VAX/VMS
Guide to Using
Command Procedures

Order No. AA-H782B-TE

May 1982

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

The postpaid READER'S COMMENTS form on the last page of this document
requests the user's critical evaluation to assist in preparing future
documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	RSX
DEC/CMS	EduSystem	UNIBUS
DECnet	IAS	VAX
DECsystem-10	MASSBUS	VMS
DECSYSTEM-20	PDP	VT
DECUS	PDT	digital
DECwriter	RSTS	

ZK2127

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

CONTENTS

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 write 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.

PREFACE

- 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 and indexed sequential (ISAM) 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 (DCL), 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, 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 source while studying the material in this manual.

CONVENTIONS USED IN THIS DOCUMENT

This manual uses the following graphic conventions.

RET   This symbol indicates that you press the RETURN 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.

x

```
$  SHOW TIME
   05-JUN-1982  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 (').

SUMMARY OF TECHNICAL CHANGES

This manual introduces new rules for assigning symbols (Chapter 3) and includes descriptions of 13 new lexical functions (Chapter 5). The new lexical functions are:

F$CVTIME	F$GETSYI	F$SEARCH
F$FAO	F$INTEGER	F$SETPRV
F$FILE_ATTRIBUTES	F$PARSE	F$STRING
F$GETDVI	F$PID	
F$GETJPI	F$PRIVILEGE	

Numerous other small changes have been made to this manual to incorporate corrections, additions, clarifications, and minor formatting and syntax improvements.

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 become more skilled in creating and using 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 written 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 files using command procedures

- Control batch jobs and batch job queues from command procedures

Some of 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.
=	Symbol assignment; equates a local symbol name to an integer expression or character string expression.
==	Symbol assignment; equates a global symbol name to an integer expression or character string expression.
:=	String assignment; equates a local symbol name to a character string. Commonly used to equate a symbol to a command string without having to enclose the string in quotation marks.
:==	String assignment; equates a global symbol name to a character string. Commonly used to equate a symbol to a command string without having to enclose the string in quotation marks.
label:	Defines a label for a GOTO statement.
ASSIGN	Equates a logical name to a physical device name, to a complete file specification, or 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.

(continued on next page)

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

Name	Function
CONTINUE	Resumes execution of a command procedure or image that was interrupted by pressing CTRL/Y or CTRL/C.
DEASSIGN	Cancels a logical name assignment made with the ALLOCATE, ASSIGN, or DEFINE command.
DECK	Marks the beginning of an input stream for a command procedure when the first nonblank character in any data record in the input stream is a dollar sign ($).
DEFINE	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.
DELETE/ENTRY	Deletes one or more entries from a print or batch job queue.
DELETE/SYMBOL	Deletes a symbol definition from a local symbol table or from the 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 integer and string expressions and about 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.

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

Name	Function
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.
SET CARD_READER	Defines the default ASCII translation mode for a card reader.
SET CONTROL	Enables interrupts caused by CTRL/Y and CTRL/T.
SET NOCONTROL	Disables interrupts caused by CTRL/Y and CTRL/T.
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 print 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.

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

Name	Function
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 text editor such as EDT or SOS 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 EDT editor:

```
$   CREATE FRED.COM(RET)
$   RUN A(RET)
$   RUN B(RET)
$   RUN C(RET)
(CTRL/Z)
$


$   EDIT FRED.COM(RET)
Input file does not exist
[EOB]
*C(RET)
$   RUN A(RET)
$   RUN B(RET)
$   RUN C(RET)
(CTRL/Z)
*EXIT
DBA1:[USER]FRED.COM;1 3 lines
$
```

You can construct command procedures that contain only data to be read by a command or program; 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@DEFLINK

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.

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

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 follow the dollar sign with no blank spaces or tabs or with one or more blank spaces or tabs.

The format used for command strings within command procedures 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 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 Using 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 V3.0
$   !
$   !       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.

1.2.2.2 Spelling 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 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 Using Indentation – In longer command procedures, you can improve readability by tabbing command lines to offset them from

labels you use. The dollar sign ($) should always be in column 1.
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.

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.

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 work at
the terminal. Upon completion, the operating system deletes the batch
process.

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 login 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 located in your default directory, issue this command:

    $ @WEATHER

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 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 can then resume interactive work.

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

    $ @DBB2:[COMMON]SETUP.FIL

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

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.

```
Username: HIGGINS
Password:
.
.
.
$ @TESTALL  ─────►
$
```

┌─────────────────────┐
│ Command interpreter │
│ finds TESTALL.COM   │
│ on default device   │
│ and directory...   │
└─────────────────────┘

DBA1:[HIGGINS]TESTALL.COM

┌─────────────────┐
│ $ RUN A   │
│ $ RUN B   │
│ $ RUN C   │
└─────────────────┘

┌─────────────────────┐
│ then executes the   │
│ TESTALL.COM com-   │
│ mands sequentially..│
└─────────────────────┘

┌─────────────────────┐
│ and returns control │
│ to interactive com- │
│ mand level after   │
│ TESTALL.COM   │
│ completes   │
└─────────────────────┘
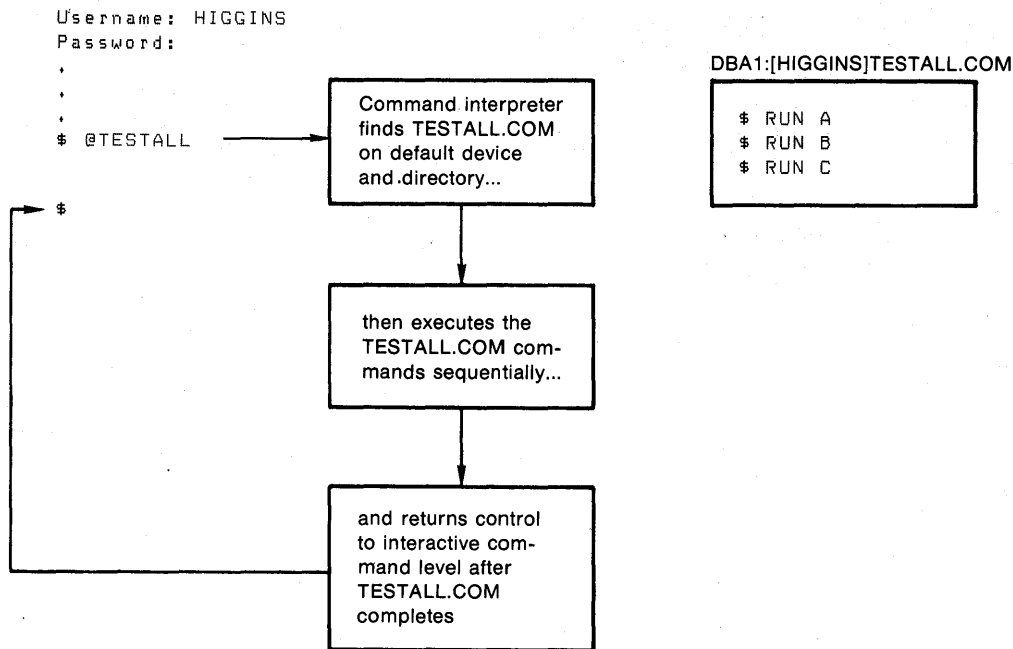
ZK-790-82

Figure 1-1: Executing a Command Procedure in Interactive Mode

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 a default file type of COM for
the command procedure. For example, to execute the command procedure
TESTALL.COM in your default 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...

DBA1:[HIGGINS]TESTALL.COM

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

then requests queue for the batch job

TESTALL.COM gets a job number and is placed in SYS$BATCH queue

SYS$BATCH QUEUE

```
    .
    .
    .
JOB NUMBER 208
JOB NUMBER 209
JOB NUMBER 210
    .
    .
    .
```

Command interpreter returns job information (and control) to interactive command level

When Job 210 can be executed, a process is created to execute the job. When the job is completed, the process is deleted

ZK-811-82

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 login 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.
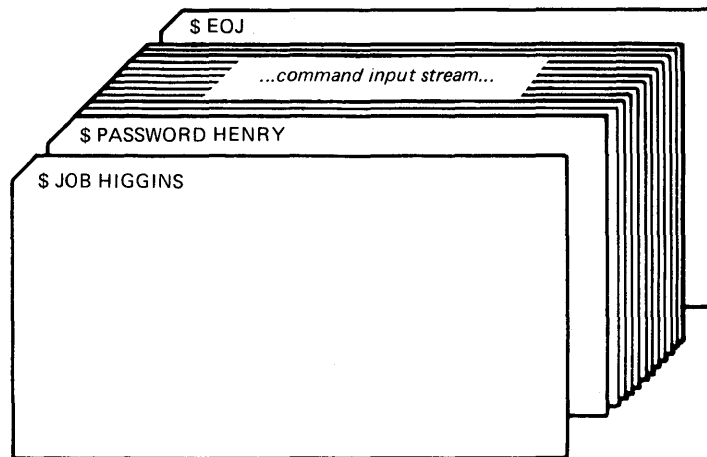
ZK-812-82

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, INPBATCH.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 INPBATCH.COM from the user's default disk and directory.
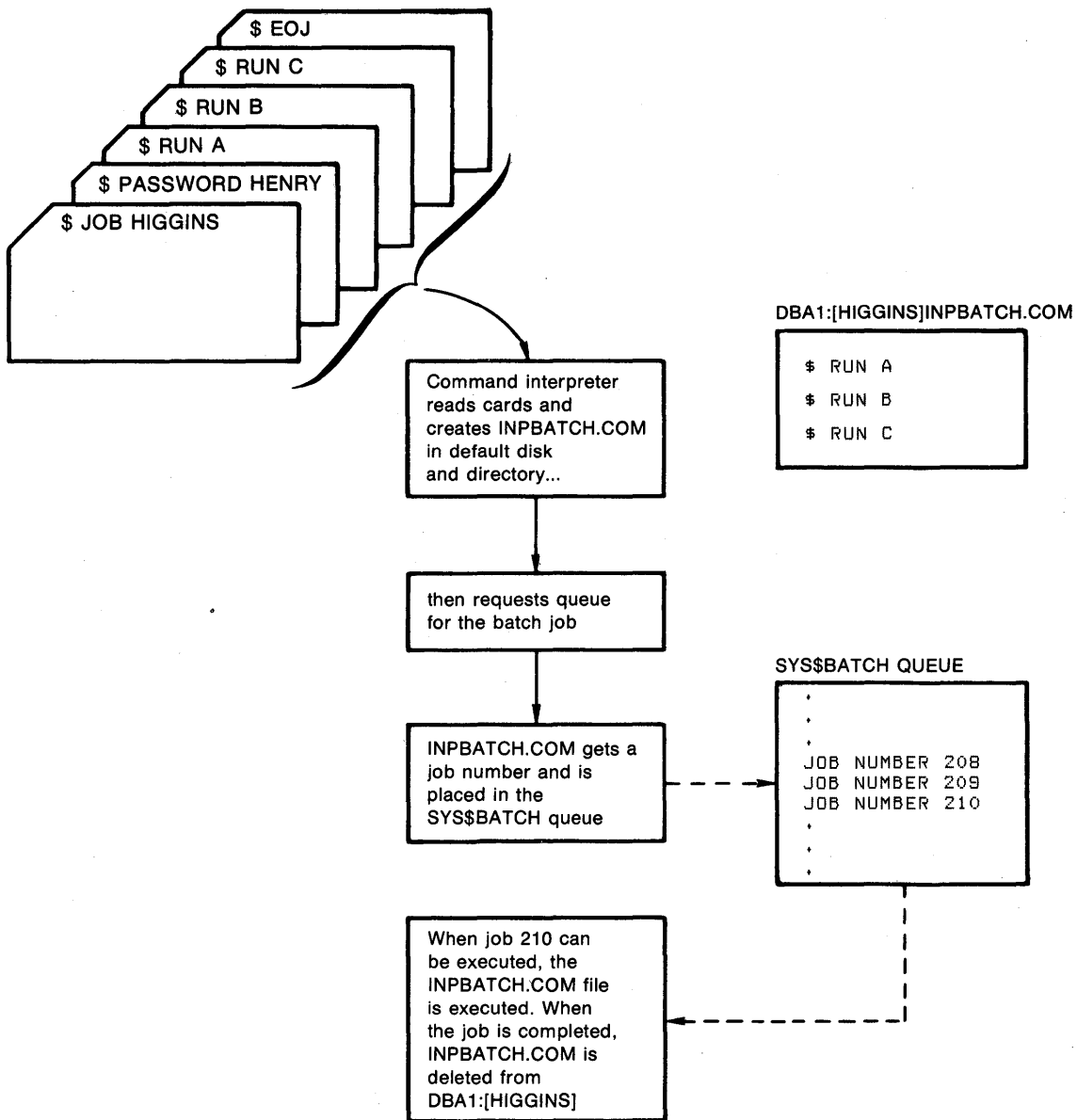
Figure 1-4:   Submitting a Batch Job Through a System Card Reader

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; it 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, 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 command or sequence 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 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
$  DEFINE PAY DBA1:[MALCOLM.PAYROLL]
$  DEFINE DOC DBA1:[MALCOLM.DOCUMENTS]
$  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 file, you should locate it on your default disk and directory, and name it LOGIN.COM. This is the default specification for the LGICMD in the user authorization file (UAF). The LGICMD defines the name of the user login file in the UAF.

The name of your login file does not have to be LOGIN.COM, although this convention predominates. The system manager, who authorizes use of the system, can assign any name in the UAF. However, the name the system manager assigns in your UAF must match your login file in order for that file to be executed at login.

Another approach to assigning login files is for the system manager to set up a login file to be executed on behalf of a selected number of users. When a system manager has defined such a file, the commands in that file are executed instead of any user-defined login files. However, the end of the system manager's file can, and usually does contain the command:

```
@LOGIN
```

which then executes the individual user-defined login files.

1.3.5.2 A System- or Group-Defined Login File - The system manager can optionally specify, for all system users or users having the same group UIC, a login file that is executed before the login file specified by the user's LGICMD. He does this by assigning the logical name SYS$SYSLOGIN to a system-defined login file. The commands in the

following example define SYS$SYLOGIN as either a system or group
logical name for SYS$MANAGER:SYSLOGIN.COM, the system-defined login
file.

    $ DEFINE/SYSTEM SYS$SYLOGIN SYS$MANAGER:SYSLOGIN.COM
or
    $ DEFINE/GROUP SYS$SYLOGIN SYS$MANAGER:SYSLOGIN.COM

The DEFINE/SYSTEM command places SYS$SYLOGIN into the system logical
name table, and the DEFINE/GROUP command places SYS$SYLOGIN into the
group logical name table. Thus, at login, the system-defined login
file is executed for all users on the system or all users in a
particular group, depending on how SYS$SYLOGIN was defined. Once the
execution of this login file completes, the individual user-defined
login files (LGICMD defined) are executed.

The difference between the user-defined login file and the
system-defined login file 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. After the system-defined
login file completes, the command interpreter locates and executes the
user-defined login file on the default disk and directory for user
HIGGINS.
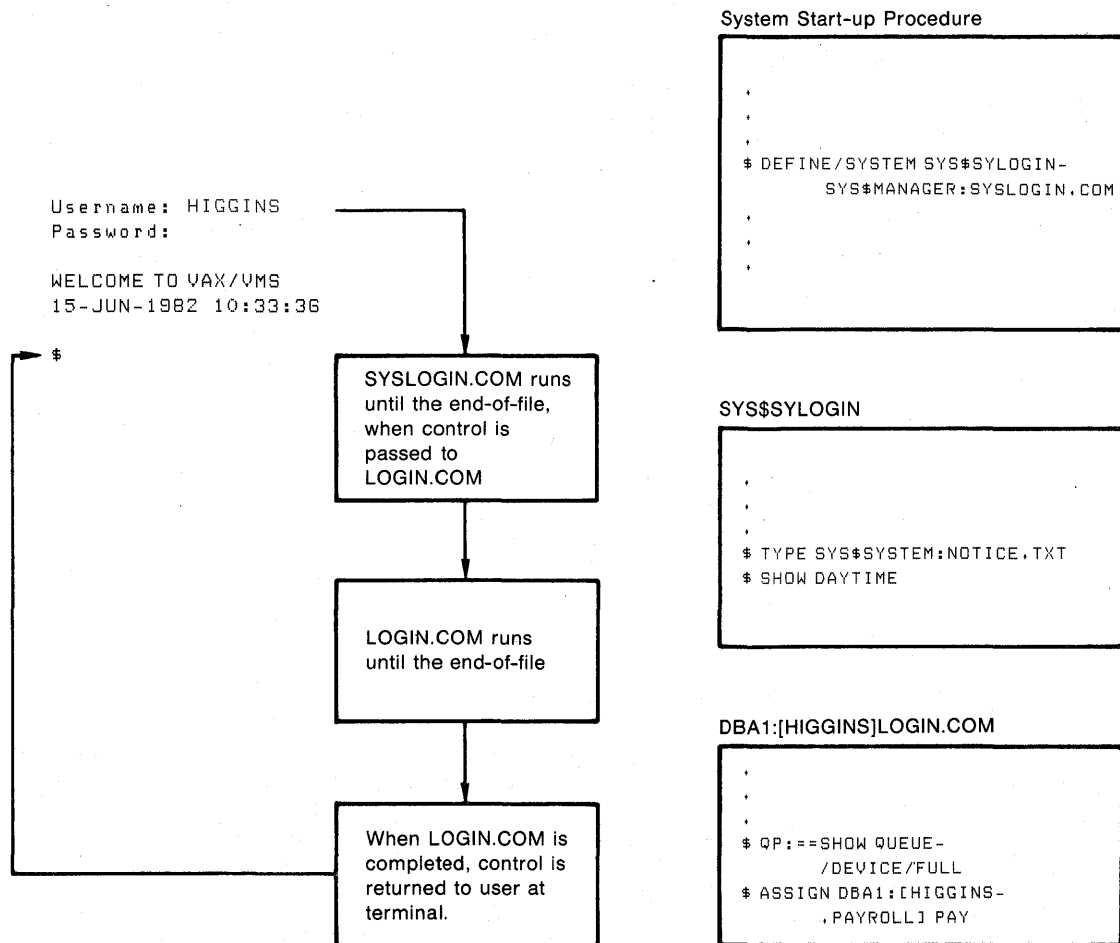
ZK-814-82

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 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. 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 2.2,
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 a new release may occasionally change 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 DEFINE 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.

- SYS$SCRATCH -- The default device and directory to which temporary files are written.

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$ERROR and SYS$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.

User name: HIGGINS
Password:

```
       input    = TTB3:
  ⓪    output   = TTB3:
       error    = TTB3:
       command  = TTB3:
```

$@PROC1   PROC1.COM

```
          input    = DBA1:PROC1.COM
  ❶       output   = TTB3:
          error    = TTB3:
          command  = TTB3:
```

$next command

$ @PROC2/OUTPUT=PROC2.OUT   PROC2.COM

```
          input    = DBA1:PROC2.COM
  ❷       output   = DBA1:PROC2.OUT
          error    = TTB3:
          command  = TTB3:
```

$ EXIT

$ @DBB2:PROC3   DBB2:PROC3.COM

```
          input    = DBB2:PROC3.COM
  ❸       output   = DBA1:PROC2.OUT
          error    = TTB3:
          command  = TTB3:
```

$ EXIT

$ EXIT

$SUBMIT BATCH1

$next-command

```
BATCH1.COM

       input    = DBA1:BATCH1.COM
  ⓪    output   = DBA1:BATCH1.LOG
       error    = DBA1:BATCH1.LOG
       command  = DBA1:BATCH1.COM
```

$ @BATCH2   BATCH2.COM

```
          input    = DBA1:BATCH2.COM
  ❶       output   = DBA1:BATCH1.LOG
          error    = DBA1:BATCH1.LOG
          command  = DBA1:BATCH1.COM
```

$ next command

$ @BATCH3/OUTPUT=BATCH3.OUT   BATCH3.COM

```
          input    = DBA1:BATCH3.COM
  ❷       output   = DBA1:BATCH3.OUT
          error    = DBA1:BATCH1.LOG
          command  = DBA1:BATCH1.COM
```

$ EXIT

$ EXIT

Key:

input	Input stream (SYS$INPUT)
output	Output stream (SYS$OUTPUT)
error	Error stream (SYS$ERROR)
command	Command stream (SYS$COMMAND)

———▶ Transfer of control

❶ Command Level

— —— — Execution occurs in a separate process

ZK-815-82

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

When you issue this command, all the data that is normally displayed on your terminal as TESTALL.COM is executed is written instead 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

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. You cannot redefine SYS$ERROR.

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	Terminal displays: All messages to SYS$ERROR
	Log File contains: All DCL command lines and comment lines	Log File contains: All messages to SYS$OUTPUT
	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 to
a 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.
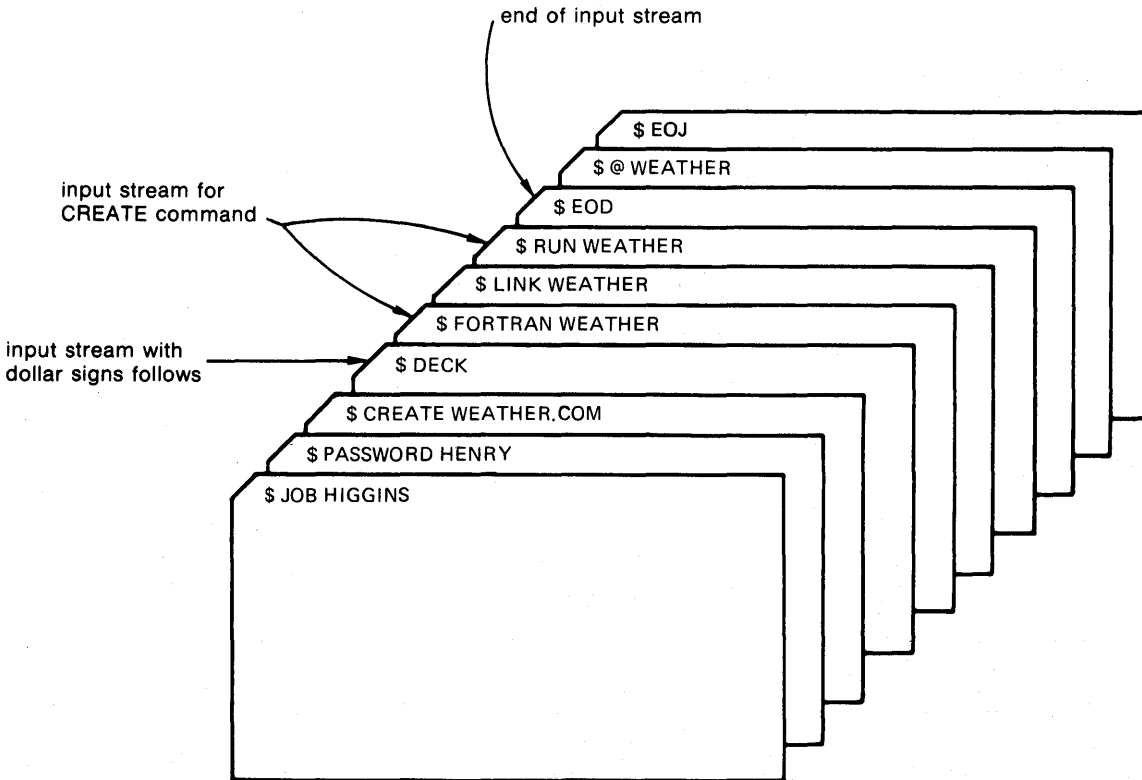
Figure 2-2: An Input Data Stream with Dollar Signs

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 a DEFINE command in the command procedure before the RUN command:

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

This DEFINE 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 a DEFINE 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:

```
$ DEFINE SYS$INPUT SYS$COMMAND:
$ 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 a DEFINE command in a command procedure to change the equivalence of a logical name for a process-permanent file (such as SYS$INPUT), you can use the DEASSIGN command to cancel the equivalence.

The following command procedure shows how the DEASSIGN command is used to change an equivalence name.

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

In this example, the DEFINE command changes SYS$INPUT so that the editor can be run from the command procedure. Later, the input data for the program AVERAGE follows the RUN command in the procedure. The DEFINE command, however, has redefined the input stream and this assignment (the terminal) is still in effect. The DEASSIGN command cancels the logical name assignment for SYS$INPUT at the end of the editing session and restores the default command input stream, so when the RUN command executes the AVERAGE program, AVERAGE reads its data from the command procedure instead of from the terminal.

A more convenient way to run the above example is to use the /USER_MODE qualifier in the DEFINE 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:

```
$ DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
$ 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.

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:

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

In the above example, statistics that the SORT command normally
displays are redirected to the file STATISTIC.SRT. The DEFINE 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. Table 2-2 lists the commands
that do not allow output to be redirected.

Table 2-2: Commands that Prohibit Redirection of Output

ALLOCATE	ASSIGN	ATTACH
DEASSIGN	DEFINE	SHOW DEFAULT
SHOW PROTECTION	SHOW QUOTA	SHOW RMS_DEFAULT
SHOW STATUS	SHOW SYMBOL	SHOW TIME
SHOW TRANSLATION	SPAWN	

The sample command procedure LISTER.COM in Appendix A illustrates
how to define SYS$OUTPUT to suppress program output.

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 3.11, 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.

```
$ 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.
$
```

ZK-817-82

Figure 2-3: Displaying Data in the Output Stream

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.

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 FILE the value ALPHA. Subsequently, the file ALPHA can be referred to symbolically by specifying 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 the symbol FILE before parsing the FORTRAN command.

This chapter describes the syntax of symbol names and gives examples of defining symbol values. Chapter 4 provides detailed information on how the command interpreter substitutes values for symbols during command processing; Chapter 5 introduces the command language's lexical functions.

3.1 SYMBOL NAMES

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

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 expressions, constant values, lexical functions, or to other variable symbol names with assignment statements (described in Sections 3.2 through 3.7)

- 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 3.10)

- Using the INQUIRE command to prompt for a symbol's value during the execution of a command procedure (described in Section 3.11)

- Using the READ command to read a character string from an input file or device and assigning the character string value that was a symbol name (described in Chapter 8)

You can delete symbol names from local and global symbol tables as described in Section 3.12.

3.1.1 Symbol Types and Expressions

You can equate a symbol to either a string expression or an integer expression. The actual value assigned to a symbol is the result of the expression. Expressions can contain character strings, integers, lexical functions, or symbols. For example:

    $ CODE = 4 + F$INTEGER("6") - A

This command assigns the symbol CODE to an expression containing an integer (4), a lexical function (F$INTEGER("6")), and a symbol (A). The actual value assigned to the symbol CODE is the result of the expression. In this example, the expression evaluates to an integer value.

The value type (string or integer) assigned to a symbol is determined by the mode (whether the expression evaluates to a string or integer value) of the expression. If the expression evaluates to a string, the symbol is assigned a string value. If the expression evaluates to an integer, the symbol is assigned an integer value.

The mode of an expression is determined by the types of values used in the expression and the operations used to manipulate them. Table 3-1 lists the rules for determining the mode of an expression.

3.1.2 Value Type Conversion in Expressions

In expressions that contain both integer and string values as operands, string values are automatically converted to integer values before any operation is performed. Numeric strings are converted to integer values; alphabetic strings are converted to the integer 1 if the string begins with T, t, Y, or y, or to the integer 0 if the string begins with any other letter. The following examples show how string values are converted to integer values:

String		Integer
"123"	$\longrightarrow$	123
"Test"	$\longrightarrow$	1 (True)
"file"	$\longrightarrow$	0 (False)

The only exception to the implicit conversion rule described above occurs in string comparison operations. In string comparison operations, any integer value in the expression is converted to a string value before the string comparison is performed. The following examples show how integer values are converted to string values in string comparison expressions:

Integer	$\longrightarrow$	String
123	$\longrightarrow$	"123"
1	$\longrightarrow$	"1"
0	$\longrightarrow$	"0"

You can also perform value conversion explicitly using the F$INTEGER and F$STRING lexical functions (see Chapter 5).

Table 3-1: Rules for Determining Expression Modes

Expression	Mode and Resultant Value Type
Integer value	Integer
String value	String
Integer lexical function	Integer
String lexical function	String
Integer symbol	Integer
String symbol	String
+, -, or .NOT. any value	Integer
Any value .AND. or .OR. any value	Integer
String + or - string	String
Integer + or - any value	Integer
Any value + or - integer	Integer
Any value * or / any value	Integer
Any value (string comparison) any value	Integer
Any value (arithmetic comparison) any value	Integer

In the above table, if "any value" is a character string value, it is converted to an integer value before the operation is performed, except for the string comparison expression. In |string comparison expressions, if "any value" is an integer, it is converted to a string value before the string comparison is performed.

3.1.3 Lexical Functions in Expressions

The command language contains constructs called lexical functions that return information about the process and system, and manipulate integer and string expressions.

You can use lexical functions in expressions in the same way you would normally use integer values, string values, or symbols. However, the value and type returned by a lexical function depends on the function. Lexical functions return either a string or integer value. For example:

```
$ A = F$STRING(81)
$ SHOW SYMBOL A
A = "81"
```

Since the value returned by the lexical function F$STRING in this example is the character string "81", the symbol A is assigned the string value "81". Thus, whenever A is used in an expression, its type is a string.

Table 3-2 lists the lexical functions and their respective value types.

Table 3-2: Summary of Lexical Function Value Types

Function	Value Type
F$CVSI	Integer
F$CVUI	Integer
F$CVTIME	String
F$DIRECTORY	String
F$EXTRACT	String
F$FAO	String
F$FILE_ATTRIBUTES	Integer or string
F$GETDVI	Integer or string
F$GETJPI	Integer or string
F$GETSYI	Integer or string
F$INTEGER	Integer
F$LENGTH	Integer
F$LOCATE	Integer
F$LOGICAL	String
F$MESSAGE	String
F$MODE	String
F$PARSE	String
F$PID	String
F$PRIVILEGE	String
F$PROCESS	String
F$SEARCH	String
F$SETPRV	String
F$STRING	String
F$TIME	String
F$USER	String
F$VERIFY	Integer

For a complete description of the lexical functions, their required formats, and their use in command procedures, see Chapter 5.

3.2 EQUATING SYMBOLS TO CHARACTER STRING EXPRESSIONS

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

        symbol-name = string-expression

A string expression can contain character strings, lexical functions that evaluate to character strings, or other symbols assigned to character strings.

Character strings can contain any alphanumeric or special characters and must be enclosed in quotation marks (").

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

```
$ NAME = "MYFILE.DAT"
$ TEMP = "TEMPORARY FILE CREATED"
$ TOPIC = "THE " + TEMP
$ COUNT = F$STRING(65)
$ OUTPUTMESSAGE = "Beginning..."
$ TOTAL = "NUMBER OF FILES = " + F$STRING(B)
$ SUBTOTAL = TOTAL
```

In all of these examples, symbol names are assigned to character string expressions. Some of these expressions contain a single value that is either a character string, symbol name, or a lexical function. Notice, however, that some string expressions are composed of character strings, symbols, and lexical functions used as operands in expressions. For example, the following assignment contains the operands "THE " (character string) and TEMP (string symbol):

```
$ TOPIC = "THE " + TEMP
```

The plus sign (+) operator in this example concatenates the two operands to form a single character string that is assigned to the symbol TOPIC. So, if the symbol TEMP was assigned the value "TEMPORARY FILE CREATED", the value for the symbol TOPIC would be "THE TEMPORARY FILE CREATED". Note that both operands in the expression must be character strings for string concatenation to occur.

You can also subtract a character string from another string in an expression using the string reduction (-) operator. Note that both operands must be character strings for string reduction to occur. In a reduction operation, the character string following the minus sign is removed from the string preceding the minus sign. For example:

```
FILENAME = "TEMPFILE" - "TEMP"
```

The minus sign in this example removes the character string "TEMP" from the string "TEMPFILE". As a result, the character string "FILE" is assigned to the symbol FILENAME.

In a subtraction operation, if the string following the minus sign occurs more than once in the preceding string, only the first occurrence is removed.

3.3 EQUATING SYMBOLS TO INTEGER AND LOGICAL EXPRESSIONS

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

```
symbol-name = integer-expression
```

Some examples are:

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

An expression can be any integer value or an integer or logical expression. You can specify a value in a non decimal 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 arithmetic comparison operators or string comparison operators, it is assigned the value 1 (true) if it results in an odd numeric value; the expression is assigned the value 0 (false) if it results in an even numeric value.

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

The following sections show how to specify integer and logical 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, in lexical functions, and in all contexts in which the command interpreter automatically performs expression evaluation. The IF command is described in Chapter 6.

For clarity, the examples in this chapter show literal integer 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.

3.4 OPERATORS IN EXPRESSIONS

Table 3-3 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 3-3, where 1 is the lowest precedence and 7 is the highest. 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 3-3:   Summary of Operators in Expressions

Type	Operator	Precedence[1]	Operation
Logical Operators	.OR. .AND. .NOT.	1 2 3	Logical OR Logical AND Logical complement
Arithmetic Comparison Operators	.EQ. .GE. .GT. .LE. .LT. .NE.	4 4 4 4 4 4	Arithmetic equal to Arithmetic greater than or equal to Arithmetic greater than Arithmetic less than or equal to Arithmetic less than Arithmetic not equal to
String Comparison Operators	.EQS. .GES. .GTS. .LES. .LTS. .NES.	4 4 4 4 4 4	String equal to String greater than or equal to String greater than String less than or equal to String less than String not equal to
Arithmetic Operators	+ - + - * /	5 5 7 7 6 6	Arithmetic sum Arithmetic difference Arithmetic unary plus Arithmetic unary negate Arithmetic product Arithmetic division (integer quotient)
String Operators	+ -	5 5	String concatenation String reduction

1. Lowest precedence is 1; highest precedence is 7.

3.4.1 Logical Operations

Use logical operators to perform logical functions on integer 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 are integer values, symbol names
equated to integer values, or expressions that yield integer values.
If you specify a character string value as an operand, it is converted
to an integer value before the operation is performed.

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.

3.4.2 Arithmetic Comparisons

Use arithmetic comparison operators to compare integer 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".EQ.0	1 (true)
"123".EQ.123	1 (true)

Operands in arithmetic comparisons are integer values, symbol names
equated to integer values, or expressions that yield integer values.
If you specify a character string value as an operand, it is converted
to an integer value before the comparison is performed.

3.4.3 String Comparisons

Use string comparison operators to compare 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 3-4. 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 nulls (an ASCII value of %X00) before
  the comparison is made. Note that a null 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)
"TRUE".EQS.1	0 (false)
"FALSE".EQS.0	0 (false)
"123".EQS.123	1 (true)

Operands in string comparisons are character strings, expressions that
yield character strings, or symbol names equated to character strings.
If you specify an integer value as an operand, it is converted to a
string before the comparison is performed.

If you do not enclose a 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 3-4: 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

ZK-820-82

3.4.4 Arithmetic Operations

Use arithmetic operators to perform calculations on integers. In arithmetic operations, all nondecimal (specified by radix operators) and character string values are converted to integer values before the operation is performed, except when string concatenation and string reduction are being performed (see Section 3.4.5). The result of the arithmetic operation is an integer. 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
G = -5 + 4	G = -1

Operands in arithmetic operations are integer values, symbol names equated to integer values, or expressions that yield integer values. If you specify a string value as an operand, it is converted to an integer value before the operation is performed.

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

3.4.5 String Operations

Use string operators to either concatentate or reduce strings. In order for string concatenation and reduction to occur, both operands must be character string values.

In a string concatenation operation, the plus sign (+) operator concatenates two character strings to form a single character string. In a string reduction operation, the minus sign (-) operator subtracts one character string from another. If the string following the minus sign in a string reduction operation occurs more than once in the preceding string, only the first occurrence is removed.

Some examples of string operations are listed below:

Expression	Result
A = "MYFILE" + ".MEM"	"MYFILE.MEM"
B = "FILENAME.MEM" - "FILE"	"NAME.MEM"
C = "LISTING.LIS" - "LIS"	"TING.LIS"

Operands in string operations are string values, symbol names equated to string values, or expressions that yield string values.

3.5 SPECIAL-PURPOSE STRING ASSIGNMENTS

An optional format for assigning a character string value to a symbol name is:

        symbol-name := character string

With this form of string assignment statement, character string values are automatically converted to uppercase. Also, any leading and trailing spaces and tabs are removed, and multiple spaces and tabs between characters are compressed to a single space. The := format is useful in command procedures that manipulate file specifications or command strings. For example:

```
$ KILL := delete  sys$batch   /entry=
$ SHOW SYMBOL KILL
  KILL = "DELETE SYS$BATCH /ENTRY="
$ KILL 132
```

In this example, the command string assigned to the symbol KILL is converted to uppercase, and the multiple spaces between the command parameters are reduced to a single space.

If you want to prohibit uppercase conversion and retain required space and tab characters in a string, you must place quotation marks around the string, for example:

```
$ NAME := "job     search"
$ SHOW SYMBOL NAME
  NAME = "job     search"
```

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
```

To assign a null string to a symbol, do not specify a string. For
example:

```
$ NULL :=
```

3.6   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 allowed
between the right bracket and the colon or between the symbol name and
the left bracket. You can specify any integer expression for offset
and size. Integer 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
assignment statement syntax 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.

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.

ZK-818-82

Figure 3-1: Replacing Character Strings in Assignment Statements

3.7 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]= integer-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 colon or between the symbol name and the left bracket. The bit-position and size are integer expressions. 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.

The arithmetic overlay technique is used in the sample procedure WAKEUP.COM in Appendix A.

3.8   CHANGING THE CONTEXT OF A SYMBOL

After a symbol is defined, it can be interpreted as character string or integer 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 or reduced with another string.

*   It can be used as a logical value and tested to see whether it is 1 (true) or 0 (false).

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.

Two lexical functions, F$INTEGER and F$STRING, enable you to change the context of a particular expression. The F$INTEGER lexical function converts a string expression to an integer value. The F$STRING function converts an integer expression to a string value. For a complete description of these two lexical functions, see Chapter 5.

3.9 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.

3.9.1 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 by you at your terminal can be accessed by command procedures you run interactively, but the converse is not true.

Use a single equal sign (= or :=) 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 the following:

```
$ 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 3.10.

3.9.2 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 (== or :==) 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 to make the definitions 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.

3.9.3 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. By default, 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.

3.10 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:

```
$ DEFINE/USER_MODE INFILE 'P1'
$ DEFINE/USER_MODE OUTFILE 'P2'
$ 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 interprets them as
character string values and gives them null values by default if you
do not specify values for them.

3.10.1 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
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.

3.10.2 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.

3.10.3 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 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.
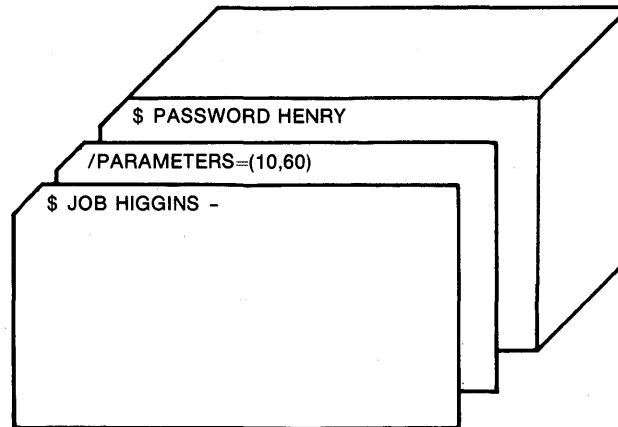
ZK-819-82

Figure 3-2: Using a /PARAMETERS Qualifier Card

3.10.4 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 an integer value, decreases the current value by one, and then stores it in a local symbol P1 as an integer value.

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.

3.11 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 executing. 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"
$ DEFINE/USER_MODE INFILE 'IN'
$ DEFINE/USER_MODE OUTFILE 'OUT'
$ RUN SORTER
```

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

```
$ @RUNTEST (RET)
  INPUT FILE: DBA1:INSORT.DAT (RET)
  OUTPUT FILE: DBA2: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, the 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 or blanks 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.

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

3.12 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. The amount of space is determined for each
  process by the SYSGEN parameter CLISYMTBL. For more
  information on this parameter, see the <u>VAX/VMS</u> <u>System</u>
  <u>Management</u> <u>and</u> <u>Operations</u> <u>Guide</u>.

- 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 evaluation (Section 4.3)

- How to use repetitive and iterative 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 a nonblank character string that is terminated with a blank or a special character. In this context, a special character is any character that is not valid in a symbol name.

If the token represents a 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 delimits the symbol name PDEL because the parentheses are valid delimiters:

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

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

4.2 USING APOSTROPHES AS SUBSTITUTION OPERATORS

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

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.

4.2.1 Substitution within Character Strings

Within character strings enclosed in quotation marks, you can request symbol substitution by placing two apostrophes before the symbol name and one apostrophe following it. 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
```


4.3 AUTOMATIC EVALUATION

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

Symbol evaluation is automatically performed in expressions in:

- Assignment statements using = or == (not := or:==)

- 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 as follows:

    $ TOTAL = COUNT + 1

No apostrophes are needed to request substitution for the symbol COUNT in this 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 example above, 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

In the first command, the command interpreter replaces the string FILENAME with its current value just after it has read the command input. The second command 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 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 iterative substitution, as described in the next section.

4.5 REPETITIVE AND ITERATIVE SUBSTITUTION

Substitution is either repetitive or iterative 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. Iterative 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.

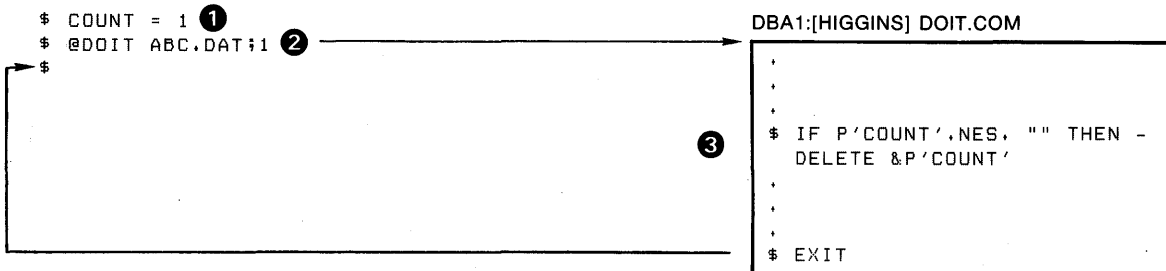### 4.5.1 Steps in Symbol Substitution

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: (1) it 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, the command interpreter replaces symbols during the actual execution of a command, for example, the IF command. This substitution is not, by default, iterative.

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                                      DBA1:[HIGGINS] DOIT.COM
$ @DOIT ABC.DAT;1 ❷ ───────────────────────────────►
▶ $
                                                      ,
                                                      ,
                                                      ,
                                         ❸    $ IF P'COUNT'.NES. "" THEN -
                                                DELETE &P'COUNT'
                                                      ,
                                                      ,
                                                      ,
                                              $ EXIT
```

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

ZK-821-82

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 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 Iterative Substitution Using Apostrophes

When you use an apostrophe to request symbol substitution, the command interpreter performs iterative 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 iterative 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

ZK-822-82

Figure 4-2: Example of Iterative Substitution

Substitution using apostrophes is not, however, iterative 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 iterative within character strings enclosed in quotation marks.

4.5.3 Iterative Substitution Using Command Synonyms

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

```
$ 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 enclose it in apostrophes, as shown below:

```
$ 'EXEC'
```

Figure 4-3 shows another example of using an apostrophe with a command synonym to force iterative substitution. The example shows the results of substitution first, without using an apostrophe and then, the results of substitution when an apostrophe is specified. The procedure in Figure 4-3 is similar to the GETPARMS.COM procedure in Appendix A.

4.5.4 Iterative 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 iterative 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:
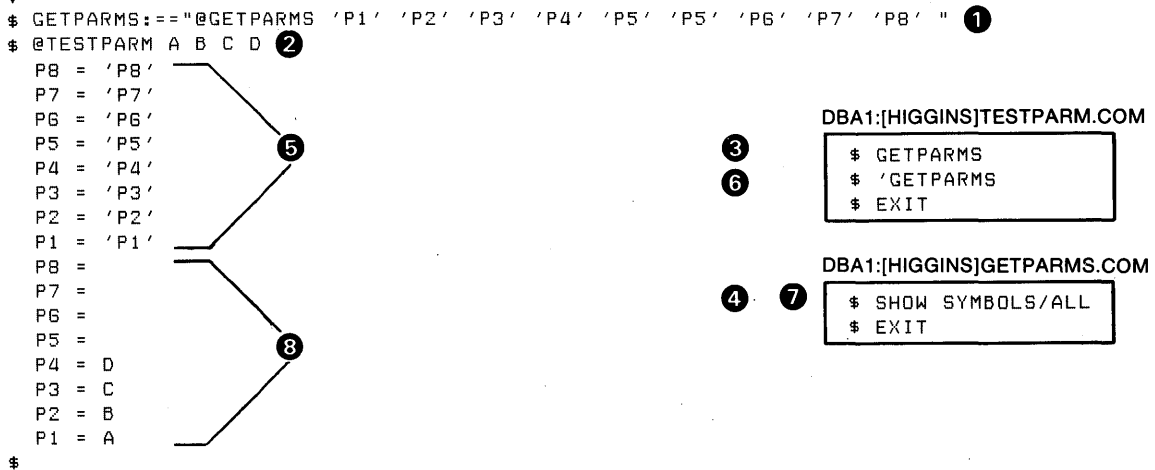
```
$ DELETE 'P''COUNT'
```

In this example, the command interpreter, during initial scanning of the command, performs normal left-to-right substitution. It attempts to replace the separate tokens P and COUNT. Because P is not a defined symbol, only COUNT is 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.

For an example of using iterative substitution to process parameters passed to a command procedure, see the procedure in Section 6.1.2.

```
   •
   •
   •
$ GETPARMS:=="@GETPARMS 'P1' 'P2' 'P3' 'P4' 'P5' 'P5' 'P6' 'P7' 'P8' "  ❶
$ @TESTPARM A B C D  ❷
   P8 = 'P8'
   P7 = 'P7'
   P6 = 'P6'                                              DBA1:[HIGGINS]TESTPARM.COM
   P5 = 'P5'                              ❺      ❸     ┌──────────────────────┐
   P4 = 'P4'                                     ❻     │  $ GETPARMS          │
   P3 = 'P3'                                           │  $ 'GETPARMS         │
   P2 = 'P2'                                           │  $ EXIT              │
   P1 = 'P1'                                           └──────────────────────┘
   P8 =
   P7 =                                                  DBA1:[HIGGINS]GETPARMS.COM
   P6 =
   P5 =                                       ❹  ❼    ┌──────────────────────┐
   P4 = D                            ❽               │  $ SHOW SYMBOLS/ALL   │
   P3 = C                                            │  $ EXIT               │
   P2 = B                                            └──────────────────────┘
   P1 = A
$
```

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.

ZK-823-82

Figure 4-3: Iterative Substitution Using a Command Synonym

4.5.5 Iterative Substitution in Expressions

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

For an example of using an apostrophe in an arithmetic assignment statement to force iteration, 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 special-purpose string assignment statements with arithmetic assignments statements. In a special purpose 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, to the command interpreter, 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 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 arithmetic and string expressions. 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. Note that 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 abbreviation.

( )

> 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.

You can specify arguments using any of the following:

- Integer values

- Numeric string or character string values (enclosed in quotation marks)

- Symbols

- Other lexical functions

- Expressions equivalent to any of the above (see Section 3.1.1 for a description of expressions)

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

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$CVSI(bit-position,width,string)	Signed value extracted from the specified string
F$CVUI(bit-position,width,string)	Unsigned value extracted from the specified string
F$CVTIME(time)	Standard ASCII date and time string of the form dd-mmm-yyyy hh:mm:ss:cc converted to a string of the form yyyy-mm-dd hh:mm:ss:cc
F$DIRECTORY()	Current default directory name string, including brackets
F$EXTRACT(offset,length,string)	Substring beginning at specified offset for specified length of indicated string
F$FAO(control-string [,arg1,arg2,...arg15])	Specified control string converted to a formatted ASCII output string
F$FILE_ATTRIBUTES (file-spec,item)	File attribute information about the specified file
F$GETDVI(device-name,item)	Specific device information about the specified device
F$GETJPI(pid,item)	Accounting, status, and identification information about the specified process
F$GETSYI(item)	Status and identification information about the system

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

Function	Value Returned
F$INTEGER(expression)	Integer equivalent of the result of the specified string expression
F$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
F$MESSAGE(status-code)	Message text associated with the specified integer status code value
F$MODE()	One of the character strings "INTERACTIVE", "BATCH", or "NETWORK"
F$PARSE(file-spec [,default-spec] [,related-spec] [,field])	Either the full file specification for the specified file or the particular file specification field that you specify
F$PID(context-symbol)	Process identification (pid) number
F$PRIVILEGE(privstates)	Either "TRUE" or "FALSE" depending on whether your current process privileges match the privilege states listed in the argument
F$PROCESS()	Current process name string
F$SEARCH(file-spec [,stream-id])	Full file specification of the file that matches the specified file specification
F$SETPRV(privstates)	List of keywords indicating the previous state of the specified privilege states; in addition, sets the process privileges to the states given in the arguments

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

Function	Value Returned
F$STRING(expression)	Numeric string equivalent of the result of the specified arithmetic expression
F$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", "BATCH", or "NETWORK".

- F$VERIFY returns an integer 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.

- F$FILE_ATTRIBUTES returns the specified item of file attribute information about the specified file. The data returned is either in integer or string format.

- F$GETDVI invokes the $GETDVI system service to return the specified item of device-related information about the specified device.

- F$GETJPI invokes the $GETJPI system service to return accounting, status, and identification information about the specified process.

- F$GETSYI invokes the $GETSYI system service to return status and identification information about the system.

- F$PARSE returns either the expanded specification for the specified file or the particular file specification field that you specify.

- F$PID returns as a character string a process identification (PID) number and updates the context symbol to remember the current position in the process list.

- F$SEARCH returns the full file specification of the file that matches the indicated file specification.

- F$PRIVILEGE returns either "TRUE" or "FALSE" depending on whether your current process privileges match the privilege states listed in the argument.

- F$SETPRV allows you to set privileges and returns a list of keywords indicating the previous state of the specified privileges.

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, but must be followed by parentheses.

For example, 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 or network job:

```
$ IF F$MODE() .NES.  "INTERACTIVE" 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 INTERACTIVE; 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 or network jobs.

5.2.2 The F$VERIFY Lexical Function

You can use the F$VERIFY function to test or to save the current
verification setting.   If you use this 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).

The format of the F$VERIFY function is:

    F$VERIFY([value])

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 disables verification. 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.

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 this example is the same as the
previous example.

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, but must be followed by
parentheses.

The following example shows how to use the F$DIRECTORY function to save the name of 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 symbol SAVE_DIR to the current directory. 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, but must be followed by parentheses.

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.

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, but must be followed by parentheses.

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 iterative, 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 a character string expression that is equivalent to the literal logical name to be translated. Since the logical name argument is case-sensitive, you must type the name in the proper case. Upper-case conversion is not performed.

You can use the F$LOGICAL function to save the current equivalence of a logical name and later restore it. The following example shows (1) the use of the F$LOGICAL function to determine the name of the current terminal device and (2) the creation of a group logical name table entry based on the equivalence string:

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

This example illustrates another important point about lexical functions: all arguments specified for lexical functions are automatically evaluated. This means that arguments that begin with alphabetic characters 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, since the argument SYS$COMMAND 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
```

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, but must be followed by parentheses.

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.

For another example of the F$TIME function, see Section 5.3.3.

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 an integer expression.

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 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.

5.2.9 The F$FILE_ATTRIBUTES Lexical Function

The F$FILE_ATTRIBUTES lexical function returns attribute information about the specified file. The data returned is either in integer or string format, depending on which item you request.

The format of the F$FILE_ATTRIBUTES function is:

```
F$FILE_ATTRIBUTES(file-spec,item)
```

The file-spec specifies the name of the file you are requesting attribute information about. You can specify the file name as a character string expression or as a symbol equated to a string expression. However, you may specify only one file name. No wild card characters are allowed in the file specification.

The item specifies which attribute of the file is to be returned. Specify the item as a character string expression or as a symbol equated to a character string. The item can be any one of the VAX-11 RMS field names listed in Table 5-2.

The following example returns the owner UIC of the file QUEST.DAT:

```
$ UIC = F$FILE_ATTRIBUTES("QUEST.DAT","UIC")
$ SHOW SYMBOL UIC
  UIC = "[023,205]"
```

5.2.10 The F$GETDVI Lexical Function

The F$GETDVI lexical function invokes the $GETDVI system service to return a specified item of information about the specified device. This function allows a process to obtain information about a device to which the process has not necessarily assigned a channel.

The format of the F$GETDVI lexical function is:

```
F$GETDVI(device-name,item)
```

The device-name is a character string expression that may evaluate to either a physical device name or a logical name equated to the physical device name. If the first character is an underscore (_),

the name is considered a physical device name. Otherwise, a single level of logical name translation is performed, and the equivalence name, if any, is used. Specify the device name as a character string expression or as a symbol equated to a character string expression.

The item specifies the type of device information to be returned. It may be specified as a character string expression or as a symbol equated to a character string expression. You can specify any one of the items listed in Table 5-3.

Table 5-2: F$FILE_ATTRIBUTES Items.

Item	Attribute	Type
ALQ	Allocation Quantity	Integer
BDT	Backup Date/Time	String
BKS	Bucket Size	Integer
BLS	Block Size	Integer
CBT	True If Contiguous-Best-Try	Boolean
CDT	Creation Date/Time	String
CTG	True If Contiguous	Boolean
DEQ	Default Extension Quantity	Integer
DID	Directory ID String	String
DVI	Device Name String	String
EDT	Expiration Date/Time	String
EOF	Number of Blocks Used	Integer
FID	File ID String	String
FSZ	Fixed Control Area Size	Integer
GRP	Owner Group Number	Integer
MBM	Owner Member Number	Integer
MRN	Maximum Record Number	Integer
MRS	Maximum Record Size	Integer
NOA	Number of Areas	Integer
NOK	Number of Keys	Integer
ORG	File Organization (SEQ, REL, IDX)	String
PRO	File Protection String	String
PVN	Prologue Version Number	Integer
RAT	Record Attributes (CR, PRN, FTN)	String
RCK	True If Read Check	Boolean
RDT	Revision Date/Time	String
RFM	Record Format String (VAR, FIX, VFC, UDF, STM, STMLF, STMCR)	String
RVN	Revision Number	Integer
UIC	Owner UIC String	String
WCK	True If Write Check	Boolean

Table 5-3: F$GETDVI Items

Item	Information Returned	Type
ACPPID	ACP process ID	String
ACPTYPE	ACP type code (F11CV1, F11CV2, MTA, NET, REM)	String
ALL	Device is allocated	Boolean
AVL	Device is available for use	Boolean
CCL	Carriage control device	Boolean
CLUSTER	Volume cluster size	Integer

Table 5-3 (Cont.): F$GETDVI Items

Item	Information Returned	Type
CYLINDERS	Number of cylinders on the volume (disk)	Integer
DEVBUFSIZ	Device buffer size	Integer
DEVCHAR	Device characteristics	Integer
DEVCLASS	Device class	Integer
DEVDEPEND	Device-dependent information	Integer
DEVDEPEND2	Additional device-dependent data	Integer
DEVNAM	Device Name	String
DEVTYPE	Device type	Integer
DIR	Device is directory structured	Boolean
DMT	Device marked for dismount	Boolean
ELG	Device has error logging enabled	Boolean
ERRCNT	Error count	Integer
EXISTS	True if device exists	Boolean
FOD	Files-oriented device	Boolean
FOR	Device is mounted foreign	Boolean
FREEBLOCKS	Number of free blocks on the volume (disk)	Integer
GEN	Device is a generic device	Boolean
IDV	Device capable of providing input	Boolean
LOGVOLNAM	Logical volume name	String
MAXBLOCK	Number of logical blocks on the volume	Integer
MAXFILES	Maximum files on volume (disk)	Integer
MBX	Device is a mailbox	Boolean
MNT	Device is mounted	Boolean
MOUNTCNT	Mount count	Integer
NET	Network device	Boolean
NEXTDEVNAM	Device name of next volume in volume set (disk)	String
ODV	Device is capable of providing output	Boolean
OPCNT	Operation count	Integer
OPR	Device is an operator	Boolean
OWNUIC	UIC of device owner	String
PID	Process identification of device owner	String
RCK	Device has read checking enabled	Boolean
REC	Device is record oriented	Boolean
RECSIZ	Blocked record size	Integer
REFCNT	Reference count of processes using device	Integer
RND	Device allows random access	Boolean
ROOTDEVNAM	Device name of root volume in volume set (disk)	String
RTM	Device is real-time	Boolean
SDI	Device is single directory structured	Boolean
SECTORS	Number of sectors per track (disk)	Integer
SERIALNUM	Volume serial number (disk)	Integer
SHR	Device is shareable	Boolean
SPL	Device is being spooled	Boolean

Table 5-3 (Cont.): F$GETDVI Items

Item	Information Returned	Type
SQD	Sequential block-oriented device (that is, magnetic tape)	Boolean
SWL	Device is software write-locked	Boolean
TRACKS	Number of tracks per cylinder (disk)	Integer
TRANSCNT	Volume transaction count	Integer
TRM	Device is a terminal	Boolean
UNIT	Unit number	Integer
VOLCOUNT	Count of volumes in volume set (disk)	Integer
VOLNAM	Volume name	String
VOLNUMBER	Number of current volume in volume set (disk)	Integer
VPROT	Volume protection mask	String
WCK	Device has write checking enabled	Boolean

The following example returns an error count for DQA0:

```
$ ERR = F$GETDVI("_DQA0","ERRCNT")
$ SHOW SYMBOL ERR
  ERR = 0  Hex = 00000000 Octal = 000000
```

5.2.11 The F$GETJPI Lexical Function

The F$GETJPI lexical function invokes the $GETJPI system service to return accounting, status, and identification information about the specified process. User privileges are required to obtain information about:

- Other processes in the same group (GROUP privilege)

- Any other process in the system (WORLD privilege)

The format of the F$GETJPI lexical function is:

    F$GETJPI(pid,item)

The pid specifies the identification number of the process for which information is being reported. When you specify the pid, you can omit the leading zeroes. Specify the pid as a character string expression or a symbol equated to a character string expression. If you specify a null string (""), the current process is used.

The item specifies the type of process information to be returned. It can be specified as a character string expression or a symbol equated to a character string expression. You may specify any one of the items listed in Table 5-4.

The following example returns the user name for the process 003B0018:

```
$ NAME = F$GETJPI("3B0018","USERNAME")
$ SHOW SYMBOL NAME
  NAME = "USER           "
```

Table 5-4:   F$GETJPI Items

Item	Information Returned	Type
ACCOUNT	Account name string (8 characters filled with trailing blanks)	String
APTCNT	Active page table count	Integer
ASTACT	Access modes with active ASTs	Integer
ASTCNT	Remaining AST quota	Integer
ASTEN	Access modes with ASTs enabled	Integer
ASTLM	AST limit quota	Integer
AUTHPRIV	Privileges the process is authorized to enable	String
BIOCNT	Remaining buffered I/O quota	Integer
BIOLM	Buffered I/O limit quota	Integer
BUFIO	Count of process buffered I/O operations	Integer
BYTCNT	Remaining buffered I/O count quota	Integer
BYTLM	Buffered I/O byte count limit quota	Integer
CPULIM	Limit on process CPU time	Integer
CPUTIM	CPU time used in hundredths of a second	Integer
CURPRIV	Process's current privileges	String
DFPFC	Default page fault cluster size	Integer
DFWSCNT	Default working set size	Integer
DIOCNT	Remaining direct I/O quota	Integer
DIOLM	Direct I/O limit quota	Integer
DIRIO	Count of direct I/O operations for the process	Integer
EFCS	Local event flags 0 through 31	Integer
EFCU	Local event flags 32 through 63	Integer
EFWM	Event flag wait mask	Integer
ENQCNT	Lock request quota remaining	Integer
ENQLM	Lock request quota limit	Integer
EXCVEC	Address of a list of exception vectors	Integer
FILCNT	Remaining open file quota	Integer
FILLM	Open file quota	Integer
FINALEXC	Address of a list of final exception vectors	Integer
FREP0VA	First free page at end of program region	Integer
FREP1VA	First free page at end of control region	Integer
FREPTECNT	Number of pages available for virtual memory expansion	Integer
GPGCNT	Global page count in working set	Integer
GRP	Group number of UIC	Integer
IMAGNAME	Current image file name	String
IMAGPRIV	Privileges the current image was installed with	String
JOBPRCCNT	Number of subprocesses owned	Integer
LOGINTIM	Process creation time	String
MEM	Member number of UIC	Integer

Table 5-4 (Cont.): F$GETJPI Items

Item	Information Returned	Type
MODE	Current process mode (BATCH, INTERACTIVE, or NETWORK)	String
OWNER	Process identification of process owner	String
PAGEFLTS	Count of page faults	Integer
PAGFILCNT	Current paging file usage	Integer
PGFLQUOTA	Paging file quota (maximum virtual page count)	Integer
PID	Process identification	String
PPGCNT	Process page count	Integer
PRCCNT	Count of subprocesses	Integer
PRCLM	Subprocess quota	Integer
PRCNAM	Process name	String
PRI	Current process priority	Integer
PRIB	Process's base priority	Integer
PROCPRIV	Process's default privileges	String
SITESPEC	Contents of per-process site-specific longword	Integer
STATE	Process state	String
S1S	Process status flags	Integer
TERMINAL	Login terminal name for interactive users	String
TMBU	Termination mailbox unit number	Integer
TQCNT	Remaining timer queue entry quota	Integer
TQLM	Timer queue entry quota	Integer
UIC	Process's UIC	String
USERNAME	User name string (12 characters filled with trailing blanks)	String
VIRTPEAK	Peak virtual address size	Integer
VOLUMES	Count of currently mounted volumes	Integer
WSAUTH	Maximum authorized working set size	Integer
WSAUTHEXT	Maximum authorized working set extent	Integer
WSEXTENT	Current working set size extent	Integer
WSPEAK	Working set peak	Integer
WSQUOTA	Working set size quota	Integer
WSSIZE	Process's current working set size	Integer

5.2.12 The F$GETSYI Lexical Function

The F$GETSYI lexical function invokes the $GETSYI system service to
return status and identification information about the system. The
format of the F$GETSYI lexical function is:

    F$GETSYI(item)

The item specifies the type of information to be reported about the
system. It can be specified as either a character string expression
or a symbol equated to a character string expression. You can
specify any one of the items listed in Table 5-5.

Table 5-5: F$GETSYI Items

Item	Information Returned	Type
SID	System identification register	Integer
VERSION	System version (8 characters filled with trailing blanks)	String
CPU	CPU type (1 represents VAX-11/780, 2 represents VAX-11/750, 3 represents VAX-11/730)	Integer

The following example returns the system identification:

```
$ SYSID = F$GETSYI("SID")
$ SHOW SYMBOL SYSID
  SID = 19923201  Hex = 01300101 Octal = 000401
```

5.2.13 The F$PARSE Lexical Function

The F$PARSE lexical function returns either the expanded file specification for the file you designate, or the particular file specification field that you request.

The format of the F$PARSE lexical function is:

```
F$PARSE(file-spec[,default-spec] [,related-spec] [,field])
```

The file-spec is a character string expression that specifies the name of the file to be returned. The device and directory names, if omitted, default to your current default disk and directory names, unless the missing fields are replaced by a default-spec or related-spec. If you omit the file name, file type, or version number, a null () specification is returned unless the missing field is specified as a default-spec or related-spec. Wild card characters are allowed in the file specification.

The default-spec is a character string expression that is a file specification. It is substituted in the output string if a particular field in the file specification is missing. You can make further substitutions in the file specification by specifying a related-spec as a character string expression. For a further description of how to use the default-spec and related-spec, see the VAX-11 Record Management Services User's Guide.

If an error is detected during the parse, the F$PARSE function returns a null string.

NOTE

> When specifying directories and devices, whether explicitly (as a device-spec or a related-spec) or implicitly (by current default), the specified device must contain the specified directory. If the directory is not on the device, a null string is returned.

In the following example, the F$PARSE lexical function returns the expanded file specification for the file JAMES.MAR:

```
$ SET DEF DBA2:[FIRST]
$ SPEC = F$PARSE("JAMES.MAR","[ROOT]")
$ SHOW SYMBOL SPEC
  SPEC = "DBA2:[ROOT]JAMES.MAR;"
```

The default device and directory in this example is DBA2:[FIRST]. Since the directory name [ROOT] is specified as the default directory in the symbol assignment, it is substituted for [FIRST] in the output string. If [ROOT] was not on DBA2:, however, a null string would be returned. Note that the default device returned in the output string is DBA2: and the default version number for the file is null.

The field argument causes the F$PARSE lexical function to return a specific portion of a file specification. The field name cannot be abbreviated. You can specify any one of the following field names:

```
NODE           Node name
DEVICE         Device name
DIRECTORY      Directory name
NAME           File name
TYPE           File type
VERSION        File version number
```

The F$PARSE lexical function in the following example returns the directory name of the specified file:

```
$ SET DEFAULT DB1:[VARGO]
$ SPEC = F$PARSE("INFO.COM",,,"DIRECTORY")
$ SHOW SYMBOL SPEC
  SPEC = "[VARGO]"
```

Note that since the default-spec and related-spec are omitted from the argument list, commas (,) must be inserted in their place.

5.2.14 The F$PID Lexical Function

The F$PID function returns, as a character string, a process identification (PID) number and updates the context symbol to remember the current position in the process list. You can use the F$PID function to obtain the PIDs of all processes in your group or on the system. The format of the F$PID lexical function is:

```
F$PID(context-symbol)
```

If the context symbol is undefined or equated to a null string (""), F$PID returns the first PID in the process list. The PID of your process is returned if you lack GROUP or WORLD privilege. If you have GROUP privilege, the first PID in your group list is returned. If you have WORLD privilege, the first PID in the system list is returned.

Once the context symbol is initialized, each subsequent F$PID function returns the next PID in sequence, and updates the context symbol. After the last PID in the process list is returned, the F$PID function returns a null string.

The following command procedure displays a list of PIDs dependent on your process's privileges:

```
$ CONTEXT = ""
$ START:
$     PID = F$PID(CONTEXT)
$     IF PID .EQS. "" THEN EXIT
$     SHOW SYMBOL PID
$     GOTO START
```

The PIDs displayed by this command procedure depend on the privilege of your process. When run with GROUP privilege, the PIDs of users in your group are displayed. In order for the command procedure to return all the PIDs on the system, your process would require WORLD privilege. Without GROUP or WORLD privilege, only your PID is displayed.

5.2.15 The F$SEARCH Lexical Function

The F$SEARCH lexical function returns, as a character string, the full file specification that matches the file-spec you name. The format of the F$SEARCH lexical function is:

```
F$SEARCH(file-spec[,stream-id])
```

The file-spec may contain one or more blank or wild card fields (for example, directory name, file name, and file type, and so on) of a file specification. A file-spec must be a character string expression or a symbol equated to a character string.

The stream-id can be any positive integer. You use it to specify the search stream when you want to maintain the context for several searches performed simultaneously. For example:

```
$ START:
$     COM = F$SEARCH(".COM.",1)
$     DAT = F$SEARCH(".DAT.",2)
$     SHOW SYMBOL COM
$     SHOW SYMBOL DAT
$     IF (COM.EQS."") .OR. (DAT.EQS."") THEN EXIT
$     GOTO START
```

This command procedure searches the default disk and directory for COM and DAT files simultaneously. Notice that the stream-id is specified for each F$SEARCH function so that context for each function is maintained throughout the search.

If you omit the stream-id, F$SEARCH assumes an implicit single stream. Each subsequent F$SEARCH function, having the same file-spec, returns the next file specification that matches the specified file-spec. After the last file specification is returned, the next F$SEARCH function returns a null string.

The following command procedure displays the file-specs of all of .EXE files in the SYS$SYSTEM directory:

```
$ START:
$     FILE = F$SEARCH("SYS$SYSTEM:*.EXE")
$     IF FILE .EQS. "" THEN EXIT
$     SHOW SYMBOL FILE
$     GOTO START
```

5.2.16   The F$PRIVILEGE Lexical Function

The F$PRIVILEGE lexical function returns a value of either "TRUE" or "FALSE" depending on whether your current process privileges match the privilege states listed in the argument.

The format of the F$PRIVILEGE function is:

    F$PRIVILEGE(privstates)

The privstates argument must be a character string expression equated to a privilege keyword or a list of privilege keywords separated by commas. Part I of the VAX/VMS Command Language User's Guide contains a list of the valid privilege names you can specify as keywords.

If "NO" precedes the privilege keyword, then the privilege must be disabled in order for the function to return a "TRUE" value. The F$PRIVILEGE function checks each of the keywords in the specified list, and if the result for any one is false, the string "FALSE" is returned.

For example, suppose your process has OPER, USER, TMPMBX, and NETMBX privileges, and you specify the following:

    $ PROCPRIV = F$PRIVILEGE("OPER,GROUP,TMPMBX,NONETMBX")
    $ SHOW SYMBOL PROCPRIV
      "FALSE"

In this example, a value of "FALSE" is returned because your process has NETMBX privilege, but NONETMBX was specified in the privstate list. Although the boolean result for the other three keywords is true, the entire expression is declared false since the result for NONETMBX was false.

5.2.17   The F$SETPRV Lexical Function

The F$SETPRV lexical function invokes the $SETPRV system service to enable or disable specified user privileges and return a list of keywords indicating the previous state of privileges for the current process. If a privilege cannot be set, the function will return the specific privilege keyword prefixed by "NO".

The format of the F$SETPRV function is:

    F$SETPRV(privstates)

The privstates argument is a character string expression that evaluates to a privilege keyword or a list of privilege keywords separated by commas. Part I of the VAX/VMS Command Language User's Guide contains a list of valid privilege names you can specify as keywords.

In addition to the privilege names listed in the VAX/VMS Command Language User's Guide, you can specify either of the following privilege keywords:

    ALL   Enables all known privileges
    NOALL   Disables all known privileges

In order to enable privileges, your process must be authorized to set the specified privilege. For detailed information on privilege restrictions, see the VAX/VMS System Services Reference Manual.

The following example enables and disables certain privileges and returns a list of keywords indicating the previous state of the specified privileges:

```
$ OLDPRIV = F$SETPRV("OPER,NOTMPMBX")
$ SHOW SYMBOL OLDPRIV
   "NOOPER,TMPMBX"
```

In this example, the privilege OPER was enabled and TMPMBX was disabled. The keywords NOOPER and TMPMBX were returned in the keyword list, showing the previous state of these privileges.

5.3   STRING MANIPULATION FUNCTIONS

A string can be either a literal character string (one enclosed in quotation marks), a symbol name that has been equated to a string value, or an expression that evaluates to a string. 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 leftmost character in the string).

The following lexical functions allow you to manipulate character strings:

- F$LENGTH returns the length of a specified string as an integer value.

- F$LOCATE returns the offset within a string of a specified character or character substring as an integer value.

- F$EXTRACT returns a substring from within a specified character string as a string value.

- F$INTEGER returns the integer equivalent of the specified string expression.

- F$STRING returns the string equivalent of the specified integer expression.

- F$CVTIME returns the standard ASCII date and time string of the form dd-mmm-yyyy hh:mm:ss:cc converted to a string of the form yyyy-mm-dd hh:mm:ss:cc.

- F$FAO returns the specified control string converted to a formatted ASCII output string.

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 a character string expression or a symbol equated to a character string expression.

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.

For additional examples on the use 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 as a character string expression or a symbol equated to character string expression.

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 NO_COLON
```

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.

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 as an integer expression or a symbol equated to an integer expression. Specify string as a character string expression or a symbol equated to a character string expression.

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:

```
$ IF F$EXTRACT(12,2,F$TIME()) .GES. 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 start of the string.

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

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. For example:

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

In this example, the lexical function F$LOCATE gives the numeric value representing the position of a period in the character string value of P1. This function is used as an argument in the F$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 fields such as these would require a more complicated sequence of functions.

5.3.4 THE F$INTEGER Lexical Function

The F$INTEGER lexical function returns the integer equivalent of the result of the specified integer expression. You can use this function to set a symbol to an integer value and use the symbol in an operation that requires an integer type value.

The format of the F$INTEGER function is:

    F$INTEGER(expression)

The following example shows how the F$INTEGER function is used to
convert a string expression:

    $ A = "23"
    $ B = F$INTEGER("-9" + A)
    $ SHOW SYMBOL B
      B = -923 Hex=FFFFFC65 Octal=176145

The F$INTEGER function in the above example returns the integer
equivalent of the expression ("-9" + A), which evaluates to the
character string "-923". Here, since the expression contains two
character strings, the plus sign (+) is a string concatenation
operator. Note that the symbol A was assigned the numeric character
string "23".

For additional details on implicit string to integer conversion, see
section 3.1.2.

5.3.5 The F$STRING Lexical Function

The F$STRING lexical function converts the result of the specified
integer expression to a string. The format of the F$STRING function
is:

    F$STRING(expression)

The following example shows how the F$STRING function is used to
convert an integer expression:

    $ A = 5
    $ B = F$STRING(-2 + A)
    $ SHOW SYMBOL B
      B = "3"

The F$STRING function in the above example converts the result of the
expression, (-2 + A) to the numeric string, "3". Note that the symbol
A was assigned the integer value 5.

For additional details on implicit integer to string conversion, see
section 3.1.2.

5.3.6 The F$CVTIME Lexical Function

The F$CVTIME lexical function converts a standard ASCII date/time
string of the form "dd-mmm-yyyy hh:mm:ss.cc" to a string of the form
"yyyy-mm-dd hh:mm:ss.cc". You can use the resultant string to compare
two dates (using .LTS and .GTS. operators).

The format of the F$CVTIME function is:

    F$CVTIME(time)

The time must be specified as a string expression or a symbol equated
to a string expression in the form "dd-mmm-yyyy hh:mm:ss.cc". The
time specification must contain some part of the date field, or an
error is displayed. Absolute time keywords (TODAY, TOMORROW, and so
on) are not allowed in the time specification.

The following example gets the system time and converts it to a
comparison time:

```
$ TIME = F$TIME()
$ SHOW SYMBOL TIME
  TIME = "13-JUN-1982 10:56:23.10"
$ TIME = F$CVTIME(TIME)
$ SHOW SYMBOL TIME
  TIME = "1982-06-13 10:56:23.10"
```

5.3.7 The F$FAO Lexical Function

The F$FAO lexical function invokes the $FAO system service to convert
the specified control string to a formatted ASCII output string. By
specifying arguments, you can use the F$FAO function to:

   ● Insert variable character string data into an output string

   ● Convert integer values into the ASCII representations of their
     decimal, hexadecimal, and octal equivalents, and substitute
     results into the output string

The format of the F$FAO function is:

    F$FAO(control-string[,arg1,arg2...arg15])

The control string may be any length and may contain any number of FAO
directives (see Table 5-6). These directives control the processing
performed by the F$FAO lexical function. An FAO directive may have
any one of the following formats:

Format	Function
!DD	One directive
!n(DD)	A directive repeated a specified number of times
!lengthDD	A directive generating an output string of a specified length
!n(lengthDD)	A directive repeated a specified number of times generating output fields of a specified length

The exclamation point (!) indicates that the following character or
characters are to be interpreted as an FAO directive. DD is a one- or
two-character uppercase code indicating the action that F$FAO is to
perform. When specifying repeat counts, n is a decimal value
specifying the number of times the directive is to be repeated.   The
length value is a decimal value that instructs F$FAO to generate an
output field of "length" characters.

Each directive may require one or more arguments to the F$FAO lexical
function.   For each directive that uses an argument, you must supply
the argument as an integer expression or character string expression
in the argument list. If you specify an argument whose type (integer
or string) does not match that of directive it follows, unpredictable
results will be returned.   You can use the F$INTEGER and F$STRING
lexical functions to convert arguments to the proper type.

In the following example, the FAO directive !SL converts the number equated to the symbol COUNT to a character string.

```
$ COUNT = 57
$ REPORT = F$FAO("NUMBER OF FORMS = !SL",COUNT)
```

Note that COUNT is assigned a integer value of 57. The F$FAO function returns the ASCII string, "NUMBER OF FORMS = 57".

Directives may also include:

● A repeat count

● An output field length

A repeat count is specified as follows:

```
!n(DD)
```

In the following example, the symbols A, B, C, and D are inserted into the control string.

```
$ A = "ERR"
$ B = "IS"
$ C = "HUM"
$ D = "AN"
$ PHRASE = F$FAO("TO !3(AS)",A,B,C+D)
```

The resultant string is "TO ERRISHUMAN". Since the specified repeat count for the !AS directive is 3, F$FAO looks for three arguments. The arguments in this example include the symbol A ("ERR"), the symbol B ("IS"), and the expression C+D ("HUMAN"). Note that the values of these string arguments are concatenated to form the string "ERRISHUMAN".

Arguments must correspond exactly to the order of their respective directives. In most cases, an error message is not displayed if you misplace an argument.

If there are not enough arguments listed, F$FAO will continue reading past the end of an argument list. Therefore, always be sure to include enough arguments to satisfy the requirements of all the directives in a control string.

An output field length is specified as follows, where length is a decimal value instructing F$FAO to place the output from a directive into a field of "length" characters in the output string.

```
!lengthDD
```

A directive may contain both a repeat count and an output length specified as:

```
!n(lengthDD)
```

Repeat counts and output lengths may also be specified as variables by specifying a number sign (#) in place of an absolute numeric value. If # is specified for a repeat count, the next argument passed to F$FAO must contain the count as a binary expression. The same is true when specifying # for a length value.

When a variable output field is specified with a repeat count, only one length parameter is required, since each output string will have the specified length.

For example:

```
$ A = "ERR"
$ B = "IS"
$ C = "HUMAN"
$ PHRASE = F$FAO("TO !#(#AS)",3,6,A,B,C)
```

The string returned by the F$FAO lexical function in this example is:

"TO ERR   IS   HUMAN "

Each argument string is output to a field having a length of six characters. Since each string is less than six characters, each field is padded with blank spaces.

Table 5-6 summarizes the FAO directives and lists the arguments required by each directive.

Table 5-6:   Summary of FAO Directives

Instruction	Description	Parameter
Character string insertion		
!AS	Inserts a character string as is; the field length defaults to the length of the character string; short values inserted in explicit-length fields are left-justified and blank filled; long values inserted in explicit-length fields are truncated on the right	Character string
Zero-filled numeric conversion		
!OB !OW !OL !XB !XW !XL !ZB !ZW !ZL	Converts a byte, word, or longword to octal notation Converts a byte, word, or longword to hexadecimal notation Converts a byte, word, or longword to decimal notation	A binary number; for byte and word conversions only the low-order 8 or 16 bits are used
	Output field lengths default to 2 (byte), 4 (word), and 8 (longword) for hexadecimal -- 3, 6, and 11 for octal -- and the required number of characters for decimal; the numbers are right-justified and zero-filled on the left; explicit-length fields longer than the default are blank-filled on the left; explicit-length fields shorter than the default are truncated on the left	
Blank-filled numeric conversion		
!UB !UW !UL !SB	Converts a byte, word, or longword to decimal notation without adjusting for negative numbers Converts a byte, word, or longword	A binary number; for byte and word conversions only the

Table 5-6 (Cont.): Summary of FAO Directives

Instruction	Description	Parameter
!SW !SL	to decimal notation with negative numbers converted properly	low-order 8 or 16 bits are used
	Output field lengths default to the required number of characters; values shorter than explicit-length fields are right-justified and blank-filled; values longer than explicit-length fields cause the field to be filled with asterisks	
Special formatting		
!/ !_ !⁻ !%S !n<...!> !nc **Time** !%T !%D	Inserts a carriage return Inserts a tab Inserts a form feed Inserts an uppercase S if the most recently converted number is not 1 Left-justifies and blank-fills all data represented by the instructions ... in fields n characters wide Repeats the character represented by c for n times Inserts the current time Inserts the current date/time	None Binary number 0 Binary number 0
Parameter interpretation		
!- !+	Reuses the last parameter Skips the next parameter	None

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,size]= integer-expression

This method is described in Section 3.7, 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 8.2.

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

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

These integer conversion functions extract bit fields from character string 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,string)

    F$CVSI(bit-position,width,string)

The bit-position is the offset of the value to be converted from the beginning of the string; the width is the number of bits that are to be extracted for conversion to an integer value; and the string is the the character string from which the bits are taken. The low-order (rightmost) bit of a string is position number 0 for determining the offset. You specify the value of the string by using an integer expression.

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 can now 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.

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 the value of an expression and executes a given command if the result of the expression is true. An expression is true if:

- It has an odd integer value.

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

- It has an odd numeric string value

An expression is false if:

- It has an even integer value.

- It has a character string value that begins with any letter except Y, y, T, or t.

- It has an even numeric string value

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 integer value, is equated to a character string that begins with the letters Y, y, T, or t (true), or has an odd numeric string value
$ IF .NOT. A THEN command	Executes the given command if the symbol A has an even integer value, is equated to a character string that begins with any letter except Y, y, T, or t (false), or has an even numeric string value
$ 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 132 characters.
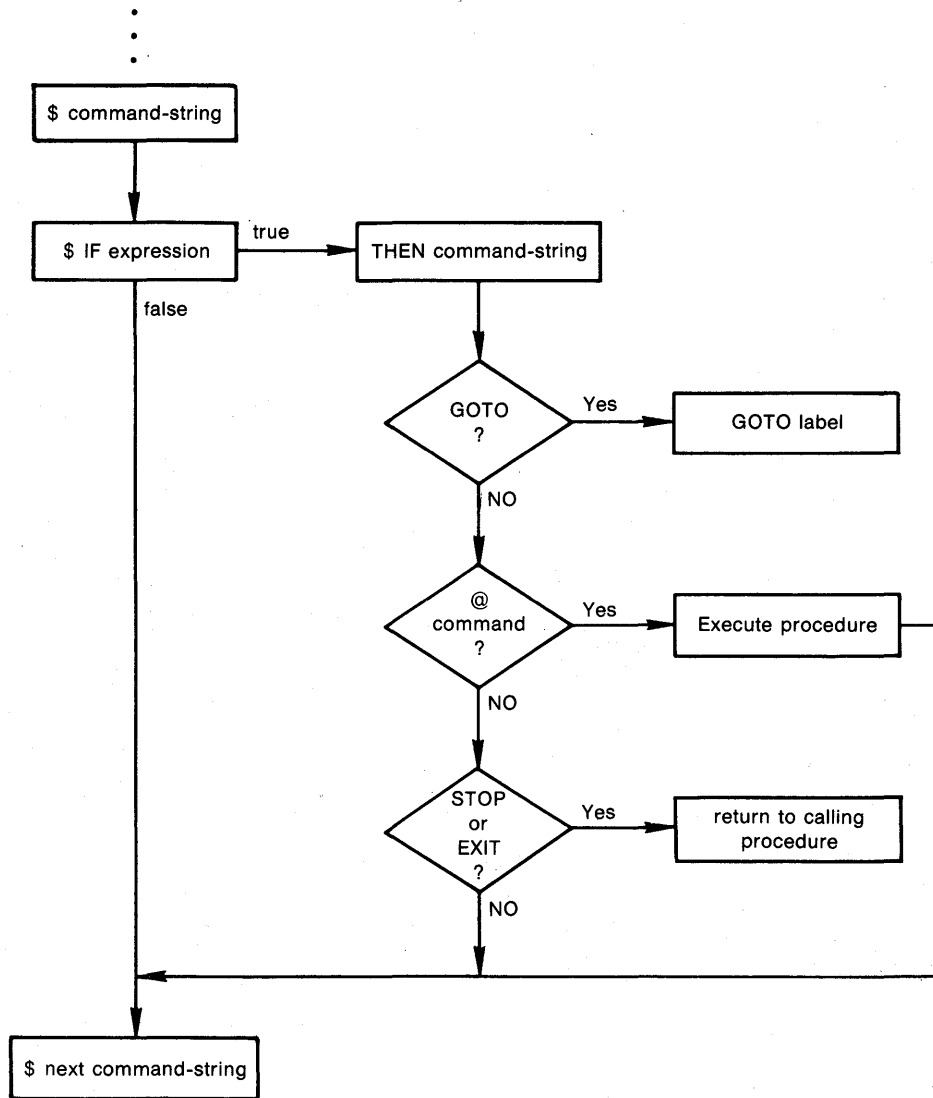
Figure 6-1: The IF Command

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 operations.

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 a letter other than T, t, Y, or y, 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 an integer 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 3.4.

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 iterative; that is, each symbol is replaced only once. However, if you want iterative substitution, you can precede a symbol name with an apostrophe (') or ampersand (&) operator so the command interpreter will also 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 iterative substitution are described in detail in Chapter 4. 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 check iteratively 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
```

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 iterative substitution. Iterative 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 nested command level; 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. 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.

ZK-825-82

Figure 6-2:   The GOTO Command

6.2.1 Using GOTO Within a THEN Clause

The GOTO command is especially useful within a THEN clause to 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
        .
        .
        .
```

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:
```

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 RETURN) 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 divide 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 using 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 3.10.

- 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 3.9.2.

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 STOP command 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 before 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. Use of the EXIT command, however, is recommended.

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
    •
    •
    •
```

and 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.

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.

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. 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.

The format of the ON command is:

        ON severity-level THEN [$] command

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.

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

```
                              DBA1:[HIGGINS]FORT.COM
    '
    '
    '                         ┌────────────────────────────┐
 $ @FORT    ──────────▶       │  '                         │
                              │  '                         │
        ┌────▶                │  '                         │
        │                     │ $ ON ERROR THEN CONTINUE   │  ❶
        │                     │ $ FORTRAN A                │  ❷
        │                     │  '                         │
        │                     │  '                         │
        │                     │  '                         │
        │◀────────────────────│ $ FORTRAN B                │
        │                     │  '                         │
        │                     │  '                         │
        │                     │  '                         │
        │                     │ $ ON WARNING THEN EXIT     │  ❸
        │                     │  '                         │
        │                     │  '                         │
        │                     │  '                         │
        │◀────────────────────│ $ FORTRAN C                │  ❹
        │                     │  '                         │
        │                     │  '                         │
        │                     │  '                         │
        └─────────────────────│ $ EXIT                     │
                              └────────────────────────────┘
```
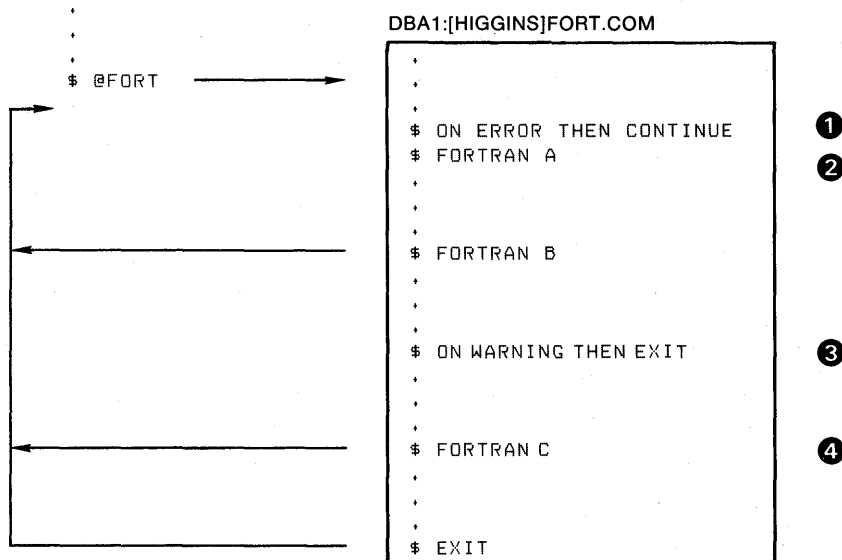
❶ This ON command overrides the default command action (on warning, continue; on error or severe error, exit). If an error or severe error occurs while A.FOR is being compiled, the command procedure continues with the next command.

❷ 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.

❸ If the command procedure does not exit before this command is executed, this command action takes effect.

❹ If a warning, error, or severe error occurs while C.FOR is being compiled, the command procedure exits.

ZK-826-82

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	IF
DEPOSIT	SHOW
EOD	STOP
EXAMINE	WAIT
GOTO	

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

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, DEPOSIT, SPAWN and ATTACH 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 EXIT command to terminate the program's execution.

- 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 before the new image is initiated.*

You cannot resume the execution of a privileged program after interrupting it by pressing CTRL/Y. For more information about privileged programs, see the software installation guide for your VAX-11 processor.

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 issue at the CTRL/Y level without causing the current image to be stopped.

---

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.

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 automatically reset the
command procedure default CTRL/Y 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

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.

The sample procedures EDITALL.COM and FORTUSER.COM in Appendix A
illustrate CTRL/Y action handling.
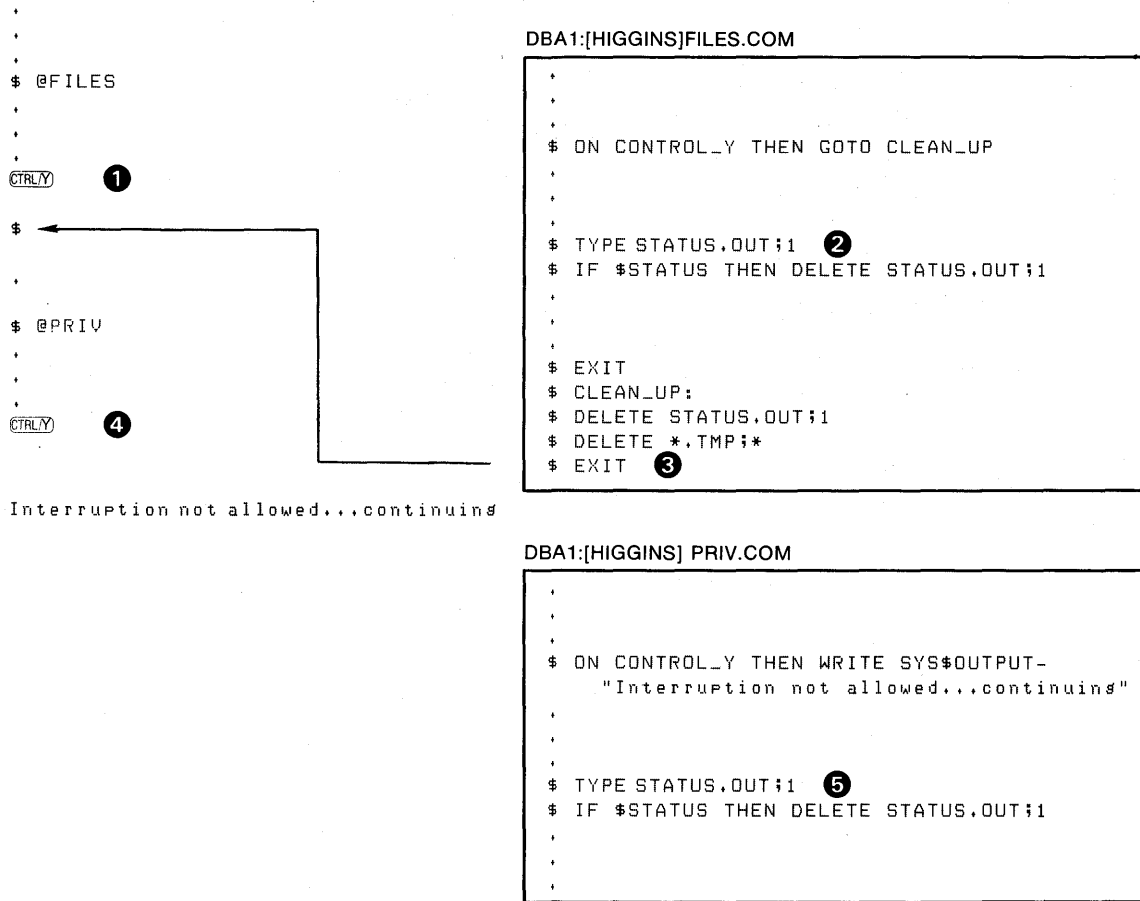
```
   +
   +
   +
$ @FILES
   +
   +
   +
(CTRL/Y)      ❶
   +
$ ◄─────────────────────────┐
   +                        │
   +                        │
$ @PRIV                     │
   +                        │
   +                        │
   +                        │
(CTRL/Y)      ❹            │
                            │
                            │
                     ───────┘

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.

ZK-827-82

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                DBA1:[HIGGINS]SEARCH.COM
      '
      '                    ┌──────────────────────────────────────┐
      '                    │  '                                   │
      '                    │  '                                   │
  (CTRL/Y)     ❶           │  '                                   │
                           │  $ ON CONTROL_Y THEN GOTO CLEAN_UP   │
                           │  '                                   │
      '                    │  '                                   │
      '                    │  '                                   │
      '                    │  $ @SUBSEARCH                        │
      '                    │  $ NEXT_STEP:                        │
                           │  '                                   │
                           │  '                                   │
                           │  '                                   │
                           │  $ EXIT                              │
                           │  $ CLEAN_UP:                         │
                           │  '                                   │
                           │  '                                   │
                           │  '                                   │
                           └──────────────────────────────────────┘


                           DBA1:[HIGGINS] SUBSEARCH.COM

                           ┌──────────────────────────────────────┐
                           │  '                                   │
                           │  '                                   │
  (CTRL/Y)     ❷           │  '                                   │
                           │  $ @SUBSUB                           │
      '                    │  '                                   │
      '                    │  '                                   │
      '                    │  '                                   │
                           └──────────────────────────────────────┘


                           DBA1:[HIGGINS] SUBSUB.COM

                           ┌──────────────────────────────────────┐
                           │  '                                   │
                           │  '                                   │
  (CTRL/Y)     ❸           │  '                                   │
                           │  $ ON CONTROL_Y THEN SHOWTIME        │
                           │  '                                   │
                           │  '                                   │
                           │  '                                   │
                           │  $ EXIT                              │
                           └──────────────────────────────────────┘
```
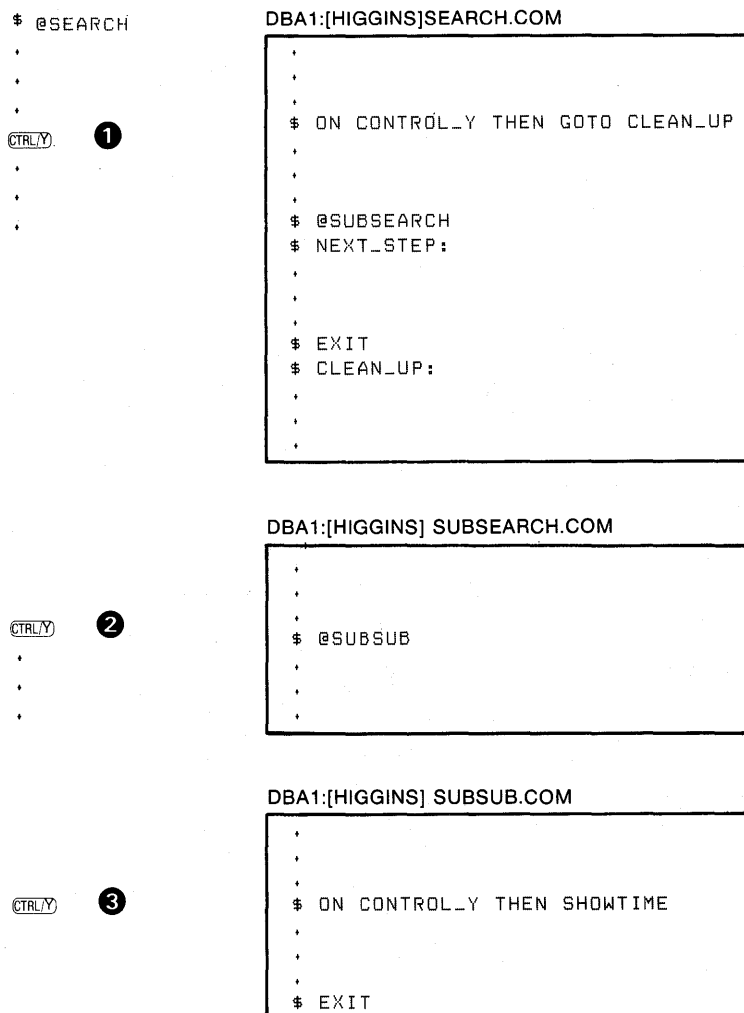
❶ If a CTRL/Y interrupt occurs while SEARCH.COM is executing, control is transferred to the label CLEAN__UP.

❷ 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.

❸ If a CTRL/Y interrupt occurs while SUBSUB.COM is executing, the SHOW TIME command is executed.

ZK-828-82

Figure 7-3:   Default CTRL/Y Action for Nested Procedures

7.2.3   Disabling CTRL/Y Interruptions

The SET NOCONTROL=Y command disables CTRL/Y handling completely: that is, if a command procedure executes the SET NOCONTROL=Y command, pressing 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.

                               CAUTION

            The ON CONTROL_Y and SET NOCONTROL=Y
            commands are intended for special
            applications. It is not recommended, in
            general, 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.

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 can combine DCL commands with the programming and symbolic capabilities of command procedures to manipulate sequential and indexed sequential access method (ISAM) files. Included are techniques for using:

- The OPEN command to create new files or access existing files

- The READ command to read from files

- The WRITE command to write to files

- 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 from or write to 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.

DBA1:[HIGGINS]FILES.COM **❶**

```
$ OPEN/READ INFILE DATA.TST
$ OPEN/WRITE OUTFIL DATA.OUT
·
·
·
$ READLOOP:
$ READ INFILE RECORD
$ WRITE OUTFIL RECORD
·
·
·
$ FINISH:
$ CLOSE INFILE
$ CLOSE OUTFIL
```

❷

DBA1:[HIGGINS]DATA.TST

❸

DBA1:[HIGGINS]DATA.OUT

❹

← MAXIMUM OF 255 BYTES →
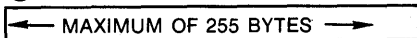
❶ The command procedure, FILES.COM

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

ZK-829-82

Figure 8-1: Steps in Reading and Writing Files

8.1 OPENING FILES

With the OPEN command you can open sequential and indexed sequential access method (ISAM) files for either reading or writing. When you open a file, you specify whether it is to be read or written and assign it a logical name that is placed in the process logical name table. This logical name is used by subsequent READ and WRITE commands to reference the file.

If the output file specification on an OPEN/WRITE command does not include a file version number and if a file with the specified file name and file type already exists, the WRITE command creates a new file with a version number one greater than the existing file, unless the /APPEND qualifier is specified. The OPEN/WRITE/APPEND command opens the file and moves the record pointer to the end-of-file, allowing new records to be appended to the end of an existing file.

The logical devices SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, and SYS$ERROR do not have to be explicitly opened before they can be read or written at the command level. All other files must be explicitly opened.

You can issue more than one OPEN command for the same file and assign it different logical names. If you specify the same logical name on more than one OPEN command without first closing the file, the file is not opened, and no warning message is issued.

The OPEN command also allows you to open a file as a shareable file to allow other users simultaneous read and/or write access. You can open a file as shareable using the /SHARE qualifier.

You should be sure that your command procedure closes any open files before the procedure terminates. If you fail to close an open file, the file remains open, and the logical name assigned to the file is not deleted from the logical name table.

For a description of the OPEN command, its format, and qualifiers, see the VAX/VMS Command Language User's Guide.

8.2 READING FILES

With the READ command, you can read sequential or indexed sequential access method (ISAM) files in which all records are less than or equal to 1024 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 four of the factors you must consider when reading files: how to define symbol names, how to handle end-of-file conditions, how to read records randomly from ISAM files, and how to delete records from ISAM files.

8.2.1 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.

8.2.2 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.

8.2.3 Reading Records Randomly from ISAM Files

You can use the READ command with the /INDEX and /KEY qualifiers to
read records randomly from indexed sequential access method (ISAM)
files. The /KEY and /INDEX qualifiers specify that a record should be
read from the file by finding the specified key in the index, and
returning the record associated with that key. If you do not specify
an index, the primary index, 0, is used.

Once a record is read randomly, you can read the remainder of the file
sequentially from that point by issuing READ commands without the /KEY
or /INDEX qualifiers.

For a description of the READ command and the /INDEX and /KEY
qualifiers, see the VAX/VMS Command Language User's Guide.

8.2.4 Deleting Records from ISAM Files

You can use the READ command with the /DELETE qualifier to delete
records from indexed sequential access method (ISAM) files. The
/DELETE qualifier causes a record to be deleted from a file after it
has been read. Use the /DELETE qualifier with the /INDEX and /KEY
qualifiers to delete a record specified by a given key.

For a description of the READ command and the /DELETE qualifier, see
the VAX/VMS Command Language User's Guide

8.3 WRITING FILES

The WRITE command can write records only to sequential files or
indexed sequential access method (ISAM) files that have been opened
for writing.

When the WRITE command writes a record, it always positions the record
pointer following the record just written.

8.3.1 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.

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, iterative 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.

8.3.2 Updating Records Using the WRITE Command

You can use the WRITE command with the /UPDATE qualifier to change a record rather than insert a new one.

8.4 APPENDING RECORDS TO EXISTING FILES

The OPEN/WRITE/APPEND command allows you to append records to the end of an existing file. When a file is opened by this command, the record pointer is positioned at the end-of-file. Thus, any records that are written to the file are added to the end of the file.

Figure 8-2 illustrates different ways of specifying data for the WRITE command.

```
     .
     .
     .
  $ @FIGURE         ──────────────────►   DBA1:[HIGGINS]FIGURE.COM
                                         ┌─────────────────────────────────────┐
  ┌─►$ TYPE DATA.OUT                      │ $ ABC := "the character string"     │
  │   the character string                │ $ COUNT = 4                         │
  │   ABC                                 │ $ P4 := fourth parameter            │
  │   COUNT IS 4                          │ $ OPEN/WRITE OUTFILE DATA.OUT       │
  │   MODE IS INTERACTIVE       ❶         │ $ WRITE OUTFILE ABC                 │
  │   FOURTH PARAMETER          ❷         │ $ WRITE OUTFILE "ABC"               │
  │   $                         ❸         │ $ WRITE OUTFILE "COUNT IS ",COUNT   │
  │                             ❹         │ $ WRITE OUTFILE "MODE IS ''F$MODE()'"│
  │                             ❺         │ $ WRITE OUTFILE P'COUNT'            │
  │                                       │ $ CLOSE OUTFILE                     │
  └───────────────────────────            └─────────────────────────────────────┘
```
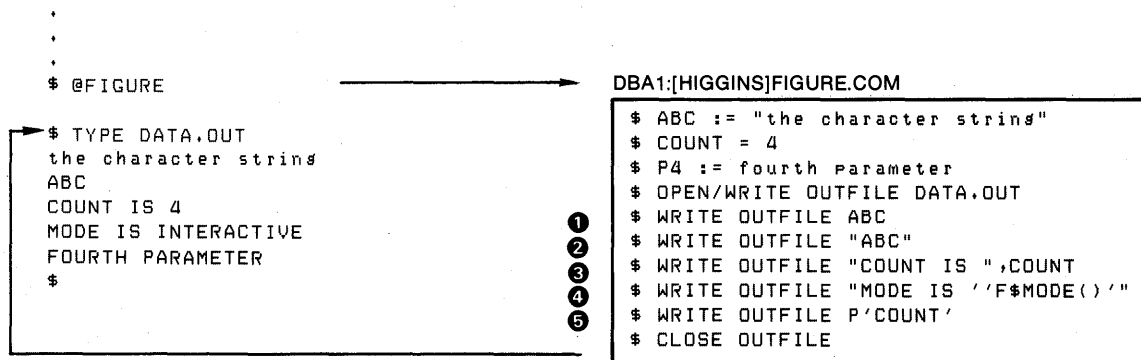
❶ The WRITE command automatically performs symbol substitution on characterstrings that are not enclosed in quotation marks; substitution is not recursive.

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.

ZK-830-82

Figure 8-2: Symbol Substitution with the WRITE Command

8.5 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 throughout 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.

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$OUTPUT       "Error       opening       file:
'F$MESSAGE($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.

8.6 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.

8.7 FILE FORMATS

You can use the READ command to read any existing sequential or indexed sequential access method (ISAM) file. The maximum record length that the READ command accepts is 1024 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 two-byte header for each record.

These files are therefore not compatible with files created by the SOS editor 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/SOS/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         │
                                             │
  $
```

```
┌──────────────────────┐       DBA1:[HIGGINS]TESTALL.COM
│ Command interpreter   │
│ finds TESTALL.COM     │      ┌─────────────────────┐
│ on default device     │      │ $ RUN A             │
│ and directory...      │      │ $ RUN B             │
└──────────────────────┘      │ $ RUN C             │
                               └─────────────────────┘
┌──────────────────────┐
│ then requests queue   │
│ for the batch job     │
└──────────────────────┘

                                        SYS$BATCH QUEUE
┌──────────────────────┐      ┌─────────────────────┐
│ TESTALL.COM gets a    │      │   *                 │
│ job number and is     │- - ->│   *                 │
│ placed in SYS$BATCH   │      │   *                 │
│ queue                 │      │ JOB NUMBER 208      │
└──────────────────────┘      │ JOB NUMBER 209      │
                               │ JOB NUMBER 210      │
┌──────────────────────┐      │   *                 │
│ Command interpreter   │      │   *                 │
│ returns job informa-  │      │   *                 │
│ tion (and control)    │      └─────────────────────┘
│ to interactive        │
│ command level         │
└──────────────────────┘

┌────────────────────────────┐      ┌─────────────────────┐
│ Input stream is            │      │ When Job 210        │
│ DBA1:[HIGGINS]TESTALL.COM  │─────>│ can be executed,    │
└────────────────────────────┘      │ a process is        │
                                     │ created to execute  │
                                     │ the job. When       │
┌────────────────────────────┐      │ the job is completed,│
│ Output stream is           │<─────│ the process is      │
│ DBA1:[HIGGINS]TESTALL.LOG  │      │ deleted             │
│ a temporary file that is   │      └─────────────────────┘
│ deleted                    │
│ after it is printed.       │
└────────────────────────────┘
```

                                                            ZK-831-82

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:

    SET QUEUE/ENTRY=nnn/PRIORITY=new-priority queue-name

    SET PROCESS/PRIORITY=new-priority JOBnnn

The privilege ALTPRI is required to raise the priority for a job.

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:

    STOP/ENTRY=nnn queue-name

Entry deletion may require GROUP or WORLD privilege.

To give a batch job a specific processing priority with respect to other processes in the system, use one of the following:

    SUBMIT/PRIORITY=priority file-spec

    JOB username /PRIORITY=priority

OPER privilege is required to enter a job at a higher priority.

To release for processing a batch job that is being held in a queue, use:

    SET QUEUE/ENTRY=nnn/release queue-name

To specify a name for a batch job, overriding the default job name assigned by the system, use one of the following:

    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:

    SUBMIT/CPUTIME=n file-spec

    JOB username /CPUTIME=n

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:

    SUBMIT/WSQUOTA=n file-spec

    JOB username /WSQUOTA=n

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:

```
$ DEFINE SYS$PRINT DUMMY
```

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:

    $ DEFINE SYS$PRINT LPB0:

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.

DBA1:[HIGGINS]MAINCOMP.COM

```
$ FORTRAN ALPHA/LIST
$ SYNCHRONIZE MINCOMP
$ LINK/MAP/FULL ALPHA,BETA
$ EXIT
```

DBA1:[HIGGINS] MINCOMP.COM

```
$ FORTRAN BETA/LIST
$ EXIT
```

```
$ SUBMIT MAINCOMP
  JOB 314 entered on queue SYS$BATCH ❶
$ SUBMIT MINCOMP
  JOB 315 entered on queue SYS$BATCH ❶
```

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.

ZK-832-82

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 five-minute interval, it retries 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.

DIR.COM   Section A.3

This procedure imitates the DCL command DIRECTORY/SIZE=ALL/DATE, displaying the block size (used and allocated) and creation date of the specified files. It illustrates use of the F$PARSE, F$SEARCH, F$FILE_ATTRIBUTES, and F$FAO lexical functions.

SYS.COM   Section A.4

This procedure returns statistics about processes in the current process list. If the current process has GROUP privilege, statistics for all processes in the group are returned. Statistics for all processes on the system are displayed if the current process has WORLD privilege. This procedure illustrates use of the FPID, FEXTRACT, and F$GETJPI lexical functions.

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.

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 .GTS. "23" .OR. ABS_MINUTES .GTS. "59" -
            THEN GOTO BADTIME
      $ !
      $ ! Convert both time values to minutes
      $ ! Note the implicit string to integer conversion being performed
      $ !
  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 == F$STRING(HOURS_TO_WAIT)+":"+F$STRING(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
$ ! Error status is returned but not displayed
⑫ $ BADTIME:
$ WRITE SYS$OUTPUT "Invalid time value: ",P1,", format must be hh:mm"
$ EXIT %x10000000
$ !
$ ! Output message and exit if user enters inquiry
$ !
⑬ $ 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]
```

Notes

❶ The procedure checks whether the parameter was omitted or whether the value entered for a parameter is the question mark character (?). In either case, 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.

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 as strings, or because of implicit string to integer conversion, integers.

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 Assignment statements implicitly convert both the current time string and the entered time string to an integer equal to the number of minutes by multiplying the hours by the intger 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 character string equivalents' 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 with an error status. The high order digit in the error status is set to 1 in order to suppress the output of an error message.

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
10-JUN-1982 10:38:26
$ @CONVERT 12:00 SHOW
   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'
   $ !
   $ ! Check the return status for success
   $ IF .NOT. $STATUS THEN GOTO END
   $ !
   $ WAIT 'WAIT_TIME'                         ! Wait the specified delta tim(
   $ !
❹ $ BELL[0,32]= %X07                          ! ASCII code for terminal bell
   $ !
   $ WRITE SYS$OUTPUT BELL,"Wake up --  Time is ''F$TIME()'"
   $ !
   $ END:
❺ $ IF SAVE_VERIFY THEN SET VERIFY            ! Restore verification, if set
```

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. At this point the return status for the CONVERT procedure is checked for success. If the conversion returns an error, control is routed to the end of the procedure.

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
 10-JUN-1982 10:39:12
$ @WAKEUP
Enter time to wake: 11:30
Wake up -- Time is 10-JUN-1982 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 DIR.COM

```
$ !
$ ! Command procedure implementation of DIRECTORY/SIZE=ALL/DATE
$ ! command
$ !
$ SAVE_VERIFY = F$VERIFY((0)              ! Turn verification off
❶ $ P1 = F$PARSE(P1,"*.*;*")              ! Create directory wild
$                                         ! card spec
$ FIRST_TIME = "TRUE"                     ! Header not printed yet
$ FILE_COUNT = 0                          ! No files found yet
$ TOTAL_ALLOC = 0                         ! No blocks allocated yet
$ TOTAL_USED = 0                          ! No blocks used yet
$
$LOOP:
❷ $        FILESPEC = F$SEARCH(P1)
$ ! Find next file in directory
$        IF FILESPEC .EQS. "" THEN GOTO DONE
$ ! If no more files, then done
$        IF .NOT. FIRST_TIME THEN GOTO SHOW_FILE
$ ! Print header only once
$ !
$ ! Construct and output header line
$ !
$        FIRST_TIME = "FALSE"
❸ $        DIRSPEC = F$PARSE(FILESPEC,,,"DEVICE") -
$              +F$PARSE(FILESPEC,,,"DIRECTORY")
$        WRITE SYS$OUTPUT ""
$        WRITE SYS$OUTPUT "Directory ",DIRSPEC
$        WRITE SYS$OUTPUT ""
$        LASTDIR = DIRSPEC
$
$ !
$ ! Put the file name together, get some of the file attributes, and
$ ! type the information out
$ !
❹ $SHOW_FILE:
$        FILE_COUNT = FILE_COUNT + 1
$        FILENAME = F$PARSE(FILESPEC,,,"NAME") -
$              + F$PARSE(FILESPEC,,,"TYPE") -
$              + F$PARSE(FILESPEC,,,"VERSION")
$        ALLOC = F$FILE_ATTRIBUTES(FILESPEC,"ALQ")
$        USED = F$FILE_ATTRIBUTES(FILESPEC,"EOF")
$        TOTAL_ALLOC = TOTAL_ALLOC + ALLOC
$        TOTAL_USED = TOTAL_USED + USED
$        LINE = F$FAO("!19AS !5UL/!5<!UL!> !17AS",FILENAME,-
$              USED, ALLOC, F$FILE(FILESPEC,"RDT"))
$        WRITE SYS$OUTPUT LINE
$        GOTO LOOP
$
$ !
$ ! Output summary information, reset verification, and exit
$ !
❺ $DONE:
$        WRITE SYS$OUTPUT ""
$        WRITE SYS$OUTPUT "Total of ''file_count' files, ",-
$                    total_used, "/", total_alloc, " blocks."
$        IF SAVE_VERIFY THEN SET VERIFY
$        EXIT
```

Notes

❶ The procedure creates a directory wild card specification using the F$PARSE lexical function. This specification wild cards any blank fields in the user-supplied file specification. If no parameter is specified when the procedure is executed, all files in the current process's default directory are displayed.

❷ The F$SEARCH lexical function searches the directory for the next file. If no more files are found, the procedure branches to DONE. (see note 5).

❸ The F$PARSE lexical function uses the first file specification found by the search to construct a header for the directory display.

❹ In this loop, the procedure uses the F$PARSE lexical function to extract the file name to be diplayed from each file name in the directory. The F$FILE_ATTRIBUTES lexical function then obtains blocks used, blocks allocated, and creation date information about each file. Finally, the file name and file attribute information are used as arguments for the F$FAO lexical function. The F$FAO function formats a single display line for each file in the directory.

❺ When no more files are found by F$SEARCH, the procedure branches to DONE and summary information is displayed showing total number of files the total blocks used, and the total blocks allocated in the directory.

Sample Execution

```
$ @DIP  VERN].COM
  Directory DBA0:[VERN]

  BATCH.COM;1              1/3        16-JUN-1982 11:43
  CALC.COM;3               1/3        16-JUN-1982 11:30
  CONVERT.COM;1            5/6        16-JUN-1982 15:23
                .
                .
                .
  LOGIN.COM;34             2/3        16-JUN-1982 13:17
  PID.COM;7                1/3        16-JUN-1982 09:49
  SCRATCH.COM;6            1/3        16-JUN-1982 11:29

  Total of 15 files, 22/48 blocks.
```

The procedure returns information on all COM files in the directory [VERN].

A.4   SYS.COM

```
$ !
$ ! Displays information about owner, group, or system processes.
$ !
$ SAVE_VERIFY = F$VERIFY(0)          ! Turn off verification
❶ $ CONTEXT = ""                        ! Initialize PID search context
$ !
$ ! Output header line.
$ !
❷ $ WRITE SYS$OUTPUT "   PID    Username    Term    UIC      Process " -
          name    State    Pri Image"
$ !
$ ! Output process information.
$ !
$LOOP:
$ !
$ ! Get next PID.  If null, then done.
$ !
❸ $        PID = F$PID(CONTEXT)
$          IF PID .EQS. "" THEN GOTO DONE
$ !
$ ! Get image file specification and extract the file name.
$ !
❹ $        IMAGNAME = F$GETJPI(PID,"IMAGNAME")
$          IMAGNAME = F$EXTRACT(F$LOCATE("]",IMAGNAME)+1,999,IMAGNAME)
$          IMAGNAME = F$EXTRACT(0,F$LOCATE(".",IMAGNAME),IMAGNAME)
$ !
$ ! Get terminal name.  If none, then describe type of process.
$ !
❺ $        TERMINAL = F$GETJPI(PID,"TERMINAL")
$          IF TERMINAL .EQS. "" THEN -
                TERMINAL = "-"+F$EXTRACT(0,3,F$GETJPI(PID,"MODE"))+"-"
$          IF TERMINAL .EQS. "-INT-" THEN TERMINAL = "-DET-"
❻ $        IF F$GETJPI(PID,"OWNER") .NE. 0 THEN TERMINAL = "-SUB-"
$ !
$ ! Get some more information, put process line together,
$ ! and output it.
$ !
❻ $        LINE = F$FAO("!AS !12AS !5AS !9AS !15AS !5AS !2UL/!UL "-
                "!10AS",PID,F$GETJPI(PID,"USERNAME"),TERMINAL,-
                F$GETJPI(PID,"UIC"),F$GETJPI(PID,"PRCNAM"),-
                F$GETJPI(PID,"STATE"),F$GETJPI(PID,"PRI"),-
                F$GETJPI(PID,"PRIB"),IMAGNAME)
$          WRITE SYS$OUTPUT LINE
$          GOTO LOOP
$ !
$ ! Restore verification and exit.
$ !
$DONE:
$ IF SAVE_VERIFY THEN SET VERIFY
$ EXIT
```

Notes

❶   The symbol CONTEXT that will be used in the F$PID lexical
    function is initialized with a null ("") value.

❷   The procedure writes a header for the display.

❸   The procedure gets the first process identification (pid)
    number.   If the current process lacks GROUP or WORLD
    privilege, the pid of the current process is returned.   If
    the current process has GROUP privilege, the first pid in the

group list is returned. The first pid in the system list is returned if the current process has WORLD privilege. The function continues to return the next pid in sequence until the last pid is returned. At this point, a null string is returned, and the procedure branches to the end.

❹ The procedure uses the F$GETJPI lexical function to get the image file specification for each pid. The F$EXTRACT function extracts the file name from the specification returned by the F$GETJPI function.

❺ The procedure uses the F$GETJPI function to get the terminal name for each pid. The F$EXTRACT function extracts the first three characters of the MODE specification returned by F$GETJPI(PID,"MODE") to determine the type of process. The F$GETJPI function is used again to determine whether the process is a subprocess.

❻ The procedure uses the F$GETJPI lexical function to get the username, user identification code (UIC), process name, process state, process priority, and process base priority for each pid returned. The F$FAO lexical function formats this information into a screen display.

Sample Execution

```
$ @DIR
```

PID	Username	Term	UIC	Process name	State	Pri	Image
00050011	NETNONPRIV	-NET-	[300,300]	MAIL_14411	LEF	9/4	MAIL
00040013	STOVE	RTA6:	[011,205]	STOVE	LEF	9/4	
00140015	MAROT	-DET-	[001,003]	DMFB0ACP	HIB	9/8	F11BACP
00080016	THOMPSON	-DET-	[001,003]	MTA0ACP	HIB	12/8	MTAAACP
00070017	JUHLES	TTF1:	[303,012]	JUHLES	LEF	9/4	
				.			
				.			
				.			
00040018	MARCO	TTA2:	[011,040]	MARCO	HIB	9/4	RTPAD
0018001A	VERN	RTA3:	[011,171]	VERN	LEF	9/4	
0033001B	YISHA	RTA7:	[360,052]	YISHA	CUR	4/4	
0002004A	SYSTEM	-DET-	[001,006]	ERRFMT	HIB	12/7	ERRFMT

This procedure returns information on all processes on the system. The current process has WORLD privilege.

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 'P1 'P2 'P3 'P4 'P5 'P6 'P7 'P8

           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
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
   $       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:   ! Redefine SYS$INPUT
   $       EDIT/SOS '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($STATUS)'"
   $       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 integer 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
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 name of file 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
           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:
   $       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
   $       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:
   $       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
   $       RUN/NODEBUG 'FILE'
   $       GOTO NEXT
   $ DEBUG0:
   $       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
   $       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.

❽   At the label INVALID_COMMAND, the procedure writes an error message and displays the help text that lists the commands that are valid.

❾   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.

❿   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.

⓫ 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 a DEFINE command that defines the input stream as SYS$COMMAND.

⓬ 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.

⓭ 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 3.0


    VAX/VMS FORTRAN Command Interpreter

    Enter name of file 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
Next:   COMPILE
Next:   LINK
Next:   RUN
Next:   FILE
File:   READFILE
Next:   EDIT
```

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 $ DEFINE/USER_MODE SYS$OUTPUT: NL:    ! 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[21,20]:="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;*
$ 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 RETURN, 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 DEFINE command directs the SORT command output to the file NL:. 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 (T, t, Y, or y) 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
Programmer: WATERS
Document file name: CRYSTAL.CAV
Programmer: JENKINS
Document file name: MARIGOLD.DAT
Programmer: MASON
Document file name: SYSTEM.SRC
Programmer: ANDERSON
Document file name: JUNK.J
Programmer: RET
Delete temporary files:y
```

The output file resulting from this execution of the procedure is:

Programmer Files as of 10-JUN-1982 16:18:58.79

Programmer Name	File Name
ANDERSON	JUNK.J
JENKINS	MARIGOLD.DAT
MASON	SYSTEM.SRC
WATERS	CRYSTAL.CAV

A.9 CALC.COM

```
   $ SET NOVERIFY
 ❶ $ ON WARNING THEN GOTO START
   $ START:
 ❷ $ INQUIRE STRING "Calc"
   $ IF STRING .EQS. "" THEN EXIT
   $ IF F$LOCATE("=",STRING) .EQ. F$LENGTH(STRING) THEN GOTO SHOW
   $        'STRING'                             ! Execute assignment statements
   $        STRING=F$EXTRACT(F$LOCATE("=",STRING)+1,999,STRING)
   $ SHOW:
   $ Q = F$INTEGER('STRING')                     ! "Q" = last quantity evaluated
   $ LINE = F$FAO("Decimal = !SL    Hex = !-!XL    Octal = !-!OL",Q)
 ❸ $ WRITE SYS$OUTPUT LINE
 ❹ $ GOTO START
```

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 procedure evaluates the expression and displays the results with the WRITE command.

❹ The GOTO command returns to the label START. The procedure loops to request another expression.

Sample Execution

```
$ @CALC
Calc: 5555*30
Decimal = 166650 Hex = 00028AFA Octal = 00000505372
Calc: 32+3
Decimal = 35 Hex = 00000023 Octal = 00000000043
Calc: TOTAL = %X3A + %X4C
Decimal = 134 Hex = 00000086 Octal = 00000000206
Calc:
$ RET
```

After each prompt from the procedure, the user enters an arithmetic expression. The procedure displays the results in decimal, hexadecimal, and octal. A null line, signaled by RETURN on a line with no data, concludes the CALC session.

INDEX

Ampersand,
  and apostrophe substitution
    operator, 4-4
  as substitution operator,
    4-3
Apostrophe,
  and ampersand substitution
    operator, 4-4
  as substitution operator,
    4-2
Appending records to a file,
  8-5
Argument,
  in lexical function, 5-1
Arithmetic comparison
    operation, 3-8
  operands for, 3-8
Arithmetic operation, 3-9
  operands for, 3-10
  value conversion in, 3-9
ASCII character set, 3-9
Assignment statement,
  equates a symbol to
    character string, 3-10
  equates a symbol to a string
    expression, 3-4
  equates a symbol to an
    integer expression, 3-5
  equates symbol to string or
    integer value, 3-1
  special-purpose, 3-10

Batch execution of command
    procedure, 1-8, 1-10
Batch job,
  abnormal termination, 9-6
  change name of, 9-3
  change priority of, 9-3
  control in queue, 9-2
  control of, 9-1
  delay processing of, 9-3
  deletion from queue, 9-3
  execution, 9-1
  execution synchronization,
    9-6
  how system executes, 9-2
  log file, 9-4
  output, 9-4
  passing parameters to, 3-17
  queue, 9-1
  saving log file, 9-5
  specification of CPU time
    limit, 9-4
  specification of working set

Batch job (Cont.)
    quota, 9-4
  submission through card
    reader, 1-11
  verification, 2-3
Batch job queue,
  control of job in, 9-2
  deletion of job from, 9-3
  monitor status of, 9-1
Binary overlay,
  in symbol name, 3-12
Branch,
  within a command procedure,
    6-7

Card reader,
  to submit batch command
    procedures, 1-11
Character string,
  see String,
Command interpreter,
  action at different severity
    levels, 7-2
  default action by CTRL/Y,
    7-5
  default action with error,
    7-2
  evaluation of lexical
    function, 5-1
  evaluation of string, 4-2
  override default error
    condition action, 7-7
  replacement of undefined
    symbol, 4-9
  steps in symbol
    substitution, 4-4
Command line,
  continuation, 1-6
Command procedure,
  action of ON command within,
    7-3
  capabilities, 1-1
  command line continuation,
    1-6
  comments in, 1-7
  concatenation into a single
    job, 9-4
  control of execution, 6-1
  creation, 1-5 to 1-6
  DCL commands used in, 1-2 to
    1-4
  default file type, 1-5, 1-9
  default output, 2-2
  definition, 1-1

Index-1

INDEX

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

☐ Assembly language programmer
☐ Higher-level language programmer
☐ Occasional programmer (experienced)
☐ User with little programming experience
☐ Student programmer
☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
                                                                     or Country
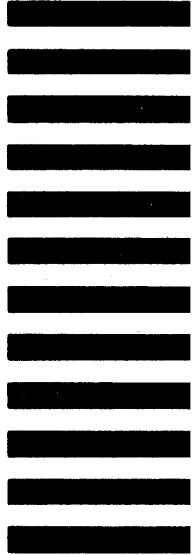
digital

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061