checking a file specification for various fields would require a more sophisticated sequence of functions.

5.4 FUNCTIONS THAT MANIPULATE BINARY DATA

There are two methods used to assign binary data to a symbol name. The first method is to perform a binary overlay in the current value of a symbol name by using the syntax:

```
     $ symbol-name[bit-position,width]= numeric-expression
```

as described in Section 3.3.6, "Arithmetic Overlays."

The second method is to use the READ command to read data into a symbol name from a file whose records contain binary data. The READ command is described in Section 8.1, "Reading Files."

Two lexical functions are provided to manipulate binary data that has been so assigned to a symbol name:

F$CVSI,	for operations on signed quantities
F$CVUI,	for operations on unsigned quantities

These integer conversion functions extract bit fields from binary data and convert the result to either signed (F$CVSI) or unsigned (F$CVUI) decimal values. The formats of these functions are:

```
     'F$CVUI(bit-position,width,integer)'

     'F$CVSI(bit-position,width,integer)'
```

The bit-position is the offset of the value to be converted from the beginning of the integer; the width is the number of bits that are to be extracted for conversion to a decimal value; and the integer is the value of the 32-bit integer from which the bits are taken. Note that the rightmost bit of an integer is the low-order bit. This bit is position number 0 for determining the offset. You specify the value of the integer by using either its literal numeric value or by using a symbol equated to its numeric value.

For example, consider the following arithmetic overlay of the symbol name A, where the hexadecimal value 2B is assigned to all 32 bits of the symbol:

```
     $ A[0,32]=%X2B
```

Note that the ASCII representation of this symbol name is now the + character; you could determine this by typing the command:

```
     $ SHOW SYMBOL A
```

The F$CVSI and F$CVUI lexical functions now can be used to extract fields from the symbol A and convert these fields to their decimal values. For example, consider the extraction (and conversion) of the low-order four bits from the symbol A:

```
     X = 'F$CVSI(0,4,A)'

     Y = 'F$CVUI(0,4,A)'
```

The results of these conversions are X = -5 and Y = 11, because the F$CVSI function treats the low-order four bits of the value %X2B as a signed integer while the F$CVUI function treats the low-order four bits of the value %X2B as an unsigned integer.

Note that the arithmetic overlay and READ command are the only ways to assign binary values to symbols: and other method used to assign values to symbols does not produce binary values. For example, assigning a numeric value of 1 to the symbol name A results in the value of A being stored by the command interpreter as an ASCII representation of the character 1.

NOTE

The sample procedure CALC.COM in Appendix A shows how to use the F$CVUI lexical function to convert a decimal value to its hexadecimal equivalent.

CHAPTER 6

CONTROLLING EXECUTION FLOW IN COMMAND PROCEDURES

The normal flow of execution in a command procedure is sequential: the commands in the procedure are executed, in order, until the end-of-file is reached. However, in many cases you will want to control (1) whether certain statements are executed or (2) the conditions under which the procedure should not continue execution.

This chapter discusses the basic commands that you can use in a command procedure to control or alter the flow of execution:

- The IF command tests the value of a symbol or expression and executes a given command string based on the result of the test.
- The GOTO command transfers control to a labeled line in the procedure.
- The Execute Procedure (@) command invokes (or calls) another command level and begins execution of another command procedure.
- The EXIT and STOP commands terminate the current procedure and restore control either to the calling command procedure or to command level 0, respectively.

6.1 THE IF COMMAND

The IF command tests a symbol value or an arithmetic expression and executes a given command if the result of the expression is true. An expression is true if:

- It has an odd numeric value.
- It has a character string value that begins with any of the letters Y, y, T, or t.

An expression is false if:

- It has an even numeric value.
- It has a character string value that begins with any of the letters N, n, F, or f.

The following examples show valid expressions used in IF commands.

Example	Explanation
`$ IF A + B .EQ. 10 THEN command`	Executes the given command if the sum of the defined symbols A and B is 10
`$ IF A THEN command`	Executes the given command if the symbol A has an odd numeric value or is equated to a character string that begins with the letters Y,y (yes) or T,t (true)
`$ IF .NOT. A THEN command`	Executes the given command if the symbol A has an even numeric value or is equated to a character string that begins with the letter N,n (no) or F,f (false)
`$ IF COUNT.LE.100 THEN command`	Executes the given command if the symbol COUNT has a current value less than or equal to 100
`$ IF P1.EQS."TYPE" THEN command`	Executes the given command if the first parameter passed to the command procedure was the word TYPE

The target command of an IF command can be any valid DCL command; you can optionally precede the command with a dollar sign. The command is executed only if the expression is true. Otherwise, execution continues with the next command in the procedure, as illustrated in Figure 6-1. After the THEN command string is executed, control returns to the next command in the sequence unless the THEN command-string results in a transfer of control.

The target command of an IF command can be another IF command. For example:

```
$ IF A .EQ.  B THEN -
  IF C .EQ.  D THEN -
  IF E .EQ.  F THEN -
  RESULT = 1
```

In this IF command, each expression is tested in turn. If the result of the first expression is true, the second IF command is executed; if that expression is true, the next IF command is executed. If all of the IF command expressions are true, RESULT is given a value of 1; otherwise, RESULT is not given a value.

The only practical limit on the number of IF commands that can be nested in a' single command string is the limit on the number of characters that can be specified in a command string. This limit is approximately 500 characters.

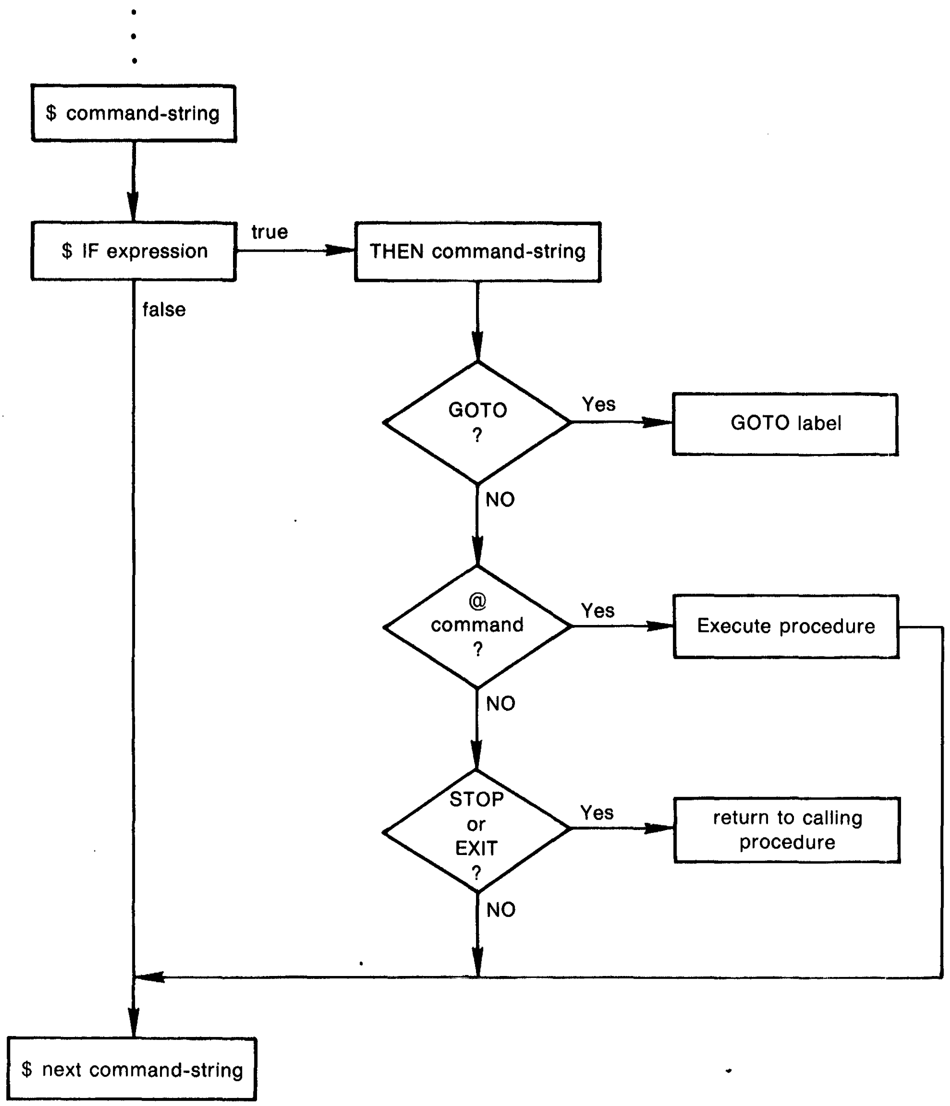


Figure 6-1 The IF Command

6.1.1 Using Logical Operators in IF Commands

The example in the preceding section can also be written using a logical AND operation, as follows:

```
$ IF A .EQ.  B .AND.  -
  C .EQ.  D .AND.  -
  E .EQ.  F THEN -
  RESULT = 1
```

This IF command expression consists of several operations; the THEN command string is executed only if all tests performed within the expression are true. Note that the .EQ. operator has a higher precedence than the .AND. operator; the arithmetic comparisons in this command will be performed before the logical comparisons.

The other logical operators, OR and NOT, are also useful in IF expressions. For example, the following command tests whether one or the other of two expressions is true:

```
     $ IF P1.EQS."DISK" .OR.  P1.EQS."TAPE" THEN GOTO 'P1'
```

The THEN command string in this example is executed if the parameter P1 is currently equal to either of the character strings DISK or TAPE.

Note that when you use the AND or the OR logical operators, the expressions on each side of the operator must be complete. The syntax is:

```
     $ IF expression .OR.  expression THEN command
     $ IF expression .AND.  expression THEN command
```

The logical NOT operator tests whether an expression is not true, and therefore reverses the sense of the test. For example:

```
     $ IF .NOT.  RESULT THEN command
```

The IF command above tests whether RESULT is false. This construct is useful following INQUIRE commands. For example:

```
     $ INQUIRE CONT "Do you want to continue"
     $ IF .NOT.  CONT THEN EXIT
```

If the response to the INQUIRE command is any even numeric value or any character string that begins with an N or n (meaning no) or an F or f (meaning false), the procedure does not continue execution.

Expressions can be simple or compound. For example, a simple expression can consist of a single symbol:

```
     RESULT
```

A compound expression can consist of several operations:

```
     (P1 .EQS.  "DISK") .OR.  (P1 .EQS.  "TAPE") .AND.  (P2 .LE.  5)
```

This expression is true if the symbol P1 is equated to either one of the character strings DISK or TAPE and the symbol P2 has a numeric value that is less than or equal to 5.

For complete details on the syntax of expressions and how to specify each type of operation that is valid, see Section 3.3, "Equating Symbols to Numeric and Logical Expressions."

6.1.2 Using Symbols in IF Commands

Expressions in IF commands are automatically evaluated during the execution of the command. All character strings beginning with alphabetic letters that are not enclosed in quotation marks are assumed to be symbol names and the IF command replaces them with their current values.

Symbol substitution in this context is not recursive; that is, each symbol is replaced only once. However, if you want recursive substitution, you can precede a symbol name with an apostrophe (') or ampersand (&) operator so the command interpreter will perform substitution during command input or parsing.

The command interpreter does not execute an IF command when it contains an undefined symbol. Instead, the command interpreter issues a warning message and executes the next command in the procedure.

Symbol substitution and recursive substitution are described in detail in Chapter 4, "Symbol Substitution in Command Procedures." The following paragraphs contain some specific examples of substitution with the IF command.

```
$ A := B
$ IF A .EQS.  "B" THEN ...
```

This IF command compares the value of the symbol A with the literal value B. Note that if you do not enclose B in quotation marks, the IF command assumes that B is a symbol name, attempts to replace it, and issues a warning message if it fails to find the symbol name B.

The next example shows how to construct an IF command to recursively check each parameter passed to a procedure:

```
$    COUNT = 0
$ LOOP:
$    COUNT = COUNT + 1
$    IF COUNT .EQ.  9 THEN EXIT
$    IF P'COUNT' .EQS.  "" THEN EXIT
$    APPEND/NEW &P'COUNT' SAVE.ALL
$    DELETE &P'COUNT';*
$    GOTO LOOP
$ EXIT
```

In this example, the IF command string is written so that each time through the loop, the command interpreter replaces the symbol COUNT with its current value. Each time the IF command executes, the resulting symbol name (P1, then P2, then P3, and so on) is replaced within the expression. The APPEND and DELETE commands, however, must use the ampersand (&) substitution operator on these parameters to force recursive substitution. Recursive substitution with ampersands is described in Section 4.5.4.

Note that P1 through P8 are never undefined when a procedure begins execution. The command interpreter defines these symbols at each command level above command level 0; they are all initially given null values. When you do specify parameters for procedures, your specifications override the default values.

The above example also shows how to use the GOTO command to establish a loop in a command procedure; the GOTO command is described next.

6.2 THE GOTO COMMAND

The GOTO command passes control to a labeled line in a command procedure. You can precede any command string in a command procedure with a label. The rules for entering labels are:

- A label must appear as the first item on a line.
- A label can have up to 255 characters.
- No blanks can be embedded within a label.
- A label must be terminated with a colon (:).

For example:

```
     $ GOTO BYPASS
         .
         .
         .
     $ BYPASS:
```

As the command interpreter encounters labels, it enters them in a table, space for which is allocated from space available in the local symbol table. If a label is encountered that already exists in the table, the new definition replaces the existing one. Note that the amount of space available for labels is limited. If a command procedure uses many symbols and contains many labels, the command interpreter may run out of table space and issue an error message.

Figure 6-2 illustrates how the GOTO command affects the flow of execution in a command procedure.

The most common uses of GOTO commands are as targets of IF commands and as a means of establishing loops, as described in Sections 6.2.1 and 6.2.2 below. You can also use labels to define segments of command procedures; to define target statements for error conditions in the CLOSE, READ, and WRITE commands; and to handle end-of-file conditions in the READ command. (See Chapter 8 for more information on the file-handling commands.)

```
$ LABELA:                       ❶
$ command-string
 .
 .
 .
$ GOTO LABELA                   ❷
$ command-string
 .
 .
 .
$ GOTO LABELB                   ❸
$ command-string
 .
 .
 .
$ LABLEB
$ command-string
 .
 .
 .
$ GOTO LABELC                   ❹
 .
 .
 .
$ EXIT
```

❶ LABELA is put into the label table for this command level.

❷ When a GOTO command is executed, the command interpreter checks the label table for this command level. If the label is found, control transfers to the command immediately after the label.

❸ If the label is not in the label table when a GOTO command is executed, the command interpreter scans forward through the procedure to locate the label. If found, the label is placed in the label table and control transfers to the command immediately following the label.

❹ If the label is not in the label table when the GOTO command is executed, and the command interpreter cannot find the label, the procedure exits immediately. The EXIT command is not executed.

Figure 6-2 The GOTO Command

6.2.1 Using GOTO as a Target of IF

The GOTO command is especially useful as the target command of an IF command. Used together, these commands can cause a procedure to branch forward or backward according to variable conditions or according to parameters that you pass to the procedure.

For example, when you use parameters in a command procedure, it is good practice to test the parameters at the beginning of the procedure. A procedure that you execute interactively could begin with the lines:

```
$ IF P1.NES."" THEN GOTO OKAY
$ INQUIRE P1 "Enter file spec"
$ OKAY:
$ PRINT/HOLD/COPIES=10/FORMS=B  'P1'
```

In this example, the IF command checks that P1 is not a null string. If P1 is a null string, the GOTO command is not executed and the INQUIRE command prompts for a parameter value. Otherwise, the GOTO command causes a branch around the INQUIRE command. In either case, the procedure executes the PRINT command following the line labeled OKAY.

6.2.2 Using GOTO to Establish Loops

With the GOTO command, you can establish several kinds of loop. Three examples follow.

You can use the GOTO command in loops that execute a defined number of times. The procedure establishes a counter, increases or decreases the counter, and tests the counter's value. When the counter reaches a defined value, the procedure exits from the loop. For example:

```
$ COUNT=0
$ LOOP:
$       COUNT=COUNT+1
      .
      .
      .
$       IF COUNT.LE.10.THEN GOTO LOOP
$ CONTINUE
```

In this example, the command procedure exits from the loop when the value of COUNT reaches 11.

You can use the GOTO command in loops that prompt for the user to indicate whether execution should continue. For each iteration of the loop, the procedure prompts for input data or a value for a variable. For example:

```
$ LOOP:
$       INQUIRE FILE "FILE"
$       IF FILE .EQS.  "" THEN GOTO SKIP
      .
      .
      .
$       GOTO LOOP
$ SKIP:  CONTINUE
```

In this example, the INQUIRE command requests a file name. If the response from the interactive level is a null value (a CTRL/Z or a RET) the loop is not executed. Otherwise, the loop executes, iteratively, until a null value is entered.

You can use the GOTO command in loops that make a specific test during each iteration. The procedure executes the loop until the test is satisfied, then branches. The example loop in Section 6.1.2 is such a test.

6.3 NESTING PROCEDURES: THE EXECUTE PROCEDURE COMMAND

The GOTO command described in the preceding section provides one way to segment command procedures into more easily read and understood sections. In a procedure that is more complex, you may find it useful to separate procedures into several smaller procedures. Or, you may find it convenient to develop small, generalized procedures that perform common functions and to invoke these procedures from other procedures that you write.

Using the Execute Procedure (@) command to invoke new levels of command execution is similar to a CALL statement in a high-level programming language. Procedures can be nested to a maximum of eight levels. At each command level, logical name assignments for process permanent files can change; these changes are discussed in Section 2.1.

Some of the techniques you can use to pass information from one command level to another involve:

- Passing parameters. You can pass up to eight variable parameters to a procedure you invoke using the Execute Procedure (@) command. Techniques for passing parameters to command procedures are described in Section 3.6.

- Using global symbols. You can use global symbols to pass variable data from one procedure to another; a global symbol defined in a nested command procedure can be referred to in all command procedures. Global symbols are described in Section 3.5.2.

NOTE

The sample procedure CONVERT.COM in Appendix A is a generalized procedure that can be called by any other procedure. It accepts a parameter passed to it and sets a global symbol value for the caller.

6.4 THE EXIT AND STOP COMMANDS

The EXIT and STOP commands both provide a way to terminate the execution of a procedure. The EXIT command terminates execution of the current command procedure and returns control to the calling command level. The STOP command also terminates execution of a procedure; however, when a STOP command is executed, the command interpreter returns to command level 0, regardless of the current command level. If you execute the procedure in a batch job, the batch job terminates.

6.4.1 Using the EXIT Command

You can use the EXIT command to ensure that a procedure does not execute certain lines. For example, if you write an error handling routine at the end of a procedure, you would place an EXIT command preceding the routine, as follows:

```
     .
     .
     .
$       EXIT !  End of normal execution path
$ ERROR_ROUTINE:
     .
     .
     .
```

The EXIT command is also useful for writing procedures that have more than one execution path. For example:

```
$ START:
$        IF P1.EQS."TAPE".OR.P1.EQS."DISK" THEN GOTO 'P1'
$        INQUIRE P1 "Enter device (TAPE or DISK)"
$        GOTO START
$ TAPE:  !  Process tape files
     .
     .
     .
$       EXIT
$ DISK:  !  Process disk files
     .
     .
     .
$       EXIT
```

To execute this command procedure, you must enter either TAPE or DISK as a parameter. The IF command uses a logical OR to test whether either of these strings was entered. If so, the GOTO command branches appropriately, using the parameter as the branch label. If P1 was neither TAPE nor DISK, the INQUIRE command prompts for a correct parameter; the GOTO START command establishes a loop.

The commands following each of the labels TAPE and DISK provide different paths through the procedure. The EXIT command before the label DISK ensures that the commands after the label DISK are not executed unless the procedure explicitly branches to DISK.

Note that the EXIT command at the end of the procedure is not required because the end-of-file of the procedure causes an implicit EXIT command.

6.4.2 Passing Status Values with the EXIT Command

The EXIT command accepts an optional parameter, called a status code value. When a command procedure has multiple levels of interaction, you can use the EXIT command to pass status values from nested levels back to their callers. The exit code defines a value for the global symbol named $STATUS. $STATUS is a special, reserved symbol name maintained by the command interpreter.

For example, suppose the procedure A.COM contains:

```
$ @B
$ IF $STATUS .EQ. 3 THEN GOTO CONTROL
    .
    .
    .
```

The procedure B.COM contains the line:

```
$ EXIT 3
```

This EXIT command places the value 3 in the global symbol $STATUS, which is tested by the calling procedure, A.COM.

Note that you can use any numeric value or expression with the EXIT command: the EXIT command automatically performs symbol substitution and expression evaluation. For example:

```
$ EXIT A+B
```

The above command is valid if the symbols named A and B are both currently defined with arithmetic values.

If you do not set a value for an EXIT command when a procedure is terminated, the command interpreter gives it a default value, based on the status value returned from the most recently executed command or program. For information on how this value is set and how you can establish default courses of action for a command procedure based on its value, see Chapter 7, "Controlling Error Conditions and CTRL/Y Interrupts."

6.4.3 Using the STOP Command

You can use the STOP command in a command procedure or batch job to ensure that all procedures are terminated if a severe error occurs.

You can also use the STOP command to halt the interactive execution of a procedure after interrupting it. For example:

```
$ @TESTALL RET
CTRL/Y

^Y
$ STOP RET
```

In the above example, the procedure TESTALL is interrupted by CTRL/Y. The STOP command terminates processing of the procedure and restores command level 0.

NOTE

The sample procedure BWAKE.COM in Appendix A uses the STOP command in a batch job to halt the job.

CHAPTER 7

CONTROLLING ERROR CONDITIONS AND CTRL/Y INTERRUPTS

This chapter describes how to control command procedure execution when an error condition or a CTRL/Y interrupt occurs.

Error conditions are detected by various VAX/VMS (or applications) components and are stored in the reserved global symbol $STATUS. The lowest three bits of this integer value provide the current value of the reserved global symbol $SEVERITY.

A CTRL/Y interrupt is the result of pressing CTRL/Y during command procedure execution.

7.1 ERROR CONDITION HANDLING

If an EXIT command does not explicitly set a value for $STATUS, the command interpreter uses the current value of $STATUS. This value is set implicitly by individual commands and programs that execute in a procedure. The values that are set, called condition codes, provide information about the termination of a program image, and you can provide action routines and error handling statements in your procedures based on values in $STATUS as described in the following sections.

7.1.1 Severity Levels

The low-order three bits of the status value contained in $STATUS represent the severity of the condition. The reserved global symbol $SEVERITY always contains only this portion of the condition code. These values, and the severity levels they represent are:

Value	**Severity**
0	Warning
1	Success
2	Error
3	Information
4	Severe, or fatal, error

Note that the success and information codes have odd numeric values and warning and error codes have even numeric values. You can test for the successful completion of a command with IF commands that perform logical tests on these values, as shown below:

```
$ IF $SEVERITY THEN ...
$ IF $STATUS THEN ...
```

When the current value in $SEVERITY or $STATUS is odd, the command or program completed successfully (the IF expressions are true). Otherwise, the IF expressions are false, indicating that the command or program did not complete successfully.

The converse of this test is a logical NOT operation, for example:

```
$ IF .NOT. $STATUS THEN ...
```

The command interpreter also uses the severity level of a condition code to determine whether to take specific action defined by the ON command. If an ON command action exists for a specific severity level, for example, for error conditions, that action will be taken. If a command results in an error, the specified action is taken and the next statement in the procedure will not be executed.

7.1.2 **The ON Command**

During the execution of a command procedure, the command interpreter checks the condition code returned from each command or program that executes. With the ON command, you can establish a course of action for the command interpreter to take based on the result of the check.

By default, the command interpreter executes an EXIT command when an error or severe error occurs, and continues when warnings occur. You can override this default with the ON command. An ON command establishes a default command action when condition code of a specified severity level and above occur.

If an ON command action is established for a specific severity level, when errors of lesser severities occur the command interpreter will continue processing the file. Table 7-1 illustrates the ON command keywords that define command actions and the action taken by the command interpreter on condition code at other severity levels.

Table 7-1
Severity Levels for ON Command Actions

ON Command Severity Level	Action Taken at Different Severity Levels		
	WARNING	ERROR	SEVERE_ERROR
WARNING	Specified action	Specified action	Specified action
ERROR	Continue	Specified action	Specified action
SEVERE_ERROR	Continue	Continue	Specified action

For example, if you want a procedure to exit when warnings, errors, and severe errors occur, use the command:

```
$ ON WARNING THEN EXIT
```

If you want the procedure to continue if a warning or an error occurs, but to exit if a severe error occurs, use the command:

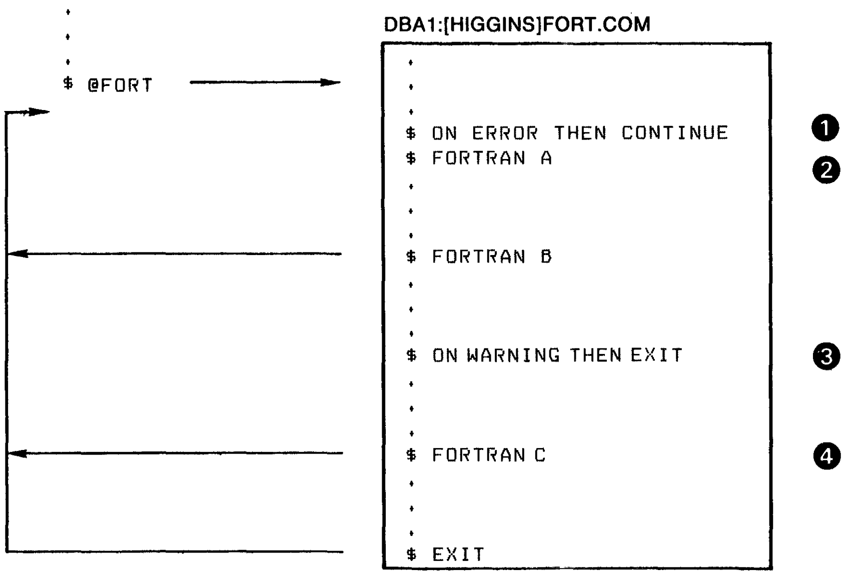
```
$ ON SEVERE_ERROR THEN EXIT
```

This ON command requests that the procedure exit only in the case of a severe error. If any command in the procedure incurs a warning or error condition, execution will continue with the next command in the procedure. If a severe error occurs, however, the procedure exits.

An ON command action is executed only once; thus, if you have used the above command, the command interpreter continues after an error occurs, but resets the default condition. If a second error occurs, and no other ON command has been encountered, the procedure exits. Figure 7-1 illustrates ON command actions.

NOTE

The sample procedures FORTUSER.COM and CALC.COM in Appendix A illustrate the use of the ON command to establish error handling.



1. This ON command overrides the default command action (on warning, continue; on error or severe error, exit). If an error or severe error occurs while A.FOR is being compiled, the command procedure continues with the next command.
2. The default command action is reset if the previous ON command takes effect. Thus, if an error or severe error occurs while B.FOR is being compiled, the command procedure exits.
3. If the command procedure does not exit before this command is executed, this command action takes effect.
4. If a warning, error, or severe error occurs while C.FOR is being compiled, the command procedure exits.

Figure 7-1 ON Command Actions

The action specified by an ON command applies only within the command procedure in which the command is executed. Therefore, if you execute an ON command in a procedure that calls another procedure, the ON command action does not apply to the nested procedure. In fact, an ON command executed at any command procedure level does not affect the error condition handling of procedures at any other level.

7.1.3 Disabling Error Checking

You can use the SET NOON command to request the command interpreter to not check the status returned from any commands. When the SET NOON command is in effect, the command interpreter does not perform any checking of $STATUS. For example:

```
$ SET NOON
$ RUN TESTA
$ RUN TESTB
$ SET ON
```

The SET NOON command preceding these RUN commands ensures that if either of the programs TESTA or TESTB return error conditions the procedure will continue. The SET ON command restores error checking by the command interpreter.

When a procedure disables error checking, it can explicitly check the value of $STATUS following the execution of each command or program. For example:

```
$ SET NOON
$ FORTRAN MYFILE
$ IF $STATUS THEN LINK MYFILE
$ IF $STATUS THEN RUN MYFILE
$ SET ON
```

In the above example, the first IF command checks whether $STATUS has a true value, that is, an odd numeric value. If so, the FORTRAN command was successful and the LINK command will be executed. After the LINK command, $STATUS is tested again. If $STATUS is odd, the RUN command will be executed; otherwise, the RUN command will not be executed. The SET ON command restores the current ON condition action; that is, whatever condition was in effect before the SET NOON command was executed.

The SET ON or SET NOON command applies only at the current command level, that is, the command level at which the command is executed. If you use the SET NOON command in a command procedure that calls another command procedure, the default error checking will be in effect within the nested procedure. Note that SET NOON has no meaning at command level 0.

7.1.4 System Messages

When a DCL command, user program, or command procedure completes execution, the command interpreter saves the condition code value in the global symbol $STATUS. For example, if an error occurs during a TYPE command, the value in $STATUS represents the specific error returned by the TYPE command. When a command or program completes successfully, $STATUS has an odd value.

Note that the command interpreter always maintains and displays the current value of $STATUS in hexadecimal.

When any command procedure exits and returns control to another command level, the command interpreter tests the current value of $STATUS. If $STATUS contains an even numeric value, and if its high-order digit is 0, the command interpreter will display the system warning or error message associated with that status code, if one exists. (Otherwise, the message NOMSG will be displayed.)

However, when a command procedure exits following a warning or error condition, the command interpreter sets the high-order digit of $STATUS to 1, leaving the remainder of the value intact. Many system programs that issue their own messages also set this field to 1 so that the command interpreter does not redisplay the message associated with the status value.

7.1.5 Commands that Do Not Set $STATUS

Most DCL commands invoke system utilities that generate unique status values and error messages based on different results. However, there are several commands that do not change the values of $STATUS and $SEVERITY if they complete successfully. These commands are:

CONTINUE	GOTO
DEPOSIT	HELP
EOD	IF
EXAMINE	SHOW
EXIT	STOP
	WAIT

If any of these commands results in a nonsuccessful status, however, that condition code will be placed in $STATUS, and the severity level will be placed in $SEVERITY.

7.1.6 Status Codes Returned by Compatibility Mode Commands

The DCL commands that invoke RSX-11M programs that execute in compatibility mode do not use the standard error reporting mechanism of VAX/VMS. These commands do not use $STATUS to return explicit values based on different results. Thus, most compatibility mode commands can test or change only $SEVERITY.

7.2 CTRL/Y INTERRUPT HANDLING

When CTRL/Y is pressed during command procedure execution, control is given to a special command level, the CTRL/Y command level. When you execute a command procedure, you can use the CTRL/Y command level in either of the following ways:

- To interrupt the execution of the procedure and execute one or more DCL commands. Then you can either stop the execution of the procedure or, depending on the commands you entered, resume execution of the procedure.

- To provide a default action for the command interpreter to take when CTRL/Y is pressed during the execution of the procedure.

These techniques are described below.

7.2.1 Interrupting a Command Procedure

You can interrupt a command procedure that is executing interactively by pressing either CTRL/C or CTRL/Y . The effect is the same: the command interpreter establishes a new command level, called the CTRL/Y level, and prompts for command input. When the interruption actually occurs depends on the command that is executing:

- If the command currently executing is a command that is executed by the command interpreter itself (for example, IF, GOTO, or an assignment statement) the command completes execution before the command interpreter prompts for a command at the CTRL/Y level.

- If the command or program currently executing is a separate image (that is, an image not executed by the command interpreter), the command is interrupted and the command interpreter prompts for a command at the CTRL/Y level.

At the CTRL/Y level, the command interpreter stores the status of all previously established command levels, so that it can restore the correct status after any CTRL/Y interrupt.

After you interrupt a procedure, you can:

- Issue a DCL command that does not replace the image that is currently executing. Among these commands are the SET VERIFY, SHOW TIME, SHOW TRANSLATION, ASSIGN, EXAMINE, and DEPOSIT commands. After you issue one or more of these commands, you can resume the execution of the procedure with the CONTINUE command.

- Issue a DCL command that executes another image. When you issue any command that invokes a new image, the command interpreter returns to command level 0 and executes the command.

- Issue the STOP command to terminate the procedure's execution. This command restores control to command level 0. Note that because commands that execute new images have the same effect as the STOP command, you do not normally need to use the STOP command.

When you interrupt a command procedure during the execution of a command or program that is not executed by the command interpreter, then the CONTINUE command resumes the execution of the interrupted command or program. If you issue a command that invokes a new image, exit handlers declared by the previous image, if any, will be allowed to execute first.[1]

The VAX/VMS Command Language User's Guide lists the DCL commands that are executed by the command interpreter, that is, the commands you can safely issue at the CTRL/Y level without causing the current image to be stopped.

7.2.2 Setting a CTRL/Y Action Routine

The ON command, which defines an action to be taken in case of error conditions, also provides a way to define an action routine for a CTRL/Y interrupt that occurs during execution of a command procedure. The action that you specify overrides the default action of the command interpreter (that is, to prompt for command input at the CTRL/Y command level).

For example:

```
$ ON CONTROL_Y THEN EXIT
```

If a procedure executes the ON command shown above, a subsequent CTRL/Y interrupt during the execution of the procedure causes the procedure to exit. Control is passed to the previous command level.

When you press CTRL/Y to interrupt a procedure that has established a CTRL/Y action, the action is taken as follows:

- If the command currently executing is a command executed by the command interpreter, the command completes before the CTRL/Y action is taken.
- If the current command is to be executed by an image other than the command interpreter, the image is forced to exit and cannot be continued following the CTRL/Y action. If the image has declared an exit handler, however, the exit handler is executed before the CTRL/Y action is taken.

The execution of a CTRL/Y action does not reset the default action. A CTRL/Y action remains in effect until:

- The procedure terminates (as a result of an EXIT or STOP command, or a default error condition handling action)
- Another ON CONTROL_Y command is executed
- The procedure executes the SET NOCONTROL_Y command

For example, a procedure can contain the line:

```
$ ON CONTROL_Y THEN SHOW TIME
```

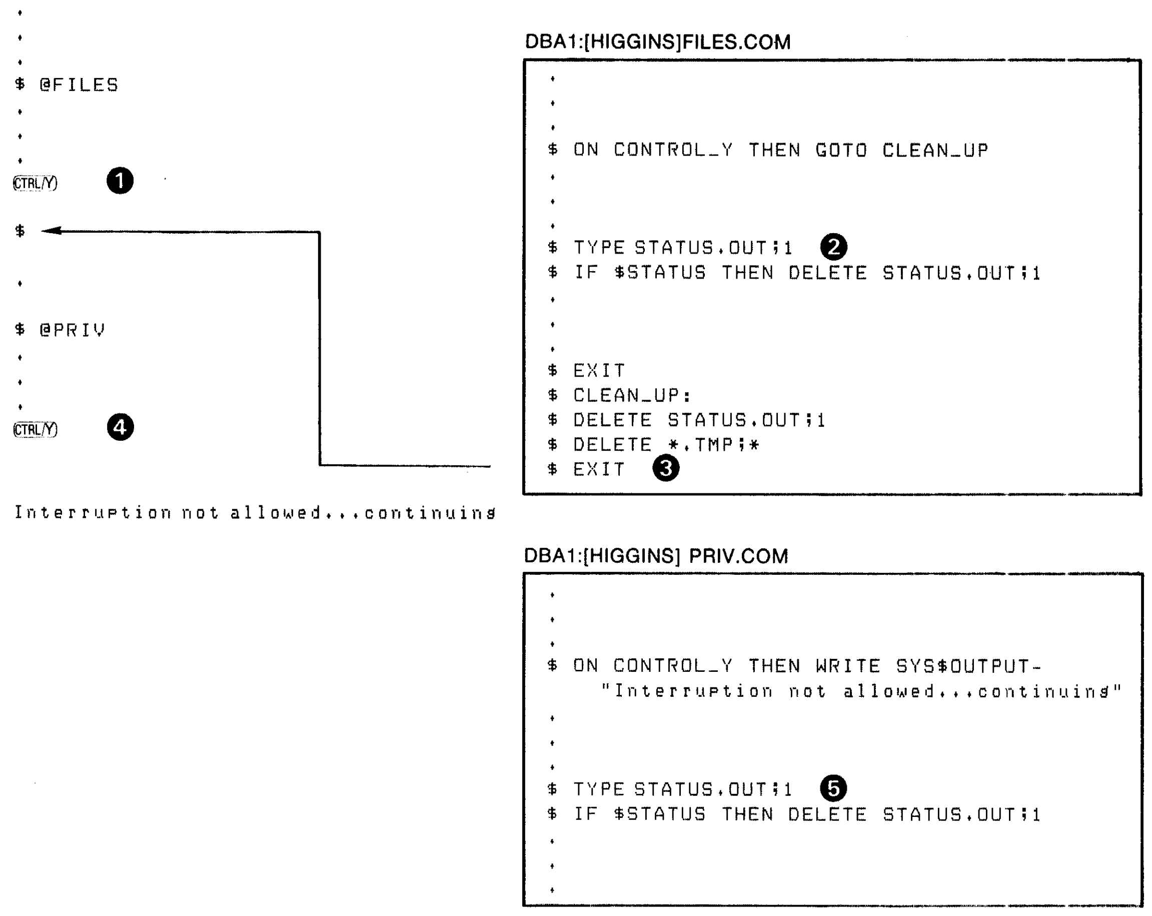
1. An exit handler is a routine that receives control to perform image-specific cleanup operations when an image exits. Exit handlers are described in detail in the VAX/VMS System Services Reference Manual and in various language reference manuals.

When this procedure executes, each CTRL/Y interrupt results in the execution of the SHOW TIME command. After each SHOW TIME command executes, the procedure resumes execution at the command following the command that was interrupted.

Figure 7-2 illustrates two ON CONTROL_Y commands and describes the flow of execution following CTRL/Y interruptions.

NOTE

The sample procedures EDITALL.COM and FORTUSER.COM in Appendix A, illustrate CTRL/Y action handling.



```
.
.
.
$ @FILES
.
.
.
CTRL/Y  ❶
$
.
$ @PRIV
.
.
.
CTRL/Y  ❹
Interruption not allowed...continuing
```

DBA1:[HIGGINS]FILES.COM

```
.
.
.
$ ON CONTROL_Y THEN GOTO CLEAN_UP
.
.
.
$ TYPE STATUS.OUT;1  ❷
$ IF $STATUS THEN DELETE STATUS.OUT;1
.
.
.
$ EXIT
$ CLEAN_UP:
$ DELETE STATUS.OUT;1
$ DELETE *.TMP;*
$ EXIT  ❸
```

DBA1:[HIGGINS] PRIV.COM

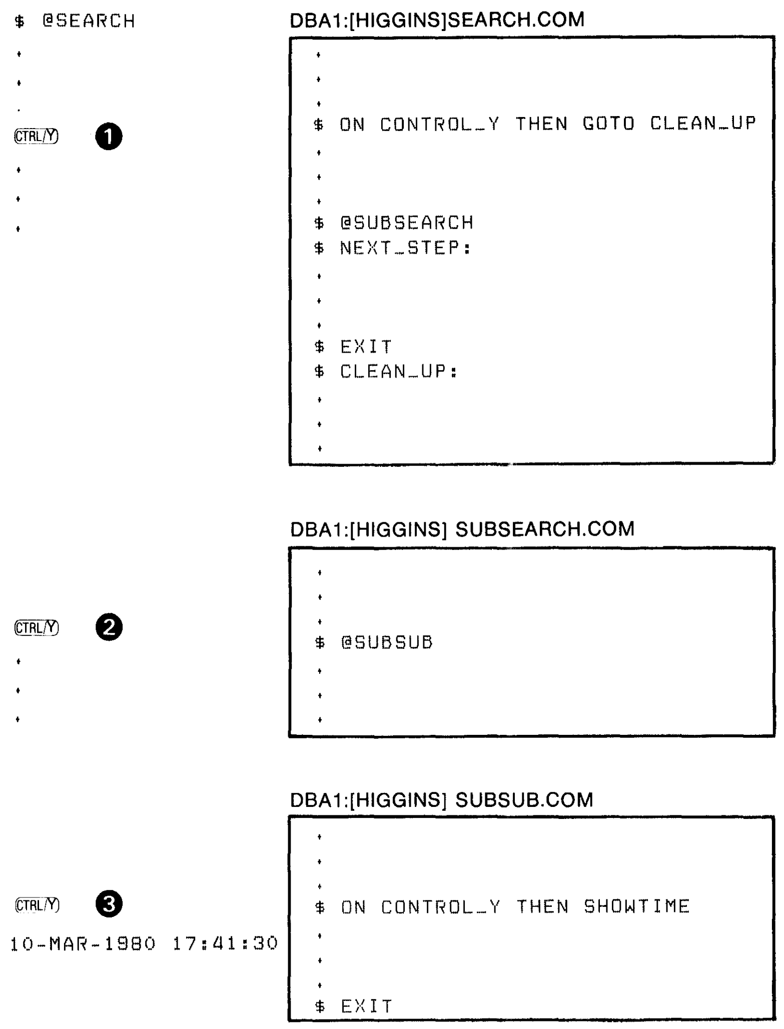
```
.
.
.
$ ON CONTROL_Y THEN WRITE SYS$OUTPUT-
    "Interruption not allowed...continuing"
.
.
.
$ TYPE STATUS.OUT;1  ❺
$ IF $STATUS THEN DELETE STATUS.OUT;1
.
.
.
```

The CTRL/Y interrupt at ❶ occurs during execution of the TYPE command, at ❷. Control is transferred to the label CLEAN__UP. After executing the routine, the command procedure exits, at ❸ and returns control to the interactive command level.

The CTRL/Y interrupt at ❹ occurs during execution of the TYPE command, at ❺. The WRITE command specified in the ON command is executed. Then, the command procedure continues execution at the command following the interrupted command.

Figure 7-2 Flow of Execution Following CTRL/Y Action

A CTRL/Y action can be specified for each active command level; the CTRL/Y action specified for the currently executing command level overrides action(s) specified for previous levels, if any. Note, however, that if a CTRL/Y action is established at a command level, the default action for subsequent command levels is to exit. Figure 7-3 illustrates what happens when CTRL/Y is pressed during the execution of a nested command procedure.



```
$ @SEARCH
.
.
.
CTRL/Y  1
.
.
.
```

DBA1:[HIGGINS]SEARCH.COM

```
.
.
.
$ ON CONTROL_Y THEN GOTO CLEAN_UP
.
.
.
$ @SUBSEARCH
$ NEXT_STEP:
.
.
.
$ EXIT
$ CLEAN_UP:
.
.
.
```

DBA1:[HIGGINS] SUBSEARCH.COM

```
CTRL/Y  2
.
.
.
```

```
.
.
.
$ @SUBSUB
.
.
.
```

DBA1:[HIGGINS] SUBSUB.COM

```
CTRL/Y  3
10-MAR-1980 17:41:30
```

```
.
.
.
$ ON CONTROL_Y THEN SHOWTIME
.
.
.
$ EXIT
```

1. If a CTRL/Y interrupt occurs while SEARCH.COM is executing, control is transferred to the label CLEAN_UP.
2. If a CTRL/Y interrupt occurs while SUBSEARCH.COM is executing, control is transferred to the label NEXT_STEP in SEARCH.COM. Because no CTRL/Y action is specified in SUBSEARCH.COM, the procedure exits to previous command level when a CTRL/Y interrupt occurs.
3. If a CTRL/Y interrupt occurs while SUBSUB.COM is executing, the SHOW TIME command is executed.

Figure 7-3 Default CTRL/Y Action for Nested Procedures

7.2.3 Disabling CTRL/Y Interruptions

CAUTION

The ON CONTROL_Y and SET NOCONTROL_Y commands are intended for special applications. It is not, in general, recommended that you disable CTRL/Y interrupts. For example, if a procedure that disables CTRL/Y interrupts begins to loop uncontrollably, you cannot gain control to stop the procedure from your terminal; you must use another terminal to terminate the procedure or you must request the system operator to terminate it for you. Termination, in this case, requires the deletion of your process.

The SET NOCONTROL_Y command disables CTRL/Y handling completely: that is, if a command procedure executes the SET NOCONTROL_Y command, pressing the CTRL/Y will have no effect.

The SET NOCONTROL_Y command also cancels the current CTRL/Y action, if any, and restores the default. Thus, the correct way to reestablish the default command interpreter action for CTRL/Y handling is to issue the two commands:

```
$ SET NOCONTROL_Y
$ SET CONTROL_Y
```

The first command disables CTRL/Y handling and cancels a current CTRL/Y action; the second command enables CTRL/Y handling. At this point, the default action is reinstated: if CTRL/Y is pressed during the execution of the procedure, the command interpreter prompts for a command at the CTRL/Y command level.

You can issue the SET NOCONTROL_Y command at any command level; it affects all command levels, until the SET CONTROL_Y command reenables CTRL/Y handling.

NOTE

For an example of a system-defined procedure that disables CTRL/Y interrupts for logged-in users, see the sample procedure FORTUSER.COM in Appendix A.

CHAPTER 8

CREATING, READING, AND WRITING FILES

This chapter describes ways you combine DCL commands with the programming and symbolic capabilities of command procedures to manipulate sequential files. Included are techniques for using:

- The OPEN command to create sequential files within a command procedure
- The READ command to read sequential files within a command procedure
- The WRITE command to write sequential files from a command procedure
- The CLOSE command to explicitly close files that have been opened during command procedure execution

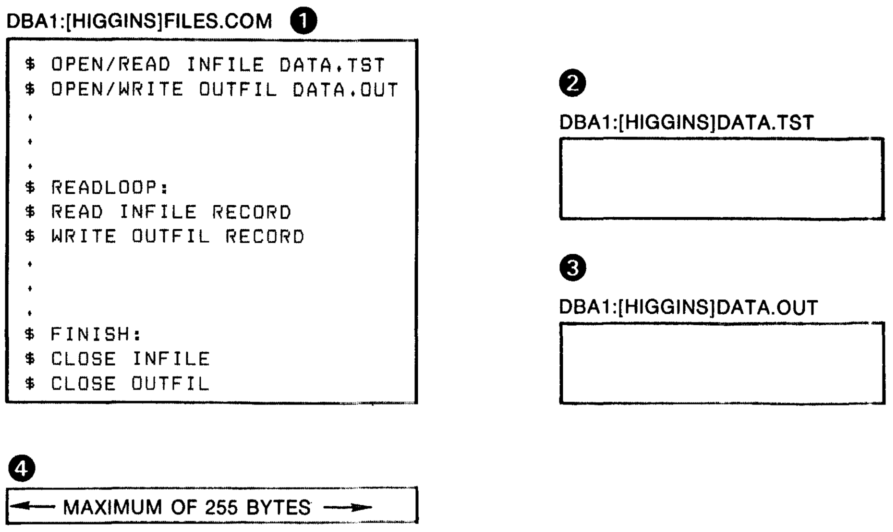
The basic steps in reading and writing files from a command procedure are:

1. Use the OPEN command to open a file. The OPEN command assigns a logical name to the file and specifies whether the file is to be read or written. If you open a nonexistent file for writing, a file will be created. Otherwise, the file specified in the OPEN command must be an existing file.

2. Use the READ or WRITE command to read or write the file. The READ and WRITE commands use command symbols to define buffers for input and output records; the READ command reads a record from a file into a symbol and the WRITE command writes one or more symbols or literal character strings from a symbol into a single record of an output file.

3. Use the CLOSE command to close the file. After you open a file with the OPEN command, it remains open until you explicitly close it or until you log out.

For example, Figure 8-1 shows a command procedure that reads a record from an input file and copies the record into an output file. The OPEN commands in the procedure specify whether the files are opened for input or output, create logical names for the files (for use in subsequent READ and write commands in this procedure), and identify the files. The READ and WRITE commands use the logical names to refer to the files and define a symbol that becomes the input/output buffer for file reads and writes. The CLOSE commands are used to explicitly close both files when the command procedure completes file processing. The CLOSE commands also deassign the logical names specified for the files in the OPEN commands.



```
$ OPEN/READ INFILE DATA.TST
$ OPEN/WRITE OUTFIL DATA.OUT
.
.
.
$ READLOOP:
$ READ INFILE RECORD
$ WRITE OUTFIL RECORD
.
.
.
$ FINISH:
$ CLOSE INFILE
$ CLOSE OUTFIL
```

❶ The command procedure, FILES.COM

❷ The input file, DATA.TST

❸ The output file, DATA.OUT

❹ The contents of this buffer is assigned to the READ and WRITE symbol name, RECORD

Figure 8-1 Steps in Reading and Writing Files

8.1 READING FILES

With the READ command, you can read only sequential files in which all records are less than 255 characters in length. After a file is opened, the command interpreter maintains a pointer to a current record in the file. Each READ command reads the next record and uses the contents of the record to assign a value to the symbol name specified by the command. The following sections discuss two of the factors you must consider when reading files: how to define symbol names and how to handle end-of-file conditions.

8.1.1 Specifying Symbol Names for the READ Command

The rules for specifying symbol names are the same as for defining symbols with assignment statements:

- A symbol name must start with an alphabetic letter, dollar sign ($), or underscore (_)

- A symbol name can have from 1 to 255 characters

When you specify a symbol name for the READ command, the command interpreter places the symbol name in the local symbol table for the current command level. If you use the same symbol name for more than one READ command, each READ command redefines the value of the symbol name. For example, you can use a loop in a command procedure to read an entire file, as shown below:

```
$ READLOOP:
$      READ   INFILE   RECORD
$      GOTO READLOOP
```

Each time through this loop, the READ command reads a record from the input file identified as INFILE and redefines the value of the symbol RECORD.

8.1.2 Handling End-of-File Conditions

When the READ command attempts to read beyond the last record in the file, an error condition indicating the end of file is returned by the VAX-11 Record Management Services (VAX-11 RMS). The completion status value is %RMS-F-EOF. Note that because the command interpreter performs normal error checking and message processing following a READ command, this condition can result in the termination of the command procedure, unless the procedure has established its own error handling.

The READ command allows you to specify, with the /END_OF_FILE qualifier, the label of a line in the command procedure to be given control when this completion value is returned. For example:

```
$ LOOP:
$       READ/END_OF_FILE=DONE  INFILE  RECORD
$       GOTO LOOP
$ DONE:
$       CLOSE INFILE
```

In this example, the procedure executes the READ command repeatedly until the end-of-file status is returned. Then, control is given to the line labeled DONE. Note that labels you specify for /END_OF_FILE qualifiers are subject to the same rules as labels specified for a GOTO command and are located in the same way.

8.2 WRITING FILES

The WRITE command can write records only to sequential files that have been opened for writing. If the output file specification on an OPEN/WRITE command does not include a file version number and if there already exists a file with the specified file name and file type, the WRITE command creates a new file with a version number one greater than the existing file. Thus, you cannot use the WRITE command to append records to an existing file.

When the WRITE command writes a record, it always positions a record pointer following the record just written.

8.2.1 Symbol Substitution in the WRITE Command

As shown in Figure 8-1, the WRITE command automatically performs symbol substitution on tokens specified as parameters. This applies to all tokens that begin with alphabetic letters and are not enclosed in quotation marks.

To specify more than one symbol name, separate them with commas. You can intersperse symbol names and literal character strings within a WRITE command. For example:

```
$ WRITE OUTFILE "Count is ",COUNT,"."
```

This WRITE command writes one data record into the output file identified by the logical name OUTFILE. If the current value of the symbol COUNT is 4, the data record that is written is:

```
Count is 4.
```

Another way to mix literal strings with symbol names is to place the entire string within quotation marks and use double apostrophes to request symbol substitution. For example:

```
$ WRITE OUTFILE "Count is ''COUNT'."
```

This WRITE command is equivalent to the preceding WRITE command example.

Note that symbol substitution performed by the WRITE command is actually performed during command execution. The WRITE command does not perform lexical processing on the result of substitution. For example, a symbol could be defined with a lowercase value as follows:

```
$ LINE := "This is the line."
```

The WRITE command will not convert this line to uppercase when it processes the symbol LINE.

NOTE

The sample procedure LISTER.COM in Appendix A illustrates the WRITE command.

If you use apostrophes or ampersands to request symbol substitution in a parameter specified for the WRITE command, recursive substitution occurs. For example, if you use a lexical function in a WRITE command as shown below, an error occurs:

```
$ WRITE SYS$OUTPUT 'F$MODE()'
```

You must place the function in quotation marks:

```
$ WRITE OUTFILE "''F$MODE()'"
```

Otherwise, the replacement of the function F$MODE would occur during command input causing the WRITE command to attempt substitution on the resulting symbol name.

Figure 8-2 illustrates different ways of specifying data for the WRITE command.

```
.
.
.
$ @FIGURE

$ TYPE DATA.OUT
the character string
ABC
COUNT IS 4
MODE IS INTERACTIVE
FOURTH PARAMETER
$
```

DBA1:[HIGGINS]FIGURE.COM

```
  $ ABC := "the character string"
  $ COUNT = 4
  $ P4 := fourth parameter
  $ OPEN/WRITE OUTFILE DATA.OUT
❶ $ WRITE OUTFILE ABC
❷ $ WRITE OUTFILE "ABC"
❸ $ WRITE OUTFILE "COUNT IS ",COUNT
❹ $ WRITE OUTFILE "MODE IS ''F$MODE()'"
❺ $ WRITE OUTFILE P'COUNT'
  $ CLOSE OUTFILE
```

❶ The WRITE command automatically performs symbol substitution on characterstrings that are not enclosed in quotation marks; substitution is not recursive.

❷ If a character string is enclosed in quotation marks, the WRITE command does not perform symbol substitution.

❸ When two or more symbol names or character strings are specified, the WRITE command concatenates the strings before it writes the record to the output file.

❹ Within character strings, the command interpreter performs substitution requested by apostrophes during command input; the WRITE command executes the results.

❺ If the data specified for a WRITE command contains an apostrophe, the command interpreter performs symbol substitution during command input (as in ❹); the WRITE command performs substitution on the resulting command string.

Figure 8-2 Symbol Substitution with the WRITE Command

8.3 ERROR HANDLING

The ON command, described in Chapter 7, provides a default course of action when errors occur during the execution of a command procedure. You can use an ON condition action to control what happens overall in a procedure. The OPEN, CLOSE, READ, and WRITE commands also allow you to specify labels to receive control in case an error occurs during the processing of the specific command.

For example:

```
$ OPEN/READ/ERROR=NOT_FOUND INFILE CONTINGEN.DOC
```

This OPEN command requests that the file named CONTINGEN.DOC be opened for reading. If the file cannot be opened for any reason, for example, if it does not exist, the OPEN command returns an error condition. Control is transferred to the label NOT_FOUND.

This example also illustrates a technique for determining whether a file with a given file name and file type already exists. The label NOT_FOUND could be a successful path for a command procedure that creates a file named CONTINGEN.DOC and wants to ensure that the file name and type are unique. For example:

```
$       OPEN/READ/ERROR=NOT_FOUND FILE 'P1'
$       WRITE SYS$OUTPUT "File name ''P1' in use"
$       CLOSE FILE
$       EXIT
$ NOT_FOUND:
$       OPEN/WRITE OUTFILE 'P1'
```

This procedure uses the OPEN command with the /ERROR qualifier to open a file whose name is specified by a parameter passed to the procedure. If the OPEN command completes execution successfully, the file already exists; the procedure issues a message, closes the file, and exits.

If the OPEN command fails, the procedure assumes that the command failed because the file does not exist; the procedure takes the error path and control goes to the label NOT_FOUND. At NOT_FOUND, the file is opened for writing. If the error occurred for some other reason, for example, an invalid file type, the OPEN/WRITE command to write the file will fail and an appropriate error message will be issued.

Any error path specified with the file-handling commands (OPEN, READ, WRITE, and CLOSE) overrides the current ON condition established for the command level. Moreover, an error path, successfully taken, changes the value of $STATUS to a success code. Thus, the procedure cannot, in this case, determine the specific reason for the error. The following examples illustrate this aspect of error handling.

```
$       ON ERROR THEN GOTO CHECK
$       OPEN/READ INFILE 'P1'
    .
    .
    .
$ CHECK:
$       WRITE SYS$OUTPUT "Error opening file:  status is ",$STATUS
```

In the above example, if an error occurs during opening of the file specified by P1, control goes to the label CHECK. At CHECK, $STATUS still contains the numeric status value associated with the specific error that occurred.

```
$       OPEN/READ/ERROR=CHECK FILE 'P1'
    .
    .
    .
$ CHECK:
$       WRITE SYS$OUTPUT "Error opening file"
```

In this example, if an error occurs opening the file, control goes to the label CHECK as a result of the /ERROR qualifier. However, at this label, the value of $STATUS is always a success code; the procedure cannot check for or display the specific status value that caused the error.

8.4 COMMUNICATING WITH PROCESS-PERMANENT FILES

You can also use the READ and WRITE commands to read data from the current input device or to write messages on the current output device. The process permanent files SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, and SYS$ERROR do not have to be explicitly opened before you refer to them in READ or WRITE commands. For example:

```
    $ READ SYS$COMMAND TESTID
```

This READ command results in a read to the current device SYS$COMMAND; thus, when the procedure is executed interactively, the read is issued to the terminal. When the READ command executes, the command interpreter displays the following prompt at the terminal:

```
    Data:
```

Whatever you type in response to this prompt is then equated to the symbol named TESTID.

Similarly, you can write a line of data to the terminal, or whatever the current output device is, with the WRITE command. For example:

```
    $ WRITE SYS$OUTPUT "Count is ''COUNT'...  continuing..."
```

Before this line is displayed on the terminal, the symbol named COUNT is replaced with its current value.

Note that the logical name TT the system equates to the current interactive terminal is not a process permanent file. To write to TT, you must explicitly open it.

8.5 FILE FORMATS

You can use the READ command to read any existing sequential file. The maximum record length that the READ command accepts is 255 characters.

When you create a file with the WRITE command, you cannot specify any attributes for the file: the command interpreter always creates a file in print file format. The record format for the file is VFC, with a 2-byte header for each record.

These files are therefore not compatible with files created by the default system editor SOS or with RSX-11M utilities invoked by DCL commands. If you create a file with the DCL command WRITE and you want to use the file as input to another program or command, you can perform an intermediate step to convert the file to a suitable format. One simple way to do this is to invoke the SOS editor to edit the file and then write the file back onto disk. SOS removes the carriage control bytes from each record as it writes the output file. For example:

```
    $ OPEN/WRITE OUTFILE DATA.OUT
        .
        .
        .
    $ CLOSE OUTFILE
    $ EDIT/NOLINES DATA.OUT
    EB
```

After a file is closed, you can specify it as an input file to the editor; the /NOLINES qualifier in this example indicates to SOS that the file does not have line numbers associated with the records in the file. The SOS command EB follows the EDIT command in the input stream for the procedure: the EB command writes the file onto disk without incrementing the version number.

CHAPTER 9

CONTROLLING BATCH JOBS

This chapter describes techniques for controlling batch jobs. It includes information useful both to batch users and to interactive users who submit command procedures to a batch job queue.

The following topics are discussed in this chapter:

- How the system executes batch jobs
- Batch job output
- Synchronizing batch job execution

9.1 HOW THE SYSTEM EXECUTES BATCH JOBS

When the system executes a command procedure submitted to a batch job queue, it creates a detached process to execute the commands. This process receives your disk and directory defaults and the same resource quotas and privileges that were given to your interactive process when you logged in. This process is given a name of the form _JOBnnn where nnn is the job number assigned to the job. The process executes under your UIC. Figure 9-1 illustrates how the system executes a batch job.

9.1.1 The Batch Job Queue

Once a job has been entered in a batch job queue, you can monitor its status with the SHOW QUEUE command. For example:

```
$ SHOW QUEUE SYS$BATCH
```

This command would show the current contents of the SYS$BATCH queue. The following command would show the current contents of all batch queues:

```
$ SHOW QUEUE /BATCH/ALL
```

All jobs in batch queues have job numbers, but no job in a queue has a process created for its execution until the job becomes a current job. Thus, jobs identified in batch queue displays as "current" jobs are active processes; jobs identified as "pending" jobs or "holding" jobs are in the queues, but processes have yet to be created for them.

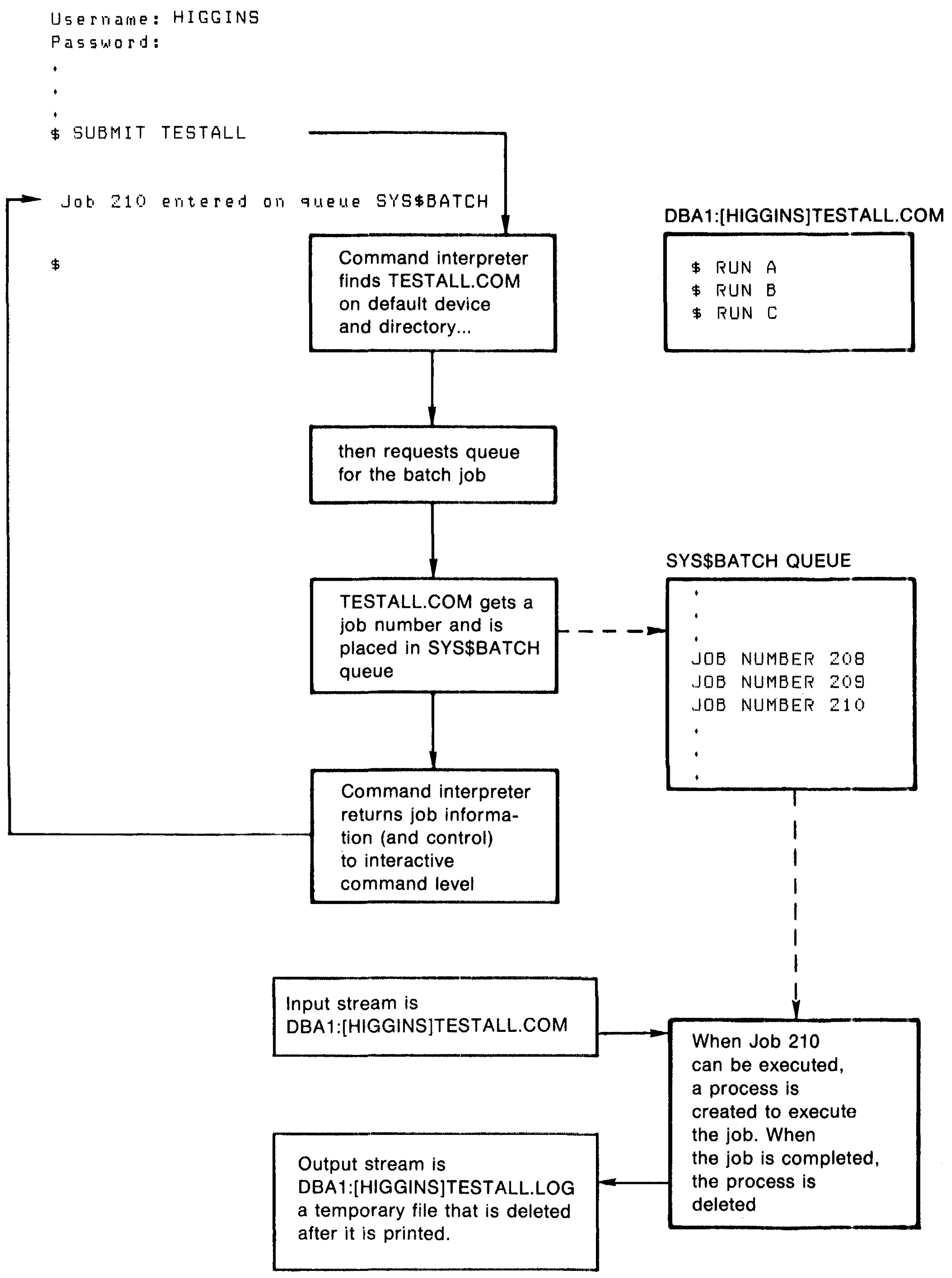

Username: HIGGINS
Password:
.
.
.
$ SUBMIT TESTALL
Job 210 entered on queue SYS$BATCH
$
Command interpreter finds TESTALL.COM on default device and directory...
then requests queue for the batch job
TESTALL.COM gets a job number and is placed in SYS$BATCH queue
Command interpreter returns job information (and control) to interactive command level
DBA1:[HIGGINS]TESTALL.COM
$ RUN A
$ RUN B
$ RUN C
SYS$BATCH QUEUE
JOB NUMBER 208
JOB NUMBER 209
JOB NUMBER 210
Input stream is DBA1:[HIGGINS]TESTALL.COM
When Job 210 can be executed, a process is created to execute the job. When the job is completed, the process is deleted
Output stream is DBA1:[HIGGINS]TESTALL.LOG a temporary file that is deleted after it is printed.

Figure 9-1 How the System Executes a Batch Job

9.1.2 Controlling Jobs in the Batch Job Queue

After a job has been submitted to the queue, there are actions you can take to control whether the job is executed, when it is executed, and so on. The following paragraphs summarize some of the actions you can take and the commands you would use to perform particular actions.

To change the name of a job after it has been queued but before the system begins processing it, use:

```
$ SET QUEUE/ENTRY=nnn/NAME=newname queue-name
```

To change the processing priority of a batch job, use one of the following:[1]

```
$ SET QUEUE/ENTRY=nnn/PRIORITY=new-priority queue-name

$ SET PROCESS/PRIORITY=new-priority JOBnnn
```

To delay processing of a batch job until a specific date and/or a specific time of day, use of one of the following:

```
$ SUBMIT/AFTER=date-time file-spec

$ JOB username /AFTER=date-time

$ SET QUEUE/ENTRY=nnn/AFTER=date-time queue-name
```

To delay processing of a batch job for an indefinite period of time, use:

```
$ SUBMIT/HOLD file-spec
```

To delete an entry from a batch job queue, either before it is processed or while it is being processed, use:

```
$ DELETE/ENTRY=nnn queue-name
```

To delete an entry from a batch job queue while it is being processed, use:[2]

```
$ STOP/ENTRY=nnn queue-name
```

To give a batch job a specific processing priority with respect to other processes in the system, use one of the following:[3]

```
$ JOB username /PRIORITY=priority

$ SUBMIT/PRIORITY=priority file-spec
```

To release for processing a batch job that is being held in a queue, use:

```
$ SET QUEUE/ENTRY=nnn/RELEASE queue-name
```

---

1. ALTPRI privilege required to raise the priority for a job.

2. May require GROUP or WORLD privilege.

3. OPER privilege required to enter a job at a higher priority.

To specify a name for a batch job, overriding the default job name assigned by the system, use:

```
$ SUBMIT/NAME=new-name file-spec

$ JOB username /NAME=new-name
```

To name a specific batch job queue in which the batch job should be entered, use one of the following:

```
$ SUBMIT /QUEUE=queue-name file-spec

$ JOB username /QUEUE=queue-name
```

To specify a lower CPU time limit than that established by the system manager for jobs in the particular queue, use one of the following:

```
$ JOB username /CPUTIME=n

$ SUBMIT/CPUTIME=n file-spec
```

To specify a lower working set quota than that established by the system manager for jobs in the particular queue, use one of the following:

```
$ JOB username /WSQUOTA=n

$ SUBMIT/WSQUOTA=n file-spec
```

NOTE

The procedures BWAKE.COM and ENDED.COM in Appendix A illustrate techniques that provide for communication between a batch job and its submitter.

9.1.3 Concatenating Procedures into a Single Job

When you issue the SUBMIT command, you can specify that more than one command procedure is to be executed in one job. For example:

```
$ SUBMIT ALPHA,BETA
```

This SUBMIT command concatenates the procedures ALPHA.COM and BETA.COM and executes them as if they consisted of a single input stream. ALPHA.COM is executed, and if it completes without an error or severe error, BETA.COM is executed.

When two or more procedures are submitted this way, the operating context of the first procedure is preserved for the second procedure; that is, local symbols defined at command level 0 continue to exist, the current ON condition action remains in effect, and so on.

9.2 BATCH JOB OUTPUT

When a batch job is executed, its output stream consists of messages written to SYS$OUTPUT and SYS$ERROR. This output stream is equated to a batch job log file. The system locates this file in your default directory, giving it a file specification of name.LOG, where name is the job name. By default, the job name is taken from the first eight characters of the file name of the command procedure. However, you can use the /NAME qualifier on the SUBMIT command to define an alternate name for the job. In either case, the system automatically queues the log file for printing when the batch job is completed and deletes the file from your directory after it is printed.

The batch job log file includes, by default, all command lines executed in the command procedure, system and user-program output to SYS$OUTPUT and SYS$ERROR, and job termination accounting information. The job termination information is equivalent to the long form of the system logout message.

9.2.1 Including All Command Output in the Batch Job Log

Typically, a batch job that compiles, links, and executes a program creates additional printed output: a compiler listing, for example, and often a linker map file. To produce printed copies of these files, a batch job can contain the PRINT command(s) necessary to print them, as in the following example:

```
$ FORTRAN BIGCOMP
$ PRINT BIGCOMP
$ LINK/MAP/FULL BIGCOMP
$ PRINT BIGCOMP.MAP
```

When this batch job completes processing, there are three separate output listings: the batch job log, the compiler listing, and the linker map.

If you want a batch job log to contain all output from the command procedure, including printed listings of compiler or linker output files, you can do either of the following:

- Use the TYPE command instead of the PRINT command in the command procedure. The TYPE command writes to SYS$OUTPUT, in this case, the batch job log.
- Use qualifiers on appropriate commands to direct the output to the current output device.

The following example shows the latter technique:

```
$ FORTRAN/LIST=SYS$OUTPUT  BIGCOMP
$ LINK/MAP=SYS$OUTPUT/FULL BIGCOMP
```

When these commands are executed in a batch job, the output files from the compiler and the linker are written directly to the batch job log. Note that if you use this technique, the output file(s) are not saved on disk.

9.2.2 Saving the Batch Job Log File

Normally, a batch job log file is written as a job is executed. When the job has been executed, the system closes the file and queues it for printing with the delete option, so that the file will be deleted from your directory after it has printed.

If you are an interactive user, however, situations will arise in which you would like either to save the output from a batch job in a disk file, or to not print the output but to examine it from your terminal. To suppress the printing and deleting of the batch job output, you must assign a dummy equivalence name for the logical name SYS$PRINT. For example:

```
$ ASSIGN DUMMY SYS$PRINT
```

The name DUMMY is any name that is not a device or queue name. The system always tries to queue the log file to the device queue named SYS$PRINT. If SYS$PRINT is equated to an invalid name (or a name that is not the name of a valid queue), the file cannot be printed.

Note that you can use the same technique to direct the batch job output log to a specific line printer. For example, if you want to ensure that the log file is printed on the printer named LPB0, you could include the following logical name assignment in the batch job log:

```
$ ASSIGN LPB0:  SYS$PRINT
```

9.2.3 Terminating a Batch Job Abnormally

A batch job terminates normally as a result of:

- The end-of-file or an EXIT command at command level 0
- A STOP or LOGOUT command at any command level

When a job terminates, that is, when there are no more files in the job to be processed, the system deletes the process that was created to execute the job. During the termination procedure, the log file is printed.

Note that the batch job log file is not printed, and is not deleted, if the job terminates as a result of a DELETE/ENTRY or STOP command: it remains in your default directory.

To use the DELETE/ENTRY command, specify the job number of the job to be deleted. For example:

```
$ DELETE/ENTRY=312 SYS$BATCH
```

To use the STOP command, you must specify the full process name assigned to the job. For example:

```
$ STOP _JOB312
```

The STOP command requires the user privilege GROUP to control other processes in the same group or the user privilege WORLD to control other processes not in the same group.

CAUTION

Terminating jobs using either of these commands is considered abnormal termination because the operating system's normal job termination activity is preempted. The batch job log does not, for example, contain the standard logout message that summarizes job time and accounting information. However, termination that results from an explicit EXIT or STOP command in the procedure or the implicit execution of either of these commands following an error condition based on the current ON condition is considered normal termination, since the operating system can perform proper run-down and accounting procedures.

9.3 SYNCHRONIZING BATCH JOB EXECUTION

The SYNCHRONIZE and WAIT commands both place a job in a wait state: the SYNCHRONIZE command waits for the completion of another job, while the WAIT command waits for a specified period of time to elapse. For example, if jobs are submitted concurrently to perform cooperative functions, one job can contain the command:

```
$ SYNCHRONIZE BATCH25
```

After this command is executed, the command procedure cannot continue execution until the job identified by the job name BATCH25 completes execution. Figure 9-2 shows an example of command procedures that are submitted for concurrent execution, but which must be synchronized for proper execution. Each procedure compiles a large source program.

```
.
.
.
$ SUBMIT MAINCOMP
  JOB 314 entered on queue SYS$BATCH ❶
$ SUBMIT MINCOMP
  JOB 315 entered on queue SYS$BATCH ❶
.
.
.
```

DBA1:[HIGGINS]MAINCOMP.COM

```
   $ FORTRAN ALPHA/LIST
❷  $ SYNCHRONIZE MINCOMP
   $ LINK/MAP/FULL ALPHA,BETA
   $ EXIT
```

DBA1:[HIGGINS] MINCOMP.COM

```
$ FORTRAN BETA/LIST
$ EXIT
```

❶ Individual SUBMIT commands are required to submit two separate jobs. Two separate processes will be created.

❷ After the FORTRAN command is executed, the SYNCHRONIZE command is executed. If job 315 has completed execution, job 314 continues with the next command. However, job 314 will not execute the next command, if job 315 is either current or pending.

Figure 9-2 Synchronizing Batch Job Execution

Job names specified for the SYNCHRONIZE command must be for jobs that are executing with the same group number in their user identification codes (UICs). To synchronize with a job that has a different group number (for example, that was submitted by a different user), you must use the jobid. For example:

```
     $ SYNCHRONIZE/ENTRY=454
```

This SYNCHRONIZE command places the current command procedure in a wait state until job 454 completes.

The WAIT command is useful for command procedures that must have access to a shared system resource, for example, a disk or tape drive. The following example shows a procedure that requests the allocation of a tape drive; if the command does not complete successfully, the procedure will place itself in a wait state. After a 5-minute interval, it retrys the request:

```
     $ TRY:
     $      ALLOCATE DM:  RK:
     $      IF $STATUS THEN GOTO OKAY
     $      WAIT 00:05
     $      GOTO TRY
     $ OKAY:
     $ REQUEST/REPLY/TO=DISKS -
          "Please mount BACK_UP_GMB on ''F$LOGICAL("RK")'"
               .
               .
               .
```

The IF command following the ALLOCATE request checks the value of $STATUS. If the value of $STATUS indicates successful completion, the command procedure will continue. Otherwise, the procedure issues the WAIT command; the WAIT command specifies a time interval of five minutes. After waiting five minutes, the next command, GOTO, is executed, and the request is repeated. This procedure continues looping and attempting to allocate a device until it succeeds or until the batch job is deleted or stopped.

APPENDIX A

ANNOTATED COMMAND PROCEDURES

This appendix contains complete command procedures that demonstrate the concepts and techniques discussed in Chapters 1 through 9. Each section in this Appendix discusses one command procedure and contains the following:

- The name of the procedure
- A listing of the procedure
- Notes that explain concepts or techniques used by the procedure.
- The results of a sample execution of the procedure

The command procedures are:

CONVERT.COM	Section A.1

This procedure converts an absolute time value (for example, 10:45) to a delta time value (for example 01:05). The procedure illustrates use of the F$TIME and F$EXTRACT lexical functions and the use of assignment statements to perform arithmetic calculations and to concatenate symbol values.

WAKEUP.COM	Section A.2

This procedure places the current interactive user process in a wait state until a specific time of day. Then, it displays a message on the terminal indicating the time. The procedure illustrates use of the INQUIRE and WRITE commands and how command levels communicate using a global symbol.

BWAKE.COM	Section A.3

When submitted to the batch job queue, this procedure sends a message to the submitter at a particular time of day. This procedure illustrates use of the REPLY command (OPER privilege required) and the group logical name table (GRPNAM privilege required) to provide communication between a batch job and the submitter of a batch job.

ENDED.COM	Section A.4

This procedure sends a message to a specified terminal indicating the end of a batch job. ENDED.COM illustrates the F$MESSAGE and F$PROCESS lexical functions. Like the BWAKE.COM procedure, ENDED.COM demonstrates a technique for a batch job to communicate with its submitter.

GETPARMS.COM Section A.5

This procedure returns the number of parameters that were passed to a procedure. Note that this procedure must be defined as a command synonym; it can be called from any procedure.

EDITALL.COM Section A.6

This procedure invokes the SOS editor repeatedly to edit a group of files with the same file type. This procedure illustrates how to use lexical functions to extract file names from columnar output. It also illustrates a way to redefine the input stream for a program invoked within a command procedure.

FORTUSER.COM Section A.7

This example of a system-defined login file provides a controlled terminal environment for an interactive user who creates, compiles, and executes FORTRAN programs. This procedure illustrates using lexical functions to step through an option table, comparing a user-entered command with a list of valid commands.

LISTER.COM Section A.8

This is a procedure that prompts for input data, formats the data in columns, and then sorts it into an output file. This procedure illustrates the READ and WRITE commands, as well as the character substring overlay format of an assignment statement.

CALC.COM Section A.9

This procedure performs arithmetic calculations and converts the resulting value to hexadecimal and decimal values. CALC.COM illustrates the F$CVUI lexical function and the arithmetic overlay syntax of the assignment statement.

A.1 CONVERT.COM

```
  $ ! Check for inquiry
  $ !
❶ $ IF P1 .EQS. "?" .OR. P1 .EQS. "" THEN GOTO TELL
  $ !
  $ ! Verify the parameter:  it must be 5 characters long
  $ !                        it must contain a colon (:)
  $ !
❷ $ IF 'F$LENGTH(P1)' .NE. 5 .OR. -
        'F$LOCATE(":",P1)' .NE. 2 -
        THEN GOTO BADTIME
  $ !
❸ $ TIME := "''F$TIME()'"                         ! Get the current time
  $ !
  $ ! Extract the hour and minute fields from both the current time
  $ ! value and the specified absolute time value
  $ !
❹ $ MINUTES := 'F$EXTRACT(15,2,TIME)'      ! Current minutes
  $ HOURS := 'F$EXTRACT(12,2,TIME)'        ! Current hours
  $ ABS_HOURS := 'F$EXTRACT(0,2,P1)'       ! Hours in absolute time
  $ ABS_MINUTES = 'F$EXTRACT(3,2,P1)'      ! Minutes in absolute time
  $ !
  $ ! Verify that the values are in correct range of 24-hour clock
  $ !
❺ $ IF ABS_HOURS .GT. 23 .OR. ABS_MINUTES .GT. 59 -
        THEN GOTO BADTIME
  $ !
  $ ! Convert both time values to minutes
  $ !
❻ $ CURRENT_TIME = HOURS*60 + MINUTES
  $ ABS_TIME = ABS_HOURS*60 + ABS_MINUTES
  $ !
  $ ! Compute difference in hours and minutes
  $ !
  $ !
❼ $ MINUTES_TO_WAIT = ABS_TIME - CURRENT_TIME
  $ !
  $ ! If the result is <0 the time is assumed to be a
  $ ! tomorrow time; more calculation is required.
  $ !
❽ $ IF MINUTES_TO_WAIT .LT. 0 THEN -
        MINUTES_TO_WAIT = 24*60 - CURRENT_TIME + ABS_TIME
  $ !
  $ ! Start looping to determine the value in hours and minutes from
  $ ! the value expressed all in minutes
  $ !
  $     HOURS_TO_WAIT = 0
❾ $ HOURS_TO_WAIT_LOOP:
  $     IF MINUTES_TO_WAIT .LT. 60 THEN GOTO FINISH_COMPUTE
  $     MINUTES_TO_WAIT = MINUTES_TO_WAIT - 60
  $     HOURS_TO_WAIT = HOURS_TO_WAIT + 1
  $     GOTO HOURS_TO_WAIT_LOOP
  $ FINISH_COMPUTE:
  $ !
  $ ! Construct the delta time string in the proper format
  $ !
❿ $ WAIT_TIME :== 'HOURS_TO_WAIT':'MINUTES_TO_WAIT':00.00
  $ !
  $ ! Examine the second parameter
  $ !
⓫ $ IF P2 .EQS. "SHOW" THEN SHOW SYMBOL WAIT_TIME
  $ !
```

```
   $ ! Normal exit
   $ !
   $ EXIT
   $ !
   $ ! Exit taken if first parameter is not formatted correctly
   $ !
⓬ $ BADTIME:
   $ WRITE SYS$OUTPUT "Invalid time value: ''P1' format must be hh:mm"
   $ EXIT
   $ !
   $ ! Output message and exit if user enters inquiry
   $ !
⓭ $ TELL: TYPE SYS$INPUT
         Converts an absolute time value to a delta time value.
         On return, the global symbol WAIT_TIME contains the
         converted time value.  If you enter the keyword SHOW
         as the second parameter, the procedure displays the
         resulting value in the output stream.
         The format is:

                      @CONVERT hh:mm [SHOW]
   $ EXIT
```

Notes

❶ The procedure checks whether the value entered for a parameter is the question mark character (?) or whether a parameter was entered. If either condition exists, the procedure will branch to the label TELL (Note 13).

❷ The procedure checks the value of the parameter. It must be a time value in the format:

```
hh:mm
```

The IF command checks (1) that the length of the entered value is 5 characters and (2) that the third character (offset of 2) is a colon. The IF command contains the logical OR operator: if either expression is true (that is, if the length is not 5 or if there is not a colon in the third character position), the procedure will branch to the label BADTIME (Note 12).

❸ The F$TIME lexical function places the current time value in the symbol TIME. The quotation marks around the lexical function ensure that if there is a leading blank in the time field the command interpreter will not suppress it.

❹ The F$EXTRACT function extracts, from the saved current time value and from the absolute time value entered as a parameter, the minutes and hours fields of each. The string functions return character strings representing decimal values. These values can now be manipulated arithmetically.

❺ The IF command verifies that the time value entered is a valid 24-hour clock time. If the value of the hours field is greater than 23 or if the value of the minutes field is greater than 59, the procedure will branch to the label BADTIME (Note 12).

❻ Arithmetic assignment statements convert both the current time value and the entered time value to minutes by multiplying the hours by 60 and adding the minutes.

❼ The procedure then subtracts the current time from the specified time in minutes.

❽ If the result is less than 0, the time will be after 24:00, that is, on the next day. In this case, the procedure calculates the minutes to wait by subtracting the current time from 24 hours to find the time remaining in the current day and then adding the entered time.

❾ The procedure enters a loop in which it calculates, from the value of MINUTES_TO_WAIT, the number of hours. Each time through the loop, it checks whether MINUTES_TO_WAIT is greater than 60. If so, it will subtract 60 from MINUTES_TO_WAIT and add 1 to the accumulator for the number of hours (HOURS_TO_WAIT).

❿ When the procedure exits from the loop, it concatenates the hours and minutes values into a time string. The symbols HOURS_TO_WAIT and MINUTES_TO_WAIT are replaced by their current values and separated with an intervening colon. The symbol WAIT_TIME has the delta time value. WAIT_TIME is defined as a global symbol so that it will not be deleted when the procedure WAIT_TIME exits.

⓫ If a second parameter, SHOW, was entered, the procedure will display the resulting time value. Otherwise, it will exit.

⓬ At the label BADTIME, the procedure displays an error message that shows the incorrect value entered as well as the format it requires. After issuing the error message, CONVERT.COM exits.

⓭ At the label TELL, the procedure displays information about what the procedure does. The next command in the procedure (EXIT) is also the end-of-file for the input data stream that the TYPE command is reading.

Sample Execution

```
$ SHOW TIME<RET>
10-APR-1979 10:38:26
$ @CONVERT 12:00 SHOW<RET>
   WAIT_TIME = 1:22:00.00
```

The SHOW TIME command displays the current date and time. CONVERT.COM is executed with the parameters 12:00 and SHOW. The procedure converts the absolute time 12:00 to a delta time value and displays it on the terminal.

A.2 WAKEUP.COM

```
❶ $ SAVE_VERIFY ='F$VERIFY("NO")'
  $ !
  $ ! Places the current process in a wait state until a specified
  $ ! absolute time.  Then, it rings the bell on the terminal and
  $ ! displays a message.
  $ !
  $ ! Prompt for absolute time
  $ !
❷ $ INQUIRE WAKE_TIME "Enter time to wake"
  $ !
  $ ! Call the CONVERT.COM procedure to convert the absolute time
  $ ! to a delta time
  $ !
❸ $ @CONVERT 'WAKE_TIME'
  $ !
  $ WAIT 'WAIT_TIME'                  ! Wait the specified delta time
  $ !
❹ $ BELL[0,32]= %X07                  ! ASCII code for terminal bell
  $ !
  $ WRITE SYS$OUTPUT BELL,"Wake up:  Time is ''F$TIME()'"
  $ !
❺ $ IF SAVE_VERIFY THEN SET VERIFY    ! Restore verification, if set
  $ EXIT
```

Notes

❶ The procedure saves the current verification setting in the symbol name SAVE_VERIFY, then sets verification off.

❷ The procedure uses the INQUIRE command to prompt for the time desired to wake up the process. The value entered in response to INQUIRE is used as input to the CONVERT procedure.

❸ The CONVERT procedure returns the delta time value corresponding to the interval from the current time until wake up time in the global symbol WAIT_TIME. The procedure uses this symbol to specify the time parameter for the WAIT command.

❹ After the specified period elapses, the process awakes and the procedure constructs a message to display on the terminal. It gives the symbol named BELL the binary value of the bell character on an ASCII terminal. The WRITE command automatically replaces this symbol with its value and concatenates the result with a literal character string. The terminal's bell rings as the message is displayed.

❺ The IF command tests whether the symbol VERIFY has a true value; if so, verification was in effect before the procedure was invoked and will be restored. Otherwise verification will remain off. Note that the EXIT command will be displayed on the terminal if verification was on before the command procedure was executed.

Sample Execution

```
$ SHOW TIME RET
 10-APR-1979 10:39:12
$ @WAKEUP RET
Enter time to wake: 11:30 RET
Wake up -- Time is 10-APR-1979 11:30:00.00
```

The procedure prompts for a time value. Then, the terminal is in a wait state until the time elapses. The terminal's bell (if there is one) sounds when the message is displayed.

Note that if you execute this procedure, you can terminate the wait state of the terminal at any time by pressing CTRL/Y and issuing any DCL command.

A.3 BWAKE.COM

```
  $ IF P1 .EQS. "?" THEN GOTO TELL
  $ !
❶ $ ! This procedure will not execute interactively because it
  $ ! would cause a logout
  $ !
❷ $ IF "''F$MODE()'" .EQS. "INTERACTIVE" THEN GOTO TELL
❸ $ IF P1 .EQS. "" THEN GOTO NO_WAKE
  $ !
  $ ! Use the CONVERT.COM procedure to convert the absolute time
  $ ! to a delta time
  $ !
❹ $ @CONVERT 'P1'
  $ !
❺ $ WAIT 'WAIT_TIME'
  $ REPLY/BELL/TERMINAL='F$LOGICAL("TERMINAL")'      "Wake Up"
❻ $ EXIT                                     ! Terminate the procedure
  $ !
❼ $ NO_WAKE:
  $ REPLY/BELL/TERMINAL='F$LOGICAL("TERMINAL")' -
          "No wake up is queued; specify a parameter"
  $ !
  $ EXIT                                     ! Terminate the procedure
  $ !
❽ $ TELL:
  $ CREATE SYS$COMMAND:
          Provides a wake up request at a specified absolute time.   Th
          procedure   must   be   submitted   to   a   batch   job   queue   fo
          execution, using the command format:

          SUBMIT BWAKE/PARAM=hh:mm

          Requires the OPER AND GRPNAM privileges.  There also must be
          group  logical  name table entry for the logical name TERMINA
          (the terminal to which the wake up message is to be sent).
❾ $ EXIT
```

Notes

❶ This procedure must be executed in a batch job. However, it allows interactive execution if a question mark is passed as a parameter. If so, it will branch to the label TELL (Note 8).

❷ Only the question mark parameter is valid in interactive mode; this second IF command checks whether the procedure is being executed interactively. If so, it will branch to the label TELL (Note 8).

❸ The next IF command checks that a parameter was entered. If not, the procedure will branch to the label NO_WAKE (Note 7).

❹ At this point in the procedure, all checks are complete: the procedure is executing in a batch job, and a parameter other than a question mark was entered. The procedure CONVERT.COM is called to perform additional checking on the format of the parameter. If the parameter is a valid time value, CONVERT.COM will convert it to a delta time value and place the delta time value in the global symbol WAIT_TIME.

❺ The batch job issues the WAIT command to wait until the delta time value elapses. When the time is up, the procedure issues the REPLY command.

The REPLY command displays a message on the terminal whose logical name is TERMINAL.

This name must be in the group logical name table for the group of the submitter of the batch job. Because the batch job executes in the same group as the submitter, the logical name is accessible to the batch job. The REPLY command contains the lexical function F$LOGICAL; the function is performed before the command is executed. Note that the submitter of the batch job can place this name in the group logical name table using the F$LOGICAL lexical function as follows:

```
$ DEFINE/GROUP TERMINAL 'F$LOGICAL("SYS$COMMAND")'(RET)
```

❻ The EXIT command terminates the batch job.

❼ At the label NO_WAKE, the procedure uses the REPLY command to issue an error message and terminates the job with the EXIT command.

❽ At the label TELL, the CREATE command reads input from the input stream and creates a file on the device SYS$COMMAND. If BWAKE.COM is executed interactively, the data following the CREATE command is displayed on the terminal.

❾ The EXIT command closes the command procedure file; the dollar sign in the record indicates the end of the input data for the CREATE command.

Sample Execution

```
$ SUBMIT BWAKE/PARAMETER=11:30(RET)
 Job 435 entered on queue SYS$BATCH
    .
    .
    .
GALAXY:: Batch, HIGGINS 11:30:11.74
"wake up"
$
```

The SUBMIT command submits the file to the batch queue. After the job is submitted, the terminal remains free to continue interactive work. After the specified time has elapsed, the job issues the REPLY command to the terminal. The REPLY command itself creates the header message giving the time of day.

A.4 ENDED.COM

```
❶ $ PROCESS_NAME := 'F$PROCESS()'          ! Get process name
  $ !
❷ $ IF P1 .NES. "" THEN STATUS := 'F$MESSAGE(P1)'
❸ $ REPLY/BELL/TERMINAL='F$LOGICAL("HIGGINS_TT")' -
          "Job ''F$EXTRACT(0,8,PROCESS_NAME)' done ''STATUS'"
  $ EXIT
```

Notes

❶ The procedure obtains the name of the current process. The F$PROCESS lexical function returns the process name string. For a batch job, the default name is in the format _JOBnnn, where nnn is the job number of the job.

❷ If a parameter was specified, the procedure assumes that it is a status value. It uses the F$MESSAGE lexical function to obtain the message associated with the value from the system message file.

❸ The REPLY command specifies a terminal whose logical name is TERMINAL. This logical name should be in the group logical name table for the group of the submitter of the batch job. The REPLY command text uses the F$EXTRACT lexical function to extract the job number portion of the process name string. The text also contains the value of the symbol STATUS. Note that if no parameter is passed to the procedure, STATUS will be null and the resulting text of the message will consist only of the first part of the message. This job number is displayed in the message.

Sample Execution

The procedure TEST.COM in the default directory contains the lines:

```
$ ON ERROR THEN GOTO END
$ FORTRAN BIGFILE
$ LINK BIGFILE,TESTLIB/LIBRARY/MAP/FULL
$ @ENDED
$ EXIT
$ END:
$ @ENDED '$STATUS'
$ EXIT
```

This procedure can be executed as follows:

```
$ SUBMIT TEST
 Job 436 entered on queue SYS$BATCH
   .
   .
   .
GALAXY::Batch, HIGGINS 13:46:50.32
"Job _JOB436 done"
```

After the job is submitted, the terminal remains free for interactive work. When the job is completed, the REPLY command message is displayed on the terminal. This sample message indicates that the procedure completed successfully.

A.5 GETPARMS.COM

```
❶ $ SAVE_VERIFY = 'F$VERIFY("NO")'
  $ !
❷ $ IF P1 .EQS. "?" THEN GOTO TELL
  $ !
  $ ! Loop to count the number of parameters passed.  Null parameters
   are
  $ ! counted until the last non-null parameter is passed.
  $ !
  $       COUNT = 0
  $       LASTNONNULL = 0
❸ $ LOOP:
  $       IF COUNT .EQ. 8 THEN GOTO END_COUNT
  $       COUNT = COUNT + 1
  $       IF P'COUNT' .NES. "" THEN LASTNONNULL = 'COUNT'
  $ GOTO LOOP
  $ !
❹ $ END_COUNT:
  $ !
  $ ! Place the number of non-null parameters passed into PARMCOUNT.
  $ !
  $ PARMCOUNT== LASTNONNULL
  $ !
  $ ! Restore verification setting, if it was on, before exiting
  $ !
❺ $ IF SAVE_VERIFY THEN SET VERIFY
  $ EXIT
  $ !
❻ $ TELL:
  $ TYPE SYS$INPUT
          Procedure used to count the number  of  parameters  passed  to
          another  procedure.   This procedure can be called by entering
          the string:

                'GETPARMS

          in any procedure.  On  return,  the  global  symbol  PARMCOUNT
          contains the number of parameters passed to the procedure.
  $ !
  $ EXIT
```

Notes

❶ The procedure saves the current verification setting in the symbol named SAVE_VERIFY before setting verification off.

❷ If a question mark character was passed to the procedure as a parameter, the procedure branches to the label TELL (Note 6).

❸ A loop is established to count the number of parameters that were passed to the procedure. The counters COUNT and LASTNONNULL are initialized to 0 before entering the loop. Within the loop, COUNT is incremented and tested against the value 8. If COUNT is equal to 8, the maximum number of parameters has been entered. Each time a non-null parameter is passed, LASTNONNULL is equated to that parameter's number.

Each time the IF command executes, the symbol COUNT has a different value. The first time, the value of COUNT is 1 and the IF command checks P1. The second time, it checks P2, and so on.

❹ When the parameter count reaches 8, the procedure branches to END_COUNT. The symbol LASTNONNULL contains the count of the last non-null parameter passed. This value is placed in the global symbol PARMCOUNT. PARMCOUNT must be defined as a global symbol so that its value can be tested at the calling command level.

❺ The value of SAVE_VERIFY is tested; if it is true, verification will be restored.

❻ At the label TELL, the TYPE command parameter is the input stream; the data in the command file provides information about the use of the procedure GETPARMS.COM.

Sample Execution

The procedure SORTFILES.COM requires the user to pass three non-null parameters. The SORTFILES.COM procedure can contain the lines:

```
$ GETPARMS:== "@GETPARMS 'P1' 'P2' 'P3' 'P4' 'P5' 'P6' 'P7' 'P8'"
$ 'GETPARMS
$ IF PARMCOUNT .NE. 3 THEN GOTO NOT_ENOUGH
    .
    .
    .
$NOT_ENOUGH:
$ WRITE SYS$OUTPUT -
"Three non-null parameters required.  Type SORTFILES HELP for info."
$ EXIT
```

The procedure SORTFILES.COM can be invoked as follows:

```
$ @SORTFILES DEF 4(RET)
Three non-null parameters required.  Type SORTFILE HELP for info."
```

In the above example, the procedure SORTFILES.COM defines the symbol GETPARMS as a synonym for @GETPARMS and its parameters. For this procedure to be properly invoked, that is, for the parameters that are passed to SORTFILES to be passed to GETPARMS intact for processing, the synonym must be preceded with an apostrophe.

If the return value from GETPARMS is not 3, SORTFILES issues an error message and exits.

A.6 EDITALL.COM

```
❶ $ ON CONTROL_Y THEN GOTO DONE          ! CTRL/Y action
  $ ON ERROR THEN GOTO DONE
  $ !
  $ ! Check for file type parameter.  If one was entered, continue;
  $ ! otherwise, prompt for a parameter.
  $ !
❷ $ IF P1 .NES. "" THEN GOTO OKAY
  $ INQUIRE P1 "Enter file type of files to edit"
  $ !
  $ ! List all files with the specified file type and write the DIRECTORY
  $ ! output to a file named DIRECT.OUT
  $ !
  $ OKAY:
❸ $ DIRECTORY/VERSIONS=1/COLUMNS=1 -
          /NODATE/NOSIZE -
          /NOHEADING/NOTRAILING -
          /OUTPUT=DIRECT.OUT *.'P1'
❹ $ IF .NOT. $STATUS THEN GOTO DIRECTORY_ERROR
  $ !
❺ $ OPEN/READ DIRFILE DIRECT.OUT
  $ !
  $ ! Loop to read directory file
  $ !
❻ $ NEWLINE:
  $       READ/END=DONE DIRFILE NAME
  $       ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT:  ! Redefine SYS$INPUT
  $       EDIT 'NAME'       ! Edit the file
  $       GOTO NEWLINE
  $ !
❼ $ DONE:
  $       CLOSE DIRFILE/ERROR=NOTOPEN   ! Close the file
  $ NOTOPEN:
  $       DELETE DIRECT.OUT;*    ! Delete temp file
  $ EXIT
  $ !
  $ DIRECTORY_ERROR:
  $       WRITE SYS$OUTPUT "Error:  ''F$MESSAGE()'"
  $       DELETE DIRECT.OUT;*
  $ EXIT
```

Notes

❶ ON commands establish condition handling for this procedure. If any error occurs or if CTRL/Y is pressed at any time during the execution of this procedure, the procedure will branch to the label DONE. Similarly, if any error or severe error occurs, the procedure will branch to the label DONE (Note 7).

❷ The procedure checks whether a parameter was entered. If not, it will prompt for a file type.

❸ The DIRECTORY command lists all files with the file type specified as P1. The command output is written to the file DIRECT.OUT. The /VERSIONS=1 qualifier requests that only the highest numbered version of each file be listed. The /NOHEADING and /NOTRAILING qualifiers request that no heading lines or directory summaries be included in the output. The /COLUMNS=1 qualifier ensures that one file name per record is given.

❹ The IF command checks the return value from the DIRECTORY command by testing the value of $STATUS. If $STATUS has an even numeric value, the procedure exits.

❺ The OPEN command opens the directory output file and gives it a logical name of DIRFILE.

❻ The READ command reads a line from the DIRECTORY command output into the symbol name NAME. After it reads each line, it redefines the input stream for the edit session with the ASSIGN command. Then, it invokes the editor specifying the symbol NAME as the file specification. When the edit session is completed, the command interpreter reads the next line in the file.

❼ The label DONE is the target label for the /END qualifier on the READ command and the target label for the ON CONTROL_Y and ON ERROR conditions set at the beginning of the procedure. At this label, the procedure performs the necessary cleanup operations.

The CLOSE command closes the DIRECTORY command output file; the /ERROR qualifier specifies the label on the next line in the file. This use of /ERROR will suppress any error message that would be displayed if the directory file is not open. For example, this would occur if CTRL/Y were pressed before the directory file were opened.

The second step in cleanup is to delete the temporary directory file.

Sample Execution

```
$ @EDITALL DAT RET
Edit: DBA1:[MALCOLM.DATAFILES]ALPHA.DAT;4
*  .
   .
   .
*E RET
Edit: DBA1:[MALCOLM.DATAFILES]BETA.DAT;14
*  .
   .
   .
```

The procedure EDITALL is invoked with P1 specified as DAT. The procedure creates a directory listing of all files in the default directory whose file types are DAT and invokes the editor to edit each one.

A.7 FORTUSER.COM

```
❶ $ SET NOCONTROL_Y
  $ SET NOVERIFY
  $ !
❷ $ OPTION_TABLE := 4EDIT7COMPILE4LINK3RUN7EXECUTE5DEBUG5PRINT4HELP4FILE4DONE
  $ TYPE SYS$INPUT

❸       VAX/VMS Fortran Command Interpreter

        Enter file name with which you would like to work.
  $ !
  $ ! Set up for initial prompt
  $ !
❹ $ PROMPT := INIT0
  $ GOTO HELP0                  ! Print the initial help message
  $ !
  $ ! after the first prompting message, use the prompt: Next
  $ !
  $ INIT0:
  $ PROMPT := NEXT
  $ GOTO FILE0                  ! Get initial file name
  $ !
  $ ! Main command parsing routine.  The routine compares the current
  $ ! command against the options in the option table.  When it finds
  $ ! a match, it branches to the appropriate label.
  $ !
❺ $ NEXT:
  $       ON CONTROL_Y THEN GOTO NEXT      ! CTRL/Y resets prompt
  $       SET CONTROL_Y
  $       ON WARNING THEN GOTO NEXT        ! If any, reset prompt
  $ !
  $       INQUIRE COMMAND "Next"
  $       IF COMMAND .EQS. "" THEN GOTO NEXT
❻ $       COMMAND_SIZE = 'F$LENGTH(COMMAND)'       ! input length
  $       INDEX = 0                                ! initial index
  $ !
❼ $ CHECK_NEXT:
  $       OPTION_LENGTH = 'F$EXTRACT(INDEX,1,OPTION_TABLE)'
  $       IF OPTION_LENGTH .EQ. 0 THEN GOTO INVALID_COMMAND
  $       INDEX = INDEX + 1                        ! advance index
  $       NEXT_COMMAND := 'F$EXTRACT(INDEX,OPTION_LENGTH,OPTION_TABLE)'
  $       IF "''F$EXTRACT(0,COMMAND_SIZE,NEXT_COMMAND)'" -
                  .EQS. COMMAND -
                  THEN GOTO 'NEXT_COMMAND'0
  $       INDEX = INDEX + OPTION_LENGTH            ! set to next command
  $       GOTO CHECK_NEXT
  $ !
  $ !
❽ $ INVALID_COMMAND:
  $       WRITE SYS$OUTPUT " Invalid command"
  $ !
❾ $ HELP0:
  $       TYPE SYS$INPUT
          The commands you can enter are:
          FILE      enter file name of FORTRAN program
          EDIT      edit the program with SOS
          COMPILE   compile the program with VAX-11 FORTRAN
          LINK      link the program to produce an executable image
          RUN       run the program's executable image
          EXECUTE   same function as COMPILE, LINK, and RUN
          DEBUG     run the program under control of the debugger
          PRINT     queue the most recent listing file for printing
          DONE      return to interactive command level
          HELP      print this help message

          Enter CTRL/Y to restart this session
   $ !
⑩ $ GOTO 'PROMPT'
   $ !
⑪ $ EDIT0:
   $       ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT:
   $       SET NOCONTROL_Y          ! Must disable CTRL/Y for SOS
   $       EDIT 'FILE'.FOR
   $       GOTO NEXT
   $ !
   $ COMPILE0:
   $       FORTRAN 'FILE'/LIST/OBJECT/DEBUG
   $       GOTO NEXT
   $ !
   $ LINK0:
   $       LINK 'FILE'/DEBUG
   $       PURGE 'FILE'.*/KEEP=2
   $       GOTO NEXT
   $ !
   $ RUN0:
   $       ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT:
   $       RUN/NODEBUG 'FILE'
   $       GOTO NEXT
   $ !
   $ DEBUG0:
   $       ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT:
   $       RUN 'FILE'
   $       GOTO NEXT
   $ !
   $ EXECUTE0:
   $       FORTRAN 'FILE'/LIST/OBJECT
   $       LINK/DEBUG 'FILE'
   $       PURGE 'FILE'.*/KEEP=2
   $       RUN/NODEBUG 'FILE'
   $       GOTO NEXT
   $ !
   $ PRINT0:
   $       PRINT 'FILE'
   $       GOTO NEXT
   $ !
⑫ $ BADFILE:
   $       WRITE SYS$OUTPUT "Illegal file name, enter name portion only!"
   $       WRITE SYS$OUTPUT ""
   $       WRITE SYS$OUTPUT " For example:  ALPHA"
   $       WRITE SYS$OUTPUT ""
   $       GOTO NEXT
   $ !
⑬ $ FILE0:
   $       INQUIRE FILE "File"
   $       IF FILE .EQS. "" THEN GOTO FILE0
   $ !
   $       IF 'F$LOCATE(".",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
   $ !
   $       IF 'F$LOCATE("[",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
   $ !
   $       IF 'F$LOCATE("]",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
   $ !
   $       IF 'F$LOCATE("<",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
   $ !
   $       IF 'F$LOCATE(">",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
   $ !
   $       IF 'F$LOCATE(";",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
$ !
$       IF 'F$LOCATE("$",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
$ !
$       IF 'F$LOCATE("_",FILE)' .NE. 'F$LENGTH(FILE)' THEN GOTO BADFILE
$ !
$       GOTO NEXT
$ !
$ DONE0:
$ EXIT
```

Notes

❶ The SET NOCONTROL_Y command ensures that the user who logs in under the control of this procedure cannot interrupt the procedure or any command or program in it.

❷ The option table lists the commands that the user will be allowed to execute. In the list, each command is preceded by a decimal number indicating the length of the command name.

❸ The procedure introduces itself.

❹ The symbol name PROMPT is given the value of a label in the procedure. When the procedure is initially invoked, this symbol has the value INIT0. The HELP command text terminates with a GOTO command that specifies the label PROMPT (Note 10): when this text is displayed for the first time, the GOTO command results in a branch to the label that (1) changes the value of the symbol PROMPT and (2) branches to the prompt for a file name. Thereafter, when the text is displayed, the GOTO command results in a branch to the label NEXT, where the prompt is the string "Next".

❺ The CTRL/Y condition action is set to return to this prompt, as is the warning condition action. The procedure prompts for a command and continues to prompt, even if nothing is entered. To terminate the session, the command DONE must be used.

❻ The procedure uses the F$LENGTH lexical function to determine the length of the command that was entered. It sets an index counter, the symbol INDEX, to 0. This counter will be used to step through the option table.

❼ The label CHECK_NEXT introduces a loop in which the procedure finds a match in the option table for the command that was entered. It takes the following steps:

a. It compares the length of the command that was entered with the length of the first (next) option in the table. A length of 0 indicates the end of the table; in this case, the procedure branches to the error label INVALID_COMMAND.

b. If OPTION_LENGTH is nonzero, the index will be increased by 1 to point to the start of the option name. Using the value of INDEX as an offset and the value of OPTION_LENGTH as the length, the procedure extracts the name of the option from the option table.

c. The string value of NEXT_COMMAND is compared with the command that the user entered. If they match, the procedure will branch to the label corresponding to the option name.

d. If the commands do not match, the value of INDEX will be increased by the length of the option and will point to the next length field. Then, the procedure will branch to the start of the loop and the next option is checked.

8 At the label INVALID_COMMAND, the procedure writes an error message and displays the help text that lists the commands that are valid.

9 The help text lists the commands that are valid. This text is displayed initially. It is also displayed whenever the user issues the HELP command or any invalid command.

10 At the conclusion of the HELP text, the GOTO command specifies the symbol name PROMPT. When this procedure is first invoked, the symbol has the value INIT0. Thereafter, it has the value NEXT.

11 Each valid command in the list has a corresponding entry in the option table and a corresponding label in the command procedure. For the commands that read input from the terminal, for example, EDIT, the procedure contains an ASSIGN command that defines the input stream as SYS$COMMAND.

12 At the label BADFILE, the procedure displays information about how to enter file names. Only FORTRAN programs can be edited, so the procedure itself defaults all file types to FOR.

13 At the label FILE0, the initial prompt for a file name, the procedure checks the syntax of the file name that was entered.

Sample Execution

```
Username: CLASS30
Password:

                 VAX/VMS Version 2.0


     VAX/VMS Fortran Command Interpreter

     Enter file name with which you would like to work.


     The commands you can enter are:

      FILE       enter file name of FORTRAN program
      EDIT       edit the program with SOS
      COMPILE    compile the program with VAX-11 FORTRAN
      LINK       link the program to produce an executable image
      RUN        run the program's executable image
      EXECUTE    same function as COMPILE, LINK and RUN
      DEBUG      run the program under control of the debugger
      PRINT      queue the most recent listing file for printing
      DONE       return to interactive command level
      HELP       print this help message
```

```
          Enter CTRL/Y to restart this session
     File:  AVERAGE(RET)
     Next:  COMPILE(RET)
     Next:  LINK(RET)
     Next:  RUN(RET)
     Next:  FILE(RET)
     File:  READFILE(RET)
     Next:  EDIT(RET)
```

This sample execution illustrates logging in, the message of the help text being displayed, and some sample commands. First, the user specifies the file AVERAGE, compiles, links, and runs it. Then the user issues the FILE command to begin working on another file.

A.8 LISTER.COM

```
  $ ! Procedure to accumulate programmer names and document
  $ ! files.  After all names and files are entered, they are
  $ ! sorted in alphabetic order by programmer name.
  $ !
❶ $ SAVE_MODE = 'F$VERIFY("NO")'
  $ !
❷ $ OPEN/WRITE OUTFILE DATA.TMP           ! Create output file
  $ !
  $ LOOP:
❸ $         INQUIRE NAME "Programmer"
  $         IF NAME .EQS. "" THEN GOTO FINISHED
  $         INQUIRE FILE "Document file name"
  $         RECORD[0,20]:='NAME'
  $         RECORD[21,20]:='FILE'
❹ $         WRITE OUTFILE RECORD
  $         GOTO LOOP
  $ !
  $ FINISHED:
  $         CLOSE OUTFILE
  $ !
❺ $ ASSIGN/USER_MODE STATISTIC.SRT SYS$OUTPUT:   ! Suppress sort output
  $ SORT/KEY=(POSITION:1,SIZE=20) DATA.TMP DOC.SRT
  $ !
❻ $ OPEN/WRITE OUTFILE DOCUMENT.DAT
  $ WRITE OUTFILE "Programmer Files as of ''F$TIME()'"
  $ WRITE OUTFILE ""
  $ RECORD[0,20]:="Programmer Name"
  $ RECORD[0,20]:="File Name"
  $ WRITE OUTFILE RECORD
  $ WRITE OUTFILE ""
  $ !
❼ $ CLOSE OUTFILE
  $ APPEND DOC.SRT DOCUMENT.DAT
  $ PRINT DOCUMENT.DAT
  $ !
❽ $ INQUIRE CLEAN_UP "Delete temporary files?"
  $ IF CLEAN_UP THEN DELETE DATA.TMP;*,DOC.SRT;*,STATISTIC.SRT;*
  $ IF SAVE_MODE THEN SET VERIFY
  $EXIT
```

Notes

❶ LISTER.COM saves the current verification setting and sets verification off.

❷ The OPEN command opens a temporary file for writing.

❸ INQUIRE commands prompt for a programmer name and for a file name. If a null line, signaled by RET , is entered in response to the INQUIRE command prompt, the procedure will assume that no more data is to be entered and will branch to the label FINISHED.

❹ After assigning values to the symbols NAME and FILE, the procedure uses the character string overlay format of an assignment statement to construct a value for the symbol RECORD. In columns 1 through 21 of RECORD, the current value of NAME is written. The command interpreter pads the value of NAME with spaces to fill the 20-character length specified.

Similarly, the next 20 columns of RECORD are filled with the value of FILE. Then, the value of RECORD is written to the output file.

❺ After the file has been closed, the procedure sorts the output file DATA.TMP. The ASSIGN command directs the SORT command output to the file STATISTIC.SRT. Otherwise, these statistics would be displayed on the terminal.

The sort is performed on the first 20 columns, that is, by programmer name.

The sorted output file has the name DOC.SRT.

❻ The procedure creates the final output file, DOCUMENT.DAT, with the OPEN command. The first lines written to the file are header lines, giving a title, the date and time of day, and headings for the columns.

❼ The procedure closes the file DOCUMENT.DAT and appends the sorted output file, DOC.SRT, to it. Then, the output file is queued to the system printer.

❽ Last, the procedure prompts to determine whether to delete the intermediate files. If a true response is entered to the INQUIRE command prompt, the files DATA.TMP and DOC.SRT will be deleted. Otherwise, they will be retained.

Sample Execution

```
$ @LISTER(RET)
Programmer: WATERS(RET)
Document file name: CRYSTAL.CAV(RET)
Programmer: JENKINS(RET)
Document file name: MARIGOLD.DAT(RET)
Programmer: MASON(RET)
Document file name: SYSTEM.SRC(RET)
Programmer: ANDERSON(RET)
Document file name: JUNK.J(RET)
Programmer:(RET)
Delete temporary files:y(RET)
```

The output file resulting from this execution of the procedure is:

```
Programmer Files as of 10-APR-1979 16:18:58.79

Programmer Name      File Name

ANDERSON             JUNK.J
JENKINS              MARIGOLD.DAT
MASON                SYSTEM.SRC
WATERS               CRYSTAL.CAV
```

A.9 CALC.COM

```
   $ SAVE_VERIFY = 'F$VERIFY("NO")'
   $ ! Desk calculator program that resolves expressions and
   $ ! displays the resulting value in both decimal and hexadecimal.
   $ !
   $ START:
   $ !
❶ $ ON WARNING THEN GOTO START
❷ $ INQUIRE EXPRESSION "Calc"
   $ IF EXPRESSION .EQS. "" THEN EXIT
   $ !
❸ $ EXP_LEN = 'F$LENGTH(EXPRESSION)'
❹ $ EXP_POS = 'F$LOCATE("=",EXPRESSION)'
   $ IF EXP_POS .EQ. EXP_LEN THEN GOTO NO_EXP
   $               EXP_LEN = EXP_LEN - EXP_POS
   $               EXP_POS = EXP_POS + 1
❺ $               DECIMAL = 'F$EXTRACT(EXP_POS,EXP_LEN,EXPRESSION)'
   $               GOTO HEX_CONV
   $ !
   $ NO_EXP:
   $ SHOW SYMBOL EXPRESSION
   $               DECIMAL = 'EXPRESSION'
   $ !
❻ $ HEX_CONV:
   $               NUM[0,32]='DECIMAL'
   $               DIGIT = 0
   $               LOOP:
❼ $                       X = 'F$CVUI(DIGIT,4,NUM)'
   $                       IF X .LE. 9 THEN GOTO A
   $                               F := ""
   $                               F[0,8]=65+(X-10)
   $                               X := 'F
❽ $                       A:
   $                       DIGIT = DIGIT + 4
   $                       HEX := 'X''HEX'
❾ $               IF DIGIT .LT. 32 THEN GOTO LOOP
   $ !
❿ $ WRITE SYS$OUTPUT "Decimal = ''DECIMAL' Hex = ''HEX'"
⓫ $ DELETE/SYMBOL HEX
   $ DELETE/SYMBOL DECIMAL
⓬ $ GOTO START
   $ EXIT:
   $ IF SAVE_VERIFY THEN SET VERIFY
   $ EXIT
```

Notes

❶ The procedure establishes an error handling condition that restarts the procedure. If a warning or error of greater severity occurs, the procedure will branch to the beginning where it resets the ON condition.

This technique ensures that the procedure will not exit if the user enters an expression incorrectly.

❷ The INQUIRE command prompts for an arithmetic expression. The procedure accepts expressions in either of the formats:

```
name = expression
expression
```

If no expression is entered, the procedure will assume the end of a CALC session and exit.

❸ The F$LENGTH lexical function determines the length of the expression entered.

❹ The F$LOCATE lexical function locates the position of the equal sign (=) in the expression entered. If the value returned from F$LOCATE is equal to the length of the expression, no equal sign was entered and the response was in the second format shown in Note 2, above.

If the expression was in the first format, the procedure will use arithmetic assignment statements to locate, specifically, the expression on the right-hand side of the equal sign and the F$EXTRACT function to extract the expression from the data that was entered.

❺ In either case, the symbol name DECIMAL is given the value of the calculated expression. If an equal sign is present, the assignment statement that extracts the value from the entered expressions gives the value to DECIMAL. If no assignment statement is present, the assignment statement at the label NO_EXP will give this value to DECIMAL. Note that the apostrophe preceding the symbol EXPRESSION is required to force recursive substitution. Otherwise, the procedure would replace EXPRESSION with its value, but would not perform the calculations.

❻ To convert the decimal value to hexadecimal, the procedure first converts the decimal value to a binary number using the arithmetic overlay syntax of an assignment statement. The symbol NUM now has a binary value equivalent to the value of DECIMAL.

❼ Using the symbol DIGIT as a counter, the procedure loops, extracting four bits at a time from the value of NUM. Each four bits are converted to their hexadecimal equivalent to construct a hexadecimal value for DECIMAL.

The procedure gives the symbol X the value of the current four bits.

❽ If the value of the 4-bit field is less than 9, the procedure prefixes the current value of the symbol HEX with the value of X. Otherwise, the procedure calculates a hexadecimal value using an intermediate symbol, F. The resulting value of F is given to the symbol X and X is prefixed to the current value of HEX. Note that regardless of whether the current four bits are greater than or less then 9, X is given an ASCII value representing the hexadecimal value of the bits.

❾ When the value of DIGIT exceeds 32, the conversion is complete.

❿ The procedure displays the results with the WRITE command.

⓫ The DELETE/SYMBOL commands delete the symbols used in the calculations.

⓬ The GOTO command returns to the label START. The procedure loops to request another value.

Sample Execution

```
$ @CALC(RET)
Calc: 5555*30(RET)
Decimal = 166650 Hex = 00028AFA
Calc: 32+3(RET)
Decimal = 35 Hex = 00000023
Calc: TOTAL = %X3A + %X4C(RET)
Decimal = 134 Hex = 00000086
Calc:(RET)
$
```

After each prompt from the procedure, the user enters an arithmetic expression. The procedure displays the results in decimal and hexadecimal. A null line, signaled by (RET) on a line with no data, concludes the CALC session.

INDEX

Please cut along this line.

VAX/VMS Guide to Using
Command Procedures
AA-H782A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)____________

Name______________ Date______________

Organization______________

Street______________

City______________ State______________ Zip Code______________
or

— — — Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

- — Do Not Tear - Fold Here —