RSTS/E
Guide to Writing Command Procedures

Order No. AA-CF03A-TC

June 1985

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

OPERATING SYSTEM AND VERSION:	RSTS/E	V9.0
SOFTWARE VERSION:	RSTS/E	V9.0

digital equipment corporation, maynard, massachusetts

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

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

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

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

digital™	DIBOL	ReGIS
	FMS-11	RSTS
DEC	LA	RSX
DECmail	MASSBUS	RT
DECmate	PDP	UNIBUS
DECnet	P/OS	VAX
DECtape	Professional	VMS
DECUS	Q-BUS	VT
DECwriter	Rainbow	Work Processor

CONTENTS

FIGURES

TABLES

Preface

Objectives

This guide presents concepts and techniques for developing command procedures using the DIGITAL Command Language (DCL) on RSTS/E. Various examples, including complete command procedures, demonstrate applications of the concepts and techniques that this guide discusses.

Audience

All RSTS/E users can benefit from using command procedures. For example, you can place frequently used command sequences into a command procedure 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.

Although you do not have to be a computer expert to use this guide, you will have an easier time if you are familiar with programming concepts such as loops, logical values, and so forth.

Document Structure

Each chapter in this guide builds on material from earlier chapters. If command procedures are new to you, study this guide chapter by chapter beginning with Chapter 1. If you already have some knowledge of command procedures, you may want to skim the Table of Contents and Index for the specific topics you need.

This manual contains nine chapters and four appendixes:

Chapter 1	Defines command procedures and describes how to develop them.
Chapter 2	Describes how to define and manipulate symbols.
Chapter 3	Describes the use of expressions and operators.
Chapter 4	Shows how to use the DCL functions to obtain information about the status of a process and to manipulate character strings.
Chapter 5	Describes how to control input to and output from command procedures.
Chapter 6	Explains how to manipulate files using command procedures.

Chapter 7	Describes how to control the sequence in which command procedure lines are executed.
Chapter 8	Shows how to set up error handling routines based on the severity of errors encountered during command procedure execution and how to handle CTRL/C interrupts that occur during command procedure execution.
Chapter 9	Describes commands for displaying command procedure output.
Appendix A	Contains sample command procedures that illustrate the techniques described in Chapters 1 through 9.
Appendix B	Lists some of the major differences between the RSTS/E and VAX/VMS command processors.
Appendix C	Lists the numeric values associated with RSTS/E error messages.
Appendix D	Lists the ASCII character codes.

Related Documents

This guide refers you to the following manuals for more detailed information:

- *RSTS/E System User's Guide*
- *RSTS/E System Manager's Guide*.

Conventions

This manual uses the following symbols and conventions:

[ ]	Square brackets show the optional parts of a command in format statements. For example: DIRECTORY [file-spec[,...]] The square brackets in this example indicate that you can include a file specification ([file-spec]), or more than one ([,...]), if you choose. Square brackets also indicate the choice you have in using a command. For example: /[NO]DELETE This means you can type either /DELETE or /NODELETE, depending on the form of the qualifier you select. Do not confuse the square brackets in command formats with the square brackets in Project-Programmer numbers (PPNs), as in [52,20].
<CTRL/x>	The control key, which you use in combination with another key. For example, enter CTRL/U by holding down the CTRL key and pressing the keyboard key labeled "U." RSTS/E displays, or echoes, CTRL/U at your terminal, as ^U.
<RET>	The key labeled RETURN on your terminal. You press the RETURN key to complete lines and commands that the system will process.
color	In examples, black characters are data produced by the computer. Red characters indicate information that you type.

Chapter 1

Introduction

This chapter introduces command procedures and tells you how to develop them.

What are Command Procedures?

Command procedures are files that contain DCL commands. To execute these commands, you run the procedure. Use command procedures to execute sequences of commands you use during interactive terminal sessions or to execute commands you submit for batch processing. As you become more experienced in creating and using command procedures, you will find many other applications for them.

Command procedures can range from simple to complex. A simple command procedure consists of one or more command lines for the DCL command interpreter to execute. For example, the following command procedure deletes all temporary (.TMP) files and then shows a directory listing:

DIRECT.COM

```
$ !  Delete .TMP files and show directory
$ DELETE *.TMP
$ DIRECTORY
```

A more complex command procedure performs program-like functions. It can:

- Contain loops and error checking routines
- Perform arithmetic calculations and input/output (I/O) operations
- Manipulate character string data
- Call or pass parameters to other command procedures

For example, the following command procedure edits all occurrences of files with a .B2S file type in a user's account:

B2S.COM

```
$ !  Edit all .B2S files
$ SET NODATA
$ NEXTFILE = F$SEARCH("SY:*.B2S")
$ LOOP:
$ IF NEXTFILE .EQS.  "" THEN EXIT
$ EDIT/EDT 'NEXTFILE'
$ NEXTFILE = F$SEARCH()
$ GOTO LOOP
```

Creating Command Procedures

To create a command procedure, use a text editor such as EDT or the DCL CREATE command. The following example shows how to create a simple command procedure using EDT:

```
$ EDIT/EDT RUN.COM <RET>
Input file does not exist
[EOB]
*C <RET>
$ !Run three programs <RET>
$ RUN PROG1 <RET>
$ RUN PROG2 <RET>
$ RUN PROG3 <RET>
<CTRL/Z>
*EXIT
RUN    .COM 3 lines
$
```

The next example shows how to create the same procedure using the DCL CREATE command:

```
$ CREATE RUN.COM <RET>
$ !Run three programs <RET>
$ RUN PROG1 <RET>
$ RUN PROG2 <RET>
$ RUN PROG3 <RET>
<CTRL/Z>
$
```

Note that the examples use the file type .COM. When you execute a command procedure, DCL uses this file type as the default.

Formatting Command Procedures

Follow these format conventions and restrictions when you create a command procedure:

- Begin each DCL command line with a dollar sign ($) character in the first space of the line -- Using this character ensures that DCL processes the command when you execute the procedure from another keyboard monitor such as BASIC-PLUS.

- Begin each line containing a DCL label with a $ character in the first space of the line -- Put labels on separate lines to help make loops and conditional coding easier to understand.

- Do NOT begin the following types of lines with the $ character:

  - Data lines

  - Command lines to keyboard monitors other than DCL

  - DCL continuation lines

- Separate command sequences -- Insert lines containing only a $ character before and after a logical sequence of commands. This convention makes it easier to see the structure of the command procedure.

- Avoid abbreviating DCL keywords to less than 4 characters -- DCL conventions ensure that all qualifiers and keywords are unique in the first 4 characters.

- Use comments -- Comments explain the procedure to anyone who must maintain it; DCL ignores them during execution. You begin comments with an exclamation point and place the comment to the right of the exclamation point. You can place a comment on a separate command line or at the end of a command line. If a comment exceeds one line, place an exclamation point at the start of each line.

  Use comments at the beginning of a procedure to describe the procedure and the parameters you pass to it or use them at the beginning of each block of commands to describe that section of the procedure. However, note that excessive use of comments can decrease performance.

Executing Command Procedures

You can execute command procedures in many different ways on RSTS/E systems:

- At login time, RSTS automatically executes the system-wide LOGIN.COM file and any private LOGIN.COM file
- At interactive level, using the at sign (@) command, or from any other keyboard monitor using the $@ combination
- In batch mode, using the SUBMIT command
- From inside another command procedure (nesting)
- With the RUN command from any keyboard monitor, provided the command file is executable and includes the DCL run-time system attribute
- With the CHAIN statement from another program, provided the command file is executable and includes the DCL run-time system attribute
- With a Concise Command Language (CCL) command from any keyboard monitor, provided the command file is executable and includes the DCL run-time system attribute
- At system start-up time or after a system crash

Note that you can execute a command file that is resident on disk only. However, DCL copies a command file from magnetic tape (DOS format only) to a temporary disk file before executing it. For example:

```
$ @MT1:MYFILE.COM
```

DCL temporarily copies the command file from magnetic tape to a temporary disk file, and then executes it.

The following sections describe the methods for executing command procedures.

Executing Command Procedures at Login

When you log in, RSTS/E executes the system-wide command file LOGIN.COM, which is located in directory [0,1] on the public structure. This file contains commands defined by the system manager that are executed for all users. In addition, the system-wide LOGIN.COM file normally contains a command that invokes a private LOGIN.COM file located in your account on the public structure. This

feature lets you tailor the system for your everyday use.

For example, if you enter the same sequence of commands every time you log in, place these commands in a command procedure named LOGIN.COM. A typical LOGIN.COM file might contain the following:

```
$ TYPE $NEWS.TXT                    !Display system messages
$ RUN DB1:[2,214]COOKIE.EXE         !Run a program
$ US-ERS:== "SHOW USERS"            !Abbreviate a command
$ ASSIGN DR3:[2,214] WORK:          !Assign a logical name
$ DIR-ECTORY :== DIRECTORY/DATE     !Redefine DIRECTORY command
```

Note that if a symbol has one or more abbreviations, DCL displays it with an embedded hyphen indicating the minimum abbreviation point.

If your system manager allows the use of private LOGIN.COM files, the system automatically executes these commands every time you log in.

Executing Command Procedures at Interactive Level

The @ command executes a command procedure at interactive level.

Format

```
@file-spec [P1 [P2 [... P8]]]
```

Prompts

```
Command file: file-spec
```

Command Parameters

file-spec

The file specification of the command file to execute. If you do not specify a file type, the system uses .COM.

[P1 [P2 [... P8]]]

Optional parameters (one to eight) to pass to the command procedure. Separate each parameter with one or more spaces or tabs. See Chapter 5 for a complete description of passing parameters.

For example, to execute the command procedure FILST.COM in your account on the system disk, enter this command:

```
$ @FILST
```

Figure 1-1 shows how the command procedure is executed at interactive level.

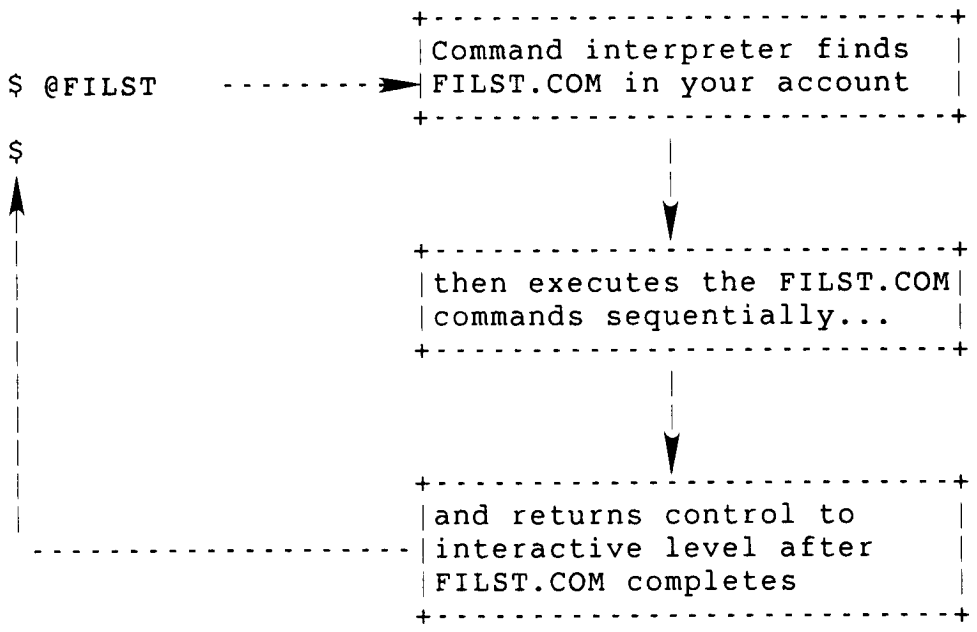

Figure 1-1: Executing a Command Procedure at Interactive Level

When you enter the @ command, the command interpreter executes the file FILST.COM located in your account on the system disk. Each command string in FILST.COM executes sequentially. After reaching the end-of-file (EOF) for FILST.COM or after executing an EXIT or STOP command, the command interpreter returns control to interactive level and displays the $ prompt at your terminal. You can then resume interactive work.

Note

You can also use the @ command from other keyboard monitors. However, you must precede it with the $ character. For example:

```
Ready

$@FILST
```

When you enter the @ command at the DCL prompt, the DCL command interpreter assumes that the the name of a file (with the file type .COM) follows it. If you enter the @ command without specifying a file, DCL prompts you for the file specification.

If a command procedure is not in your account on the system disk or does not have the file type .COM, give the complete file specification. For example:

```
$ @DB2:RESET.FIL
```

This command executes a command procedure located on DB2:. The command procedure file name is RESET.FIL.

If you execute command procedures frequently, you can define a symbol name that you can use in place of the entire command line. For example:

```
$ RESET :== @DB2:RESET.FIL
```

This assignment statement defines the symbol name RESET to be equivalent to the string "@DB2:RESET.FIL." You can then use this symbol as a command name during the current terminal session.

If you want to use a symbol every time you log in, include the symbol definition in your login command file. See Chapter 2 for more information about using symbols.

Executing Command Procedures in Batch Mode

To execute a command procedure in batch mode, use the SUBMIT command followed by the file specification of the procedure. The file type defaults to .COM. When you use this method, the batch processor executes the command procedure while you continue to work at your terminal. You should submit procedures that require lengthy processing time for batch processing.

For example, to execute the command procedure COPY.COM in batch mode, enter the command:

```
$ SUBMIT COPY
```

In this example, the command interpreter finds the file COPY.COM in your account and submits it to the default batch queue for processing. After the batch job is queued, your terminal is free for you to continue interactive work.

See the *RSTS/E System User's Guide* and *RSTS/E System Manager's Guide* for more information on batch processing.

Executing Nested Command Procedures

Executing one command procedure from inside another is called nesting. To execute one command procedure from inside another, use the @ command followed by the name of the nested procedure. This process is similar to using a CALL statement in a high-level language.

When the DCL command interpreter finds a nested command procedure, it reads input from the second procedure until it reaches the end of the file or until the procedure exits. Control then returns to the first command procedure at the line following the @ command.

Note that you can nest a maximum of 13 command procedures on RSTS/E. See Chapter 7 for more information about nesting command procedures.

Executing Command Procedures with the RUN Command

You can use the RUN command from any keyboard monitor to execute command procedures as you would a program on RSTS/E. A command file that you execute with the RUN command must follow these rules:

- The file must be on disk
- You must set the execute bit (64) in the file's protection code
- The file must have DCL as its run-time system

Note that you cannot pass any parameters to command files that you execute with RUN. See Chapter 5 for a complete discussion of parameter passing.

Chaining to Command Procedures from a Program

You can also execute a command procedure using the CHAIN statement or the .RUN directive. Command files you chain to must follow the same rules as command files you execute using the RUN command.

To pass parameters when you chain to a command procedure, you must specify a nonzero parameter word and load core common with one or more parameter values separated by spaces or tabs. See Chapter 5 for more information about passing parameters.

Executing Command Procedures with CCLs

The Concise Command Language (CCL) lets you execute command procedures that are defined as CCLs by your system manager.

When you execute a command procedure with a CCL command, you must follow the same rules as command files you chain to; that is, to pass parameters you must specify a nonzero parameter word and load core common with one or more parameter values separated by spaces or tabs. For complete information about parameter passing, see Chapter 5.

See the *RSTS/E System Managers Guide* for more information about defining or using CCLs.

Command Procedures and System Management

The RSTS/E system manager can create a system-wide LOGIN.COM file containing commands that are executed for all users. If you are a system manager, you also use command files in these ways:

- When you start timesharing, the system automatically loads DCL and executes the file [0,1]SYSINI.COM. SYSINI.COM performs preliminary startup procedures and then executes [0,1]START.COM, passing it the parameter "START" in P1.

- When the system restarts after a system crash or power failure, it automatically loads DCL and executes the file [0,1]SYSINI.COM. SYSINI.COM then executes [0,1]START.COM, passing it the parameter "CRASH" in P1.

See the *RSTS/E System Manager's Guide* for complete information about using command files in system management.

Chapter 2

Using Symbols in Command Procedures

This chapter tells you how to define and use symbols in command procedures. It describes:

- The syntax of symbol names
- How to define symbol values
- How the command interpreter substitutes values for symbols during command processing

Symbols in DCL

A symbol is a name that represents a number, character, or logical value. You define a symbol in an assignment statement. You use symbols in command procedures as constants or variables, and manipulate them in much the same way as you manipulate variables in a programming language. This capability, combined with the ability to control execution flow in command procedures, makes the DCL command language very much like a programming language.

The following example shows how you can define a symbol to represent a string:

```
$ FILE = "SOURCE.CBL"
```

This assignment statement gives the symbol name FILE the string value "SOURCE.CBL". You can then refer to the file SOURCE.CBL symbolically by using the symbol name FILE. For example:

```
$ COBOL 'FILE'
```

The apostrophes around the symbol name FILE tell the command interpreter that the enclosed word is a symbol name. The command interpreter substitutes the string value "SOURCE.CBL" for FILE before executing the COBOL command.

Symbol Names

A symbol name is a string from 1 to 255 characters long that can consist of:

- Uppercase letters A-Z
- Lowercase letters a-z
- Numbers 0-9
- Dollar signs ($)
- Underscores (_)

Note that the command interpreter always converts lowercase letters in symbol names to uppercase.

Also note the following restrictions for symbol names:

- You cannot begin a symbol name with a numeric character, an underscore, or the characters F$ (reserved for DCL function names).
- Symbols that begin with the $ character are reserved symbols. You cannot create or assign new values to reserved symbols. You can, however, use a $ character anywhere else within a symbol name.
- In a symbol name, the underscore (_) character is interpreted in the same way as any other character. This differs from its use with DCL qualifiers, where the underscore is transparent. For example, the symbol names MYFILE and MY_FILE are different symbols.
- You cannot begin a symbol name with a hyphen (-) character or an asterisk (*) character.

When you create symbol names, you can use the hyphen (-) character or the asterisk (*) character to specify an abbreviation point for the symbol. Specifying an abbreviation point lets you reference the symbol name by any one of its abbreviations in any DCL command that allows symbol names. This feature also lets you create command synonyms that act like DCL commands. For example, in DCL you can specify the PRINT command as PR, PRI, PRIN, or PRINT. Likewise, you

can create the symbol:

```
$ US-ERS = "SHOW USERS"
```

You can then refer to the symbol USERS as US, USE, USER, or USERS (in command synonym substitution only).

Note that DCL displays an error message if you:

- Use more than one hyphen or asterisk in a symbol name.
- Create a symbol name with an abbreviation that matches any existing abbreviation of another symbol. For example:

```
$ PR-INT = "PRINT/COPIES=2"

$ PR-OTECT = "SET PROTECTION=60"
```

  In this example, DCL displays an error message when you make the second assignment, because the abbreviations match.

Symbol Values

You can assign values to symbols using any of the following methods:

- Assigning symbol names to expressions, constant values, DCL functions, or to other variable symbol names with assignment statements.
- Passing up to eight parameters to a command procedure when you invoke it, or to a batch job when you submit it to a batch queue. (Chapter 5 discusses parameter passing.)
- Using the INQUIRE command to prompt for a symbol's value. (Chapter 5 describes the INQUIRE command.)
- Using the READ command to read a record from a file and assign it as a string value to a specified symbol. (Chapter 6 describes the READ command.)

Note

RSTS/E restricts the space available for storing symbols. Whenever insufficient space (less than 100 bytes) remains to define a symbol and its value, an error message appears. Therefore, you should delete symbols you no longer need and avoid excessively long symbol names.

Symbol Types

A symbol type can be either string or integer, depending on the value or expression you assign. When you use an expression, DCL evaluates it to determine the value to assign to the symbol. If the expression evaluates to a string, DCL assigns a string value to the symbol. If the expression evaluates to an integer, DCL assigns an integer value to the symbol.

DCL determines the type of expression by the types of values you use in the expression and by the type of operations you use to manipulate these values. Chapter 3 describes the rules that DCL uses to determine an expression's type.

The following sections describe how you assign values to symbols.

Symbol Assignment

Use an assignment statement to define a symbol and assign it a string or integer value. The format of an assignment statement is:

```
symbol-name =[=] expression
```

where:

symbol-name	is a name from 1 to 255 characters long that contains the characters listed in the "Symbol Names" section at the beginning of this chapter.
=	assigns the value you specify and places the symbol name and value in the local symbol table.
==	assigns the value you specify and places the symbol name and value in the global symbol table.
expression	is an integer or string expression. The expression's value is assigned to the symbol name.

If the symbol name does not exist when you enter the assignment statement, then DCL creates the symbol and assigns its value. If the symbol name already exists, then DCL assigns the new value, replacing the previous value. For example:

```
$ COUNT = 0
```

This assignment statement assigns the value 0 to the local symbol COUNT.

Use double equal signs (==) to assign a symbol to the global symbol table. For example:

```
$ NEWCOUNT == 10
```

This assignment statement assigns the value 10 to the global symbol NEWCOUNT.

You must enclose string literals in quotation marks ("). For example:

```
$ TERM == "SET TERMINAL"
```

This assignment statement assigns the string value "SET TERMINAL" to the global symbol TERM. If DCL finds unmatched quotation marks in a command string that you assign to a symbol, it displays an error message.

If you specify an expression that contains other symbols, DCL uses the value of those symbols when evaluating the expression. For example:

```
$ TOTAL = NEWCOUNT + 10

$ SHOW SYMBOL TOTAL
TOTAL = 20
```

In this example, you use an arithmetic assignment statement to assign a value to the symbol TOTAL. Note that DCL automatically substitutes the value 10 for the symbol NEWCOUNT before evaluating the expression. See the section "Symbol Substitution" for further discussion.

String Assignment

To assign a string to a symbol, you can use a special string assignment statement. The format of a string assignment statement is:

```
symbol-name :=[=] string
```

where:

symbol-name	is a name from 1 to 255 characters long that contains the characters listed in the "Symbol Names" section at the beginning of this chapter.
:=	assigns the string value you specify and places the symbol name and value in the local symbol table.

:==	assigns the string value you specify and places the symbol name and value in the global symbol table.
string	specifies the string value to assign to the symbol. The string can be from 0 to 255 characters long.

If the symbol name does not exist when you enter the assignment statement, then DCL creates the symbol name and assigns its string value. If the symbol name already exists, then DCL assigns the new string value, replacing the previous value.

When you use the special string assignment statement, you do not have to enclose the string in quotation marks. For example:

```
$ TEMP_STRING :== error   number   11
```

This assignment statement assigns the string "ERROR NUMBER 11" to the global symbol TEMP_STRING.

Note that DCL performs the following actions when you use the special string assignment statement to assign a string not enclosed in quotation marks:

- Converts lowercase characters to uppercase
- Removes leading and trailing spaces or tabs
- Reduces multiple spaces or tabs between characters to a single space

When you enclose the string in quotation marks, DCL does not perform case conversion and keeps all spaces and tabs. For example:

```
$ TEMP_STRING :== "error   number   11"
```

This assignment statement assigns the string "error   number   11" to the global symbol TEMP_STRING.

Symbol Tables

The DCL command interpreter stores symbol names and their associated values in two types of symbol tables:

- A local symbol table, which contains symbols you can use at the current command level
- A global symbol table, which contains symbols you can use at all command levels

DCL has a separate local symbol table for each command level except the interactive level. A command level is the DCL environment from which you issue commands. When you nest a command procedure, you increase the command level by one.

For example, when you log in and type commands at your terminal, you are issuing commands from command level 0 (the interactive level). If you execute a command procedure, the commands in the procedure are executed at command level 1. When the procedure terminates and the DCL prompt reappears on your screen, you are back at command level 0.

To create a local symbol, use a single equal sign (=) in the assignment statement. For example:

```
$ TEMP = 1
```

When a nested command procedure ends, DCL deletes the local symbol table current to that command level and makes current the next higher level local symbol table. Because each local symbol table is unique, the same symbol can exist in more than one local symbol table. This feature lets each command file use symbols that are separately defined, thus avoiding conflicts with the use of those symbols elsewhere. For example, you can define the local symbol FILE at several command levels, to have different values at each command level. When the symbol value is needed, DCL first searches the local symbol table at the current command level: if no symbol is found, it then searches the global symbol table.

Note that no local symbol table exists at the interactive command level. DCL places any symbol that you define at the interactive level in the global symbol table, whether you use a single or double equal sign (==) in the assignment statement.

DCL maintains only one global symbol table, which is recognized at every command level including the interactive level. To create a global symbol, use a double equal sign in the assignment statement. For example:

```
$ VAL == 45
```

The following sections describe how DCL performs symbol substitution.

Symbol Substitution

DCL performs symbol substitution by replacing symbol names in the command string with their current values. To use symbols in commands and command procedures, you must understand the following mechanics of symbol substitution:

- Command synonym substitution
- Apostrophe substitution operators
- Automatic evaluation
- Undefined symbols
- Verification of symbol substitution

Command Synonyms

You frequently use global symbols to define command synonyms. Normally, you place command synonyms in your LOGIN.COM file to make the synonyms available each time you log in. For example:

```
$ KB == "SET TERMINAL/DEVICE="
```

This assignment defines KB as a new DCL command. You can now type:

```
$ KB VT125
```

DCL executes the command as if you typed:

```
$ SET TERMINAL/DEVICE= VT125
```

Note that the DCL command interpreter processes a command string by examining the first command keyword to determine if it is a defined symbol. If it is a defined symbol, DCL automatically substitutes the value of the symbol for the symbol name before executing the command string.

You can also use command synonyms to redefine existing DCL commands, which lets you change a default qualifier or parameter. For example:

```
$ PRINT == "PRINT/COPIES=2"
```

When you later issue the PRINT command, DCL automatically prints two copies of the file you specify.

You can override automatic command substitution by placing an underscore (_) before the first character in the command keyword.

Then, even if the keyword is a defined symbol, no automatic substitution occurs. For example:

```
$ _PRINT MY.FIL
```

This command prints one copy of the file MY.FIL, because the preceding underscore tells DCL not to perform automatic substitution.

Note that you can use symbol name abbreviations in command synonym substitution only.

Apostrophes in Symbol Substitution

If you use a symbol name in place of a command parameter or qualifier, you must enclose it in apostrophes ('...'). When DCL finds a symbol name enclosed in apostrophes, it replaces 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 tell DCL that FILENAME is a symbol name and not a literal string.

If you want to include a symbol name within a literal string, place two apostrophes before the symbol name and one apostrophe after it. For example:

```
$ TEXT = "File ''FILENAME' deleted"
```

If the current value of the symbol FILENAME is MYFILE.DAT, then the symbol TEXT is given the string value:

```
File MYFILE.DAT deleted
```

Note that when you use apostrophes to request symbol substitution, DCL performs iterative substitution from left to right in the command string. This means that for each symbol name found in the command line, the string resulting from the substitution is scanned again from the beginning 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. For example:

```
$ FILE := "'A'"
$ A := FILE1.MEM
$ TYPE 'FILE'
```

DCL processes this example as follows:

1. DCL assigns the symbol name FILE to the string value 'A'. The quotation marks prevent DCL from substituting a value for 'A' (which you have not yet defined).

2. DCL assigns the string value FILE1.MEM to the symbol name A.

3. DCL substitutes the current value of 'FILE', when it scans the TYPE command string, resulting in:

   ```
   TYPE 'A'
   ```

4. Because the current value contains apostrophes, DCL scans the line again and substitutes the value of 'A'. Finally, DCL executes the command string:

   ```
   TYPE FILE1.MEM
   ```

When scanning a command string for apostrophe substitution, DCL performs a maximum of 100 iterations. If the command string still contains apostrophes after 100 iterations, DCL displays an error message and does not process the command.

DCL performs apostrophe substitution both before and after the command substitution occurs. For example:

```
$ EXEC := "'COMMAND'"
$ COMMAND := "DIRECTORY [1,2]"
$ QUAL := "/DATE"
$ EXEC 'QUAL'
```

In this example, when DCL processes the command string EXEC 'QUAL,' it first performs apostrophe substitution on 'QUAL,' resulting in EXEC /DATE. Then command substitution occurs, resulting in 'COMMAND' /DATE. Finally, DCL performs apostrophe substitution again, resulting in DIRECTORY [1,2] /DATE.

Automatic Evaluation

When DCL evaluates an expression, it assumes that a keyword beginning with an alphabetic character is a symbol name. In this case, evaluation is automatic and apostrophes are not needed. In fact, if you use apostrophes, the results may be quite different because iterative substitution occurs.

For example, when you use an arithmetic assignment statement, DCL automatically evaluates the expression on the right hand side of the statement:

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

Note that you do not need apostrophes to request substitution for the symbol COUNT because DCL automatically substitutes values for symbols as it executes arithmetic assignments.

Similarly, no apostrophes are necessary in the following IF command:

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

In this example, the IF command assumes that both A and B are symbol names and uses their current values to test their equality.

Undefined Symbols

If a symbol is not defined when you use it in a command string, then DCL displays an error message and sets the reserved symbols $SEVERITY and $STATUS to the exit status ERROR. In addition, DCL does not process the command that contains the undefined symbol. See Chapter 8 for more information about $SEVERITY and $STATUS.

Verifying Symbol Substitution

The SET VERIFY and SET NOVERIFY commands control whether DCL displays lines in a command procedure as it executes them. Use the SET VERIFY command when you want to determine the cause of errors that occur within a command procedure. When verification is in effect, DCL displays each command line exactly as it appears in the command file, before any substitution.

Format	
SET [NO]VERIFY	
Command Qualifiers	**Defaults**
/[NO]DEBUG	/NODEBUG

Command Qualifiers

/[NO]DEBUG

Specifies whether DCL enables command debugging. You can use this qualifier only with SET VERIFY. When enabled, DCL displays each command line twice: once as it appears in the command file, and again as it appears after apostrophe and command substitution. The default is /NODEBUG.

SET NOVERIFY is the default setting. When you use this setting, DCL does not display command lines read from a command file. However, DCL always displays any error messages that occur.

When SET VERIFY is in effect, it is global to all command procedures and remains in effect until you change it with SET NOVERIFY. Note that you can also use the DCL function F$VERIFY to change the current verification setting. See Chapter 4 for more information about F$VERIFY.

Displaying Symbols

The SHOW SYMBOL command lets you display the value of a specified symbol or of all symbols in a local symbol table, the global symbol table, or both.

Format	
SHOW SYMBOL [symbol-name]	
Command Qualifiers	**Defaults**
/ALL	/ALL
/GLOBAL	
/LOCAL	

Command Parameters

symbol-name

The name of the symbol to display. DCL returns an error if the symbol is not found. If you do not specify a symbol name, /ALL is assumed. If you specify both a symbol name and the /ALL qualifier, the symbol name takes precedence.

Command Qualifiers

/ALL

Tells DCL to display all symbols in the specified symbol table. It is the default.

/GLOBAL

Tells DCL to display only global symbols.

/LOCAL

Tells DCL to display only local symbols.

Within a command file, you can use the /LOCAL or /GLOBAL qualifiers with a symbol-name to restrict searching to either the local or global symbol table. If you do not specify the /LOCAL or /GLOBAL qualifier, DCL searches both tables. If DCL finds a symbol name in both tables, it displays the local symbol definition before the global symbol definition. Global symbols are always displayed with a double equal sign (==), while local symbols are displayed with a single equal sign (=).

When you are at the interactive command level, you can only display global symbols because no local symbol table exists at this level. DCL displays an error message if you use the /LOCAL qualifier at this level.

If you want to display a symbol that has an abbreviation point, do not specify the hyphen character along with the symbol name. For example:

```
$ SHOW SYMBOL USERS
US-ERS = "SHOW USERS"
```

Note that if a symbol has one or more abbreviations, DCL displays it with an embedded hyphen indicating the minimum abbreviation point.

Other examples of the SHOW SYMBOL command follow:

```
$ SHOW SYMBOL/LOCAL
```

This command displays all symbols in the local symbol table.

```
$ SHOW SYMBOL COUNT
COUNT = 4
```

This command displays the local symbol COUNT. Because the value 4 is not in quotation marks, the symbol has a numeric value.

```
$ SHOW SYMBOL PRINT
PRINT == "PRINT/NOFEED"
```

This command displays the global symbol PRINT. Because the value "PRINT/NOFEED" is in quotation marks, the symbol has a string value.

Deleting Symbols

Use the DELETE/SYMBOL command to delete 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.

Format	
DELETE/SYMBOL [symbol-name]	
Command Qualifiers	**Defaults**
/ALL	
/GLOBAL	
/LOCAL	

Command Parameters

symbol-name

The name of the symbol to delete. DCL returns an error if the symbol is not found. If you specify both a symbol name and the /ALL qualifier, the symbol name takes precedence.

Command Qualifiers

/ALL

Tells DCL to delete all symbols in the specified symbol table.

/GLOBAL

Tells DCL to delete the global symbol you specify. By default, DCL searches the global symbol table when you attempt to delete a symbol at the interactive level.

/LOCAL

Tells DCL to delete the local symbol you specify. By default, DCL searches the local symbol table when you attempt to delete a symbol at command levels other than the interactive level.

Note that DCL automatically deletes the symbols in a local symbol table whenever the command procedure at that level exits. In addition, DCL deletes symbols in the global symbol whenever you log out.

Remember that because RSTS/E restricts the space available for storing local and global symbols, you should always delete symbols that you no longer need.

Examples of the DELETE/SYMBOL command follow:

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

This command deletes all user-defined global symbols.

```
$ DELETE/SYMBOL/LOCAL COUNT
```

This command deletes the local symbol COUNT.

Chapter 3

Expressions and Operators

This chapter describes the types of values and operators you can use in expressions. Chapter 2 described how you can equate a symbol to either a string or an integer expression. The value that DCL assigns to the symbol is the result of the expression. For example:

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

This command assigns a symbol (C) to an expression containing an integer (4), a DCL function (F$INTEGER("6")), and a symbol (A). The value that DCL assigns to the symbol C is the result of the operations in the expression. In this example, the expression evaluates to an integer value.

DCL also lets you use expressions:

- As DCL function arguments (see Chapter 4)
- With OPEN, CLOSE, READ, and WRITE commands (see Chapter 6)
- With IF commands (see Chapter 7)

Operands in Expressions

Expressions can contain strings, integers, DCL functions, symbols, or combinations of these values. Expressions consist of operands and operators. Operands are the values on which to perform an operation. Operators specify the operation for DCL to perform in evaluating the expression.

Strings

A string can contain any printable characters. When you use strings in expressions, you must enclose them in quotation marks. To include quotation marks in a string, type two consecutive sets of quotation marks. For example:

```
$ PROMPT = "Type ""YES"" or ""NO"""
$ SHOW SYMBOL PROMPT
PROMPT == "Type "YES" or "NO""
```

When you make the symbol assignment, you must use two sets of quotation marks around the words YES and NO. Also, you must enclose the entire string in quotes because the string is used in an expression. Note that DCL preserves uppercase and lowercase, spaces, and tabs when you make the symbol assignment in this way. In addition, DCL places the symbol in the global symbol table because you defined it at the interactive command level.

You can continue a string over two lines by using a plus sign (+) for string concatenation, and a hyphen (-), for continuation. For example:

```
$ MONTHLY = "MONTHLY REPORT --" +-
Continue:  " JUNE 1985"

$ SHOW SYMBOL MONTHLY
MONTHLY == "MONTHLY REPORT -- JUNE 1985"
```

You can also concatenate several symbols to create a long string. For example:

```
$ FIRST = "If at first you don't succeed,"

$ SECOND = " try, try again."

$ THIRD = FIRST + SECOND

$ SHOW SYMBOL THIRD
THIRD == "If at first you don't succeed, try, try again."
```

Note that both operands must be strings for string concatenation to occur. If either operand to the plus sign (+) operator is an integer, then DCL performs addition, not concatenation.

Integers

The DCL command interpreter treats all numeric values as decimal integers. The following examples show how to use integers in expressions:

```
$ COUNT = 4 + 1

$ DATA = COUNT + 1

$ SHOW SYMBOL COUNT
COUNT == 5

$ SHOW SYMBOL DATA
DATA == 6
```

In this example, you define two symbols using integer expressions and use the SHOW SYMBOL command to display the value of the symbols.

Note that you do not use quotation marks with integers.

DCL Functions

DCL functions let you obtain information about a requested item. You invoke a DCL function by typing its name, which always begins with the characters F$, and its argument list:

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

You must enclose the argument list in parentheses. Use DCL functions in expressions in the same way you use strings, integers, and symbols. The following example uses the F$LENGTH function in an expression. (The F$LENGTH function returns an integer that specifies the length of a string.) DCL assigns the returned value to the symbol LEN. For example:

```
$ LEN = F$LENGTH ("Roses are red and violets are blue.")

$ SHOW SYMBOL LEN
LEN == 35
```

Note that you do not enclose DCL functions in quotation marks. In addition, you must specify the arguments for DCL functions as expressions. Therefore, if an argument contains a string, you must enclose the string in quotation marks. If an argument contains an integer, a symbol, or another DCL function, do not enclose these values in quotation marks. In the previous example, the F$LENGTH function is not enclosed in quotation marks; however, the argument (a string) is.

The value and data type returned by a DCL function depend on the function. See Chapter 4 for complete descriptions of each DCL function.

Symbols

When you use a symbol in an expression, DCL automatically substitutes the symbol's value. For example:

```
$ COUNT = 3

$ NEWCOUNT = COUNT + 1

$ SHOW SYMBOL NEWCOUNT
NEWCOUNT == 4
```

See Chapter 2 for more information about symbol substitution.

The following sections describe the conversion of value types in expressions; that is, how strings are converted to integers, and how integers are converted to strings.

Value Type Conversion in Expressions

Expressions can contain both integer and string values as operands, which are the values on which to perform an operation. DCL automatically converts such expressions to either a string or integer type, depending on the types of values used in the expression and on the types of operations used to manipulate those values.

You can use the F$TYPE function to determine an expression's type. You can also perform value conversion explicitly using the F$INTEGER and F$STRING DCL functions. See Chapter 4 for more information on these functions.

Table 3-1 lists the rules that DCL uses to determine an expression's type.

Table 3-1: Rules for Determining Expression Types

Expression	Type
Integer value	Integer
String value	String
Integer DCL function	Integer
String DCL function	String
Integer symbol	Integer
String symbol	String
+, -, or .NOT. Any value	Integer
Any value .AND. or .OR. Any value	Integer
Any value +, -, *, or / Any value	Integer
String + String	String
Any value (string comparison) Any value	Integer
Any value (integer comparison) Any value	Integer

Note that DCL automatically converts the expression to either a string or integer type before performing further operations.

Converting Strings to Integers

DCL converts numeric strings, which consist of the numbers 0 through 9, optionally preceded by one or more plus (+) or minus (-) characters, to their signed 32-bit values.

DCL converts alphabetic strings according to these rules:

- Strings that begin with the letter T, t, or Y, y, are converted to the value 1 (true).
- All other strings are converted to the value 0 (false).

For example:

String	---➤	Integer
"4789"	---➤	4789
"-1"	---➤	-1
"yes"	---➤	1 (true)
"FILE"	---➤	0 (false)

Converting Integers to Strings

DCL converts integers to strings by generating a string consisting of the decimal numbers of the integer. If the integer is negative, then the minus sign (-) character precedes the numeric characters. Note that DCL suppresses leading zeros. For example:

Integer	---➤	String
1000	---➤	"1000"
-47	---➤	"-47"
0	---➤	"0"
007	---➤	"7"

Operators in Expressions

Use the operators shown in Table 3-2 to form an expression and define the order of evaluation priority. Logical and comparison operators must be preceded by a period (.) with no intervening blanks. You must terminate an operator with a period. You can type any number of blanks or tabs between operators and operands. For example, the following expressions are equivalent:

```
X.EQS.Y
X .EQS.  Y
```

When you specify more than one operation in an expression, DCL performs the operations in the order of priority listed in Table 3-2, where 1 is the highest priority (performed first) and 6 is the lowest (performed last). (For example, multiplication is performed before addition.) You can use parentheses to override the order in which operators are evaluated, because expressions within parentheses are evaluated first.

Table 3-2 lists the valid operators in expressions. Operators of the same priority are performed from left to right, as they appear in the command.

Table 3-2: Summary of Operators in Expressions

Type	Operator	Priority	Operation
Arithmetic Operators	+	1	Arithmetic unary plus
	-	1	Arithmetic unary minus
	*	2	Arithmetic product
	/	2	Arithmetic division
	+	3	Arithmetic sum
	-	3	Arithmetic difference
String Operator	+	3	String concatenation
Logical Operators	.NOT.	1	Logical Complement
	.AND.	5	Logical AND
	.OR.	6	Logical OR
Arithmetic Comparison Operators	.EQ.	4	Arithmetic equal to
	.GE.	4	Arithmetic greater than or equal to
	.GT.	4	Arithmetic greater than
	.LE.	4	Arithmetic less than or equal to
	.LT.	4	Arithmetic less than
	.NE.	4	Arithmetic not equal to
String Comparison Operators	.EQS.	4	String equal to
	.GES.	4	String greater than or equal to
	.GTS.	4	String greater than
	.LES.	4	String less than or equal to
	.LTS.	4	String less than
	.NES.	4	String not equal to

Arithmetic Operations

Use arithmetic operations to perform calculations on integer expressions. The result of an arithmetic operation is an integer. Operands in arithmetic operations are integer expressions. If you specify a string value as an operand, DCL converts it to an integer value before performing the operation.

Note that DCL uses the plus sign (+) character for both the arithmetic sum operator and the string concatenation operator. DCL performs addition whenever one or both of the operands is an integer. However, if both operands are strings, then DCL performs string concatenation.

Examples of arithmetic operations follow:

Expression	Value of Symbol
A = 5 + 10 / 2	A = 10
B = 5 * 3 - 4 * 6 / 2	B = 3
C = 5 * (6 - 4) - 8 / (2-1)	C = 2

Logical Operations

Use logical operators to perform logical functions on integer values. The result of a logical operation is an integer.

Operands for logical operations are integer expressions. If you specify a string value as an operand, DCL converts it to an integer value before performing the operation.

Examples of logical operations follow:

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

Arithmetic Comparisons

Use arithmetic comparison operators to compare integer values. If the result of an arithmetic comparison is true, the expression has the value 1. If the result of the comparison is false, the expression has the value 0.

If you specify a string value as an operand, DCL converts it to an integer before performing the comparison. If a string begins with an uppercase or lowercase T or Y, DCL converts it to the integer 1. If the string begins with any other letter, DCL converts it to the integer 0.

Examples of arithmetic comparisons follow:

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)

String Comparisons

Use string comparison operators to compare strings. DCL compares character strings based on the binary values of the characters in the string. If the result of the string comparison is true, the expression has the value 1. If the result of the comparison is false, the expression has the value 0.

If you specify an integer value as an operand in a string comparison, DCL converts the integer to a string before performing the comparison. In addition, if one string is longer than the other, DCL pads the shorter string with null characters to match the length of the longer string before making the comparison.

Examples of string comparisons follow:

Expression	Value of Expression
"ABCD" .LTS. "EFG"	1 (true)
"YES" .GTS. "YESS"	0 (false)
"FALSE" .EQS. 0	0 (false)

String Concatenation

Use a plus sign character (+) between strings to link or concatenate strings. The result of this operation is a string value. When you specify an integer value as an argument to the plus sign (+) operator, DCL performs addition, not concatenation. The following example concatenates two strings to form a single string:

```
$ A = "MYFILE" + ".MEM"

$ SHOW SYMBOL A
A == "MYFILE.MEM"
```

Chapter 4

DCL Functions in Command Procedures

This chapter describes how to use DCL functions to manipulate strings or integers, or return information about a file, a job, or the system. You can use DCL functions in the same places you use symbols, expressions, or literal values. In command procedures, you can use DCL functions to perform operations such as:

- Translate logical names
- Manipulate strings
- Determine the current processing mode of the procedure

Format of DCL Functions

The general format of a DCL function is:

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

where:

F$	indicates a DCL function.
function-name	is the function to be evaluated. You can truncate function names to any unique abbreviation.
( )	enclose function arguments. Functions that have no arguments do not need parentheses.
arg,...	specify arguments for the function, if any.

Function arguments can consist of symbols, integer expressions, string expressions, or other DCL functions.

DCL performs automatic symbol substitution on all function arguments. Therefore, when you use a symbol name as a DCL function argument, do not enclose the symbol name within apostrophes.

Note that in some cases, one or more of a function's arguments are optional. If an argument is optional, you can exclude it from the argument list. You can also use commas to indicate which argument you are excluding from the list. For example, the F$PARSE function has three arguments defined, all of which are optional. You can specify the first argument, as follows:

```
F$PARSE(ARG1)
```

In this example, you do not specify the second and third arguments. You could also use commas to indicate that you are excluding the second and third arguments:

```
F$PARSE(ARG1,,)
```

If you want to specify the first and second arguments, but omit the third, you can use either:

```
F$PARSE(ARG1,ARG2)
or
F$PARSE(ARG1,ARG2,)
```

If you want to specify the first and third arguments, but omit the second, you must use commas to separate the arguments. For example:

```
F$PARSE(ARG1,,ARG3)
```

Note that, in some cases, specifying an argument as a null string or as the integer zero is not the same as excluding the argument from an argument list.

DCL Functions

Table 4-1 summarizes the DCL functions. The rest of this chapter describes DCL functions in greater detail and gives examples of their use.

Table 4-1: Summary of DCL Functions

Function	Description
F$ACCESS	Returns the current job's access mode
F$ASCII	Converts the first character in a string to its ASCII value
F$CHR	Converts an integer to its ASCII value
F$CVTIME	Converts a date/time string to a string suitable for comparisons
F$EDIT	Edits a string
F$INSTR	Returns the location of a substring
F$INTEGER	Converts a string to an integer
F$JOB	Returns the current job number
F$LEFT	Extracts a substring from a string, beginning at position 1, and ending at the position you specify
F$LENGTH	Returns the length of a string
F$MESSAGE	Returns the message text associated with a RSTS/E error number
F$MID	Extracts a substring from a string beginning at the position you specify
F$NODE	Returns the current node name

Table 4-1: Summary of DCL Functions (Cont.)

Function	Description
F$PARSE	Returns a complete RSTS/E file specification or a specified field within the file specification
F$PRIVILEGE	Returns the status of a job's privileges
F$RIGHT	Extracts a substring from a string, beginning at the position you specify, and ending at the rightmost position of the string
F$SEARCH	Searches a disk directory for a file and returns the complete RSTS/E file specification of the next occurrence of the file
F$STRING	Converts an integer to a string
F$TERMINAL	Returns the KB number for the current RSTS/E job
F$TIME	Returns the current time and date
F$TYPE	Returns a string indicating the type of a symbol or expression
F$USER	Returns the project-programmer number for the current RSTS/E job
F$VERIFY	Enables or disables the verification setting

The following sections describe the DCL functions in alphabetical order.

F$ACCESS

The F$ACCESS function returns a string that shows the current job's execution mode.

The format of the F$ACCESS function is:

F$ACCESS()

The F$ACCESS function returns one of the following keyword strings:

Keyword	Meaning
BATCH	The job is running on a pseudo keyboard, under the control of a batch processor.
DIALUP	The job is running over a dial-up line.
LOCAL	The job is running at a local terminal.
NETWORK	The job was created over the network, and is running on a pseudo keyboard under control of the job on the remote system.
SERVER	The job is running detached under the control of a network controller.

Use the F$ACCESS function in command procedures that must act differently depending on the current job's access mode.

The following example uses the F$ACCESS function:

```
$ !Exit unless in BATCH mode
$ IF F$ACCESS .NES.  "BATCH" THEN EXIT
      .
      .
      .
```

In this example, the IF command compares the character string returned by F$ACCESS with the character string BATCH. If the strings are equal, processing continues. When the strings are not equal, the procedure exits.

F$ASCII

The F$ASCII function converts the first character in a string to its ASCII value. This function is similar to the BASIC-PLUS ASCII function.

The format of the F$ASCII function is:

F$ASCII(string)

where string is a string expression.

The F$ASCII function always returns an integer in the range 0 to 255. If you pass a null string as an argument to this function, it returns the value 0. If you pass an integer as an argument to this function, DCL converts the integer to a string and returns the first character of the string.

The following example uses the F$ASCII function:

```
$ INQUIRE ANSWER "Type Y to continue"
$ CHAR = F$ASCII(ANSWER)
$ !Check for 'Y' or 'y' as the first character
$ IF (CHAR .EQ.  89) .OR.  (CHAR .EQ.  121) THEN GOTO CONTINUE
$ EXIT
$ CONTINUE:
        .
        .
        .
```

This command procedure uses an INQUIRE command to request user input, which is assigned to the symbol ANSWER. The next assignment statement uses the F$ASCII function to convert the first character of the input string to its ASCII value and assigns that value to the symbol CHAR. The IF command then tests the first character of the string and branches to the label CONTINUE if the first character is Y or y. If the string begins with any other character, the procedure exits.

F$CHR

The F$CHR function converts an integer to its ASCII value. This function is similar to the BASIC-PLUS CHR$ function. See Appendix D for a table of the ASCII character codes and their decimal and octal values.

The format of the F$CHR function is:

F$CHR(integer)

where integer is an integer value that corresponds to the character to be returned.

Use this function to assign the ASCII value of a nonprinting character to a symbol. DCL treats all arguments as eight-bit unsigned integers in the range 0 to 255. If you specify an integer outside this range, the low byte is returned as the ASCII character. For example, F$CHR(257) is equivalent to F$CHR(1), and F$CHR(-1) is equivalent to F$CHR(255).

The following example uses the F$CHR function:

```
$ !Define the standard report heading
$ FORM_FEED = F$CHR(12)
$ HEADING = "INTEROFFICE MEMORANDUM"
$ NEW_LINE = F$CHR(13) + F$CHR(10)
$ INDENT = F$CHR(9)
        .
        .
        .
$ !Write out the report heading
$ WRITE/NODELIMITER 1 FORM_FEED , HEADING , NEW_LINE , INDENT
        .
        .
        .
```

This example:

- Assigns the ASCII character 12 to the symbol FORM_FEED indicating a form feed.
- Assigns the string "INTEROFFICE MEMORANDUM" to the symbol HEADING.
- Assigns the ASCII characters 13 and 10 to the symbol NEW_LINE indicating a carriage return and line feed.

- Assigns the ASCII character 9 to the symbol INDENT indicating a horizontal tab.

- Uses the WRITE command along with the symbols to format a report heading.

  Note that because you specified /NODELIMITER, the next write to channel 1 will be indented to the first tab stop.

F$CVTIME

The F$CVTIME function converts a standard DCL date/time string or the date/time string returned by the F$TIME function to a date/time string of the form:

yy.mm.dd hh:mm

You can use the resultant string to compare two dates.

The format of the F$CVTIME function is:

F$CVTIME([date/time])

where:

date/time	is the date and time to be converted. If you do not supply a date and time, then the current date/time is returned. DCL displays an error message if you specify an illegal date/time argument.

The date/time string can be any valid DCL date/time string, including relative dates/times. This function also accepts a string consisting of a date field, one or more spaces or tabs, and a time field, which lets you convert a string returned by the F$TIME function.

The function always returns a date/time string that is 14 characters long, with a single space between the date and time fields. In addition, the time field is always returned in 24-hour format, beginning at position 10 in the string.

The following example converts a system date/time string to a comparison string:

```
$ TIME = F$TIME
$ SHOW SYMBOL TIME
TIME = "01-May-85 03:37 PM"

$ TIME = F$CVTIME(TIME)
$ SHOW SYMBOL TIME
TIME = "85.05.01 15:37"

$ TIME = F$CVTIME("+6DAYS")
$ SHOW SYMBOL TIME
TIME = "85.05.07 15:37"
```

In this example, the F$TIME function assigns the current date/time string to the symbol TIME. Then the F$CVTIME function converts the date/time string to a string that you can use for comparisons. The example also shows that you can specify relative dates/times with this function.

F$EDIT

The F$EDIT function edits a string. This function is similar to the BASIC-PLUS CVT$$ function.

The format of the F$EDIT function is:

```
F$EDIT(string,integer)
```

where:

string	is the string to edit.
integer	is a bit-coded value that specifies the editing function (or functions) to perform.

You can use F$EDIT to edit two strings before performing a comparison of those strings. For example, you can convert strings to uppercase, strip off leading or trailing spaces and tabs, and so forth. After you execute this function, DCL returns an edited string. Note that you can supply an integer argument, which is the sum of individual function values. For example, the value 48 performs both edit functions 16 and 32.

The following example uses the F$EDIT function:

```
$ INQUIRE COMMAND "Command"
$ !Strip spaces/tabs and convert to uppercase
$ COMMAND = F$EDIT(COMMAND,2+32)
        .
        .
        .
```

In this example, the INQUIRE command prompts the user for a string to edit. After the user enters the string, the F$EDIT function removes any spaces or tabs and converts the string to uppercase characters.

Table 4-2 lists the values for the F$EDIT function.

Table 4-2: Summary of F$EDIT Functions

Value	Editing Function
2	Discards all spaces and tabs
4	Discards any of the following characters: o Null (ASCII code 0) o Line feed (ASCII code 10) o Form feed (ASCII code 12) o Carriage return (ASCII code 13) o Escape (ASCII code 27) o Delete (ASCII code 127)
8	Discards leading spaces and tabs
16	Converts multiple spaces and tabs to a single space
32	Converts lowercase characters to uppercase
64	Converts left and right square brackets ([]) to left and right parentheses (())
128	Discards trailing spaces and tabs
256	Disables editing of characters within quotation marks

F$INSTR

The F$INSTR function locates a substring within a string and returns the position of the substring as an integer. This function is similar to the BASIC-PLUS INSTR function.

The format of the F$INSTR function is:

```
F$INSTR(position,string,substring)
```

where:

position	is an integer, expression, or symbol that indicates the position in the string at which to begin searching for the substring. If the position is negative or 0, then the string is searched starting at position 1. If the position argument is greater than the length of the string, the function returns the value 0.
string	is the string to search.
substring	is the substring to locate. If the substring argument is null, and the position argument is not greater than the length of the string, the function returns the specified start position. If the substring is not located, the function returns the value 0.

You can specify the string and substring as either a string expression, or as a symbol equated to a string expression.

The following example uses the F$INSTR function to count the number of spaces in a string:

```
$ SPACES = 0
$ POS = 0
$ INQUIRE STRING "String"
$ LOOP:
$ POS = F$INSTR(POS + 1,STRING," ")
$ IF POS .EQ.  0 THEN GOTO END
$ SPACES = SPACES + 1
$ GOTO LOOP
$ END:
$ WRITE 0 "Number of spaces = ",SPACES
        .
        .
        .
```

This example uses an INQUIRE command to prompt for a string value. The first time DCL executes the loop, POS is equal to 1, and STRING is equal to 0. After DCL locates a space, it assigns a new value to POS, which is the integer position of the space within the string. In

addition, each time DCL locates a space, it increments SPACE by 1. Finally, after DCL locates all spaces, the procedure branches to the label END, and the WRITE command displays a message indicating the number of spaces in the string.

F$INTEGER

The F$INTEGER function converts a string expression to an integer.

The format of the F$INTEGER function is:

F$INTEGER(string)

where string is a string expression or symbol.

Use the F$INTEGER function to set a symbol to an integer value and then use the symbol in an operation that requires an integer value. For example:

```
$ A = "23"
$ B = F$INTEGER("-9" + A)
$ SHOW SYMBOL B
B = -923
```

In this example, the F$INTEGER function returns the integer equivalent of the expression ("-9" + A), which evaluates to the character string "-923". Because the expression contains two character strings, use the plus sign (+) character for concatenation. The F$INTEGER function ignores leading and trailing blank spaces. Note that the symbol A was assigned the numeric character string "23".

F$JOB

The F$JOB function returns the current job number as an integer.

The format of the F$JOB function is:

F$JOB( )

Use the F$JOB function in command procedures that use their own job number as input to a task, or that need to extract job information from a file, such as a SYSTAT listing. For example:

```
$ JOB_NO = F$JOB
$ WRITE 0 "This is job number ''JOB_NO'."
        .
        .
        .
```

This command procedure assigns the current job number to the symbol JOB_NO and then displays the job number on your terminal.

Note

Be careful using the F$JOB function in command procedures that contain programs that are meant to detach during the procedure's execution. RSTS/E creates a new job (containing the same command procedure) when the program detaches. Therefore, the F$JOB function will return the new job number.

F$LEFT

The F$LEFT function extracts a substring from a string, starting at position 1 and ending at the position you specify. This function is similar to the BASIC-PLUS LEFT function.

The format of the F$LEFT function is:

F$LEFT(string,position)

where:

string	is the string from which to extract the substring.
position	is an integer that specifies the ending position of the substring to extract. If the position argument is negative or 0, the function returns a null string. If the position argument is greater than the length of the string, the function returns the entire string.

The F$LEFT function returns a string that consists of all characters from the first or leftmost position in the string through the position you specify. For example:

```
$ CHARS = F$LEFT("ABCDEFGHI",6)
$ SHOW SYMBOL CHARS
CHARS = "ABCDEF"
```

In this example, the F$LEFT function returns the string starting at position 1 and ending at position 6. DCL then assigns the resultant substring, "ABCDEF", to the symbol CHARS.

F$LENGTH

The F$LENGTH function returns the length of the string that you specify. This function is similar to the BASIC-PLUS LEN function.

The format of the F$LENGTH function is:

F$LENGTH(string)

where string is a string expression or a symbol that equates to a string expression.

The following example uses the F$LENGTH function:

```
$ MSG_NO=5
$ MESSAGE = F$MESSAGE(MSG_NO)
$ MSG_LEN = F$LENGTH(MESSAGE)
$ IF MSG_LEN .EQ.  0 THEN GOTO NULMSG
        .
        .
        .
$ NULMSG:
```

In this example, the F$LENGTH function determines if message text was returned by the F$MESSAGE function. If no message was returned, then the procedure branches to the NULMSG label.

F$MESSAGE

The F$MESSAGE function returns message text corresponding to a RSTS/E error number.

The format of the F$MESSAGE function is:

F$MESSAGE(error number)

where error number is an integer expression resulting in a value from 0 to 255. DCL returns a null string if you specify a number that is not within this range.

Use F$MESSAGE in command procedures that display either error message text or the system installation name (error number 0). For example:

```
$ !Get the error message
$ ERROR_TEXT = F$MESSAGE(2)
```

After you make this assignment, the symbol ERROR_TEXT has the string value:

```
?Illegal file name
```

Although each error message in the system error message file has a numeric value, many numeric values do not have corresponding error messages. See Appendix C for a complete list of the numeric values associated with RSTS/E error messages.

F$MID

The F$MID function extracts a substring from a string, starting at the position you specify. This function is similar to the BASIC-PLUS MID function.

The format of the F$MID function is:

```
F$MID(string,position,length)
```

where:

string	is the string from which to extract the substring.
position	is an integer that specifies the starting position of the substring. If the position argument is negative or 0, the starting position is 1. If the position argument is greater than the length of the string, the function returns a null string.
length	is an integer that specifies the length of the substring to extract. If the length argument is negative or 0, the function returns a null string.

The following example uses the F$MID function:

```
$ DEF = F$MID("ABCDEFGHI",4,3)
$ SHOW SYMBOL DEF
DEF = "DEF"
```

In this example, the F$MID function returns a string starting at position 4. The resultant substring, "DEF", is 3 characters long.

F$NODE

The F$NODE function returns the node name of the system on which the job is running.

The format of the F$NODE function is:

F$NODE()

If DECnet/E is installed and enabled on the system, DCL returns the node name followed by two colons (::). If DECnet/E is not installed, or is not enabled, then DCL returns a null string. For example:

```
$ NODE_NAME = F$NODE
$ IF F$LENGTH(NODE_NAME) .GT.  0 THEN -
        WRITE 0 "Your node is ''NODE_NAME'"
```

In this example, if the job is running on an active DECnet/E node, DCL assigns the node name to the symbol NODE_NAME. The F$LENGTH function compares the length of the string to 0 to determine if a node name was returned. If so, the WRITE command displays the node name at your terminal.

F$PARSE

The F$PARSE function returns one of three values:

- A full RSTS/E file specification for the file you specify
- A specified field within the file specification, when you include a field keyword
- A 32-bit number containing information about the file specification

This function is similar to the BASIC-PLUS File Name String Scan system function call. See the *RSTS/E Programming Manual* for a description.

The format of the F$PARSE function is:

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

where:

file-spec is a string expression that specifies the name of the file to be returned. If any part of the file-spec is invalid, DCL returns an error. However, if you include the FLAGS keyword, DCL returns the value -1. If you omit the file-spec or supply a null string, DCL uses the entire default-spec.

default-spec is a second file specification that defines defaults for any missing fields in the file-spec string. If you omit a field in the file-spec string, DCL returns the corresponding field in the default-spec string. If you omit a field in both the file-spec and the default-spec strings, the corresponding field in the returned string is null. If both strings are null, DCL returns a null string. DCL returns an error if any part of the default-spec is invalid.

field is an optional keyword string that returns a specific portion of the file-spec. If you do not specify this argument, F$PARSE returns a complete filespec, based on the other two arguments. The valid field keywords are:

- DEVICE -- Returns the device name. The device name can be either a physical or logical device name. When you specify a physical device name, DCL returns the device name preceded by an underscore (_). When you specify a logical device name, DCL returns the associated physical device

name preceded by an underscore. However, if the logical name is not found, DCL returns the logical name with no leading underscore. DCL returns all device names followed by a single colon (:). Note that if you specify a file-spec without a device, DCL returns a null string.

- PPN -- Returns the project-programmer number (PPN), in the form [proj,prog]. You can specify the asterisk (*) wildcard character in either field of the PPN. If you specify a file-spec without a PPN, DCL returns a null string.

- NAME -- Returns the file name. If the file name field consists of six or more question marks (??????) or an asterisk (*), DCL returns an asterisk (*). If you specify a file-spec without a file name, DCL returns a null string.

- TYPE -- Returns the file type, preceded by a period (.). If the file type field consists of three or more question marks (.???) or an asterisk (.*), DCL returns an asterisk preceded by a period (.*). If you specify a file-spec without a file type, DCL returns a null string.

- STATUS -- Returns the device status word value as a 32-bit integer, which contains information about the device returned by the F$PARSE function. If you specify a file-spec without a device, DCL returns the value 0. You can test each bit to determine status. The status word corresponds to the BASIC-PLUS STATUS variable.

  Table 4-3 shows the information, the tests, and the meaning of each bit.

- FLAGS -- Returns the flag word value as a 32-bit integer, which contains information about the file specification returned by the F$PARSE function. You can test each bit to determine information about elements found in the file specification. Note that if you specify a null file-spec, DCL returns the value 0, which indicates that no file specification was found. In addition, if you supply an invalid file-spec and include this keyword, DCL returns the value -1. Thus, anytime you request FLAGS, you must first check to see if the value returned is -1. If it is, then the file-spec is invalid, and the other bits are meaningless. Use FLAGS when you want to check for

an invalid file-spec without having DCL display an error message. The flag word corresponds to "flag word 2" in the BASIC-PLUS file name string scan system function call.

Table 4-4 shows the information, the tests, and the meaning of each bit.

Table 4-3: Status Word Values

Bit	Test	Meaning
0-7	(STATUS .AND. 255)	The first eight bits of the word contain the handler index. The following values apply for various devices: 0 Disk 2 Terminal 4 DECtape 6 Line Printer 8 Paper Tape Reader 10 Paper Tape Punch 12 Card Reader 14 Magnetic Tape 16 PK: device (pseudo keyboard) 18 DX: device (flexible diskette) 20 RJ: device (2780 remote job entry) 22 NL: null device 24 DMC11/DMR11/DDCMP Interface 26 Auto-Dialer 28 X-Y Plotter 30 Reserved 32 KMC11 34 IBM Interconnect 38 DMP11/DMV11
8	(STATUS .AND. 256).NE.0	The device is open for non-file-structured processing or is a non-file-structured device.
9	(STATUS .AND. 512).NE.0	The job does not have read access to the device.

Table 4-3: Status Word Values (Cont.)

Bit	Test	Meaning
10	(STATUS .AND. 1024).NE.0	The job does not have write access to the device.
11	(STATUS .AND. 2048).NE.0	The device maintains its own horizontal position. Such devices are keyboards and line printers.
12	(STATUS .AND. 4096).NE.0	The device accepts modifiers. Such devices use the record number as a modifier word rather than the physical position of the device. Keyboards, line printers, and card readers are such devices.
13	(STATUS .AND. 8192).NE.0	Device is a character device.
14	(STATUS .AND. 16384).NE.0	Device is an interactive device (keyboard).
15	(STATUS .AND. 32768).NE.0	Device is a random-access blocked device, such as disk and non-file-structured DECtape.
16-31		Reserved; returned as 0.

Table 4-4 shows the information, the tests, and the meaning of each bit.

Table 4-4: Flag Word Values

Bit	Test	Meaning
0	(FLAGS .AND. 1).NE.0	File name was found in the string.
	(FLAGS .AND. 1).EQ.0	No file name was found (and bits 1 and 2 of this word are also 0).
1	(FLAGS .AND. 2).NE.0	File name was an asterisk (*) character.
	(FLAGS .AND. 2).EQ.0	File name was not an * character.
2	(FLAGS .AND. 4).NE.0	File name contained at least one question mark (?) character.
	(FLAGS .AND. 4).EQ.0	File name did not contain any ? characters.
3	(FLAGS .AND. 8).NE.0	A period (.) was found.
	(FLAGS .AND. 8).EQ.0	No period was found, implying that no file type was specified (and bits 4, 5, and 6 of this word are also 0).
4	(FLAGS .AND. 16).NE.0	A file type was found (that is, the field after the period was not null).
	(FLAGS .AND. 16).EQ.0	No file type was found. (The field after the period was null -- and bits 5 and 6 of this word are also 0.)
5	(FLAGS .AND. 32).NE.0	File type was an * character.
	(FLAGS .AND. 32).EQ.0	File type was not an * character.

Table 4-4: Flag Word Values (Cont.)

Bit	Test	Meaning
6	(FLAGS .AND. 64).NE.0	File type contained at least one ? character.
	(FLAGS .AND. 64).EQ.0	File type did not contain any ? characters.
7	(FLAGS .AND. 128).NE.0	A project-programmer number (PPN) was found.
	(FLAGS .AND. 128).EQ.0	No PPN was found (and bits 8 and 9 of this word are also 0).
8	(FLAGS .AND. 256).NE.0	Project number was an * character; that is, the PPN was of the form [*,PROG].
	(FLAGS .AND. 256).EQ.0	Project number was not an * character.
9	(FLAGS .AND. 512).NE.0	Programmer number was an * character; that is, the PPN was of the form [PROJ,*].
	(FLAGS .AND. 512).EQ.0	Programmer number was not an * character.
10	(FLAGS .AND. 1024).NE.0	A protection code was found.
	(FLAGS .AND. 1024).EQ.0	No protection code was found.
11	(FLAGS .AND. 2048).NE.0	The protection code currently set as default by the current job was used.
	(FLAGS .AND. 2048).EQ.0	The assignable protection code was not used.
12	(FLAGS .AND. 4096).NE.0	A colon (:), but not necessarily a device name, was found in the string.

Table 4-4: Flag Word Values (Cont.)

Bit	Test	Meaning
	(FLAGS .AND. 4096).EQ.0	No colon was found (no device was specified); bits 13, 14, and 15 of this word are also 0.
13	(FLAGS .AND. 8192).NE.0	A device name was found.
	(FLAGS .AND. 8192).EQ.0	No device name was found; bits 14 and 15 of this word are also 0.
14	(FLAGS .AND. 16384).NE.0	Device name specified was a logical device name.
	(FLAGS .AND. 16384).EQ.0	Device name specified was an actual device name; bit 15 of this word is also 0.
15	(FLAGS .AND. 32768).NE.0	The logical device name specified was invalid for one of the following reasons: o The device name contained an underscore but did not correspond to any physical device on the system. o The device name did not contain an underscore but could not be translated to a physical device name.
	(FLAGS .AND. 32768).EQ.0	The device name specified, if any, was either an actual device name or a logical device name to which a physical device has been assigned.

Table 4-4: Flag Word Values (Cont.)

Bit	Test	Meaning
16-31		Reserved; returned as 0, except when the file-spec is invalid, in which case all bits (0-31) are set and the value returned is -1.

The following example uses the F$PARSE function to determine if a device is a disk:

```
$ !Define default list file
$ DEFAULT = "_SY:.LST"

$ !Prompt for output file-spec
$ INQUIRE OUTFIL "Output to"

$ !Build complete file-spec
$ OUTFIL = F$PARSE(OUTFIL,DEFAULT)

$ !Save device status
$ DEV_STS = F$PARSE(OUTFIL,,"STATUS")

$ !Ensure device is a disk
$ IF (DEV_STS .AND.  255) .NE.  0 THEN GOTO NOTDSK
        .
        .
        .
$ NOTDSK:
```

This example:

- Assigns the default-spec value "_SY:.LST" to the symbol DEFAULT.
- Uses the INQUIRE command to prompt for the file-spec to be parsed and assign the result to the symbol OUTFIL.
- Uses the F$PARSE function to return a complete RSTS/E file specification after parsing both OUTFIL and DEFAULT. DCL assigns the new file-spec to the symbol OUTFIL, overriding the previous assignment.

- Uses the F$PARSE function to assign the device status (as a 32-bit integer) to the symbol DEV_STS.
- Uses the IF command to test the device status to determine if the device is a disk.

F$PRIVILEGE

The F$PRIVILEGE function returns one of two values:

- The integer value 1 (true) if your job's current privileges match those specified in the list of privilege keywords
- The integer value 0 (false) if the job's privileges do not match

The format of the F$PRIVILEGE function is:

```
F$PRIVILEGE("[NO]privilege[,...]")
```

where:

[NO]privilege is a privilege keyword optionally preceded by "NO". If you specify several privilege keywords, you must separate them with commas (,) and you must enclose the entire list within quotation marks ("). When you specify [NO]privilege, the function returns the value 1 (true) only if the current job does not have the specified privilege.

Table 4-5 lists the valid privilege keywords and describes the function of each.

Table 4-5: Summary of Privileges

Privilege	Description
DATES	Change system clock and file dates.
DEVICE	Access restricted devices.
EXQTA	Exceed disk quota or memory maximum. (Not usually given to users; used by privileged programs.)
GACNT	Perform accounting operations on accounts in the user's group.
GREAD	Read or execute any file in the user's group, regardless of protection code.

Table 4-5: Summary of Privileges (Cont.)

Privilege	Description
GWRITE	Write or create/rename any file in the user's group, regardless of protection code.
HWCFG	Set hardware configuration parameters; for example, set terminal characteristics.
HWCTL	Control devices; for example, disable a device or hang up a dial-up line.
INSTAL	Install run-time systems, swap files, and resident libraries.
JOBCTL	Manipulate other jobs; for example, detach or kill a job.
MOUNT	Mount or dismount disks other than /NOSHARE.
PBSCTL	Control print/batch services; for example, turn servers on or off, change printer forms.
RDMEM	PEEK at memory. (Not usually given to users; used by privileged programs.)
RDNFS	Read disks non-file-structured.
SEND	Broadcast to terminals and send messages to restricted receivers.
SETPAS	Change your own password.
SHUTUP	Shut down the system.

Table 4-5: Summary of Privileges (Cont.)

Privilege	Description
SWCFG	Set software configuration parameters; for example, installation name.
SWCTL	Control software components; for example, turn DECnet/E on and off.
SYSIO	Perform restricted I/O operations; for example, gain write access to files in account [0,*], or set the privilege bit on executable files.
SYSMOD	Perform functions that could easily modify the system; for example, poke memory.
TUNE	Control system tuning parameters; for example, caching or job priority.
USER0-7	Available for customer applications. Not used by RSTS/E.
WACNT	Perform accounting operations on any account.
WREAD	Read or execute any file regardless of protection code.
WRTNFS	Read/write a disk non-file-structured.
WWRITE	Write any file regardless of protection code. Create/rename any file except in account [0,*].

The following example uses the F$PRIVILEGE function to determine if a user has HWCTL privilege:

```
$ HWCTL_PRIV = F$PRIVILEGE("HWCTL")
$ IF HWCTL_PRIV THEN GOTO CONTINUE
$ WRITE 0 "?Privilege HWCTL required"
$ EXIT
$ CONTINUE:
        .
        .
        .
```

In this command procedure, if the user has HWCTL privilege, the procedure continues to execute. If the user does not have HWCTL privilege, DCL displays a message on the user's terminal and the procedure exits.

F$RIGHT

The F$RIGHT function extracts a substring from a string starting at the position you specify and ending at the right-most position of the string. This function is similar to the BASIC-PLUS RIGHT function.

The format of the F$RIGHT function is:

```
F$RIGHT(string,position)
```

where:

string	is the string from which to extract the substring.
position	is an integer that specifies the starting position of the substring. If the position argument is negative or 0, the function returns the entire string. If the position argument is greater than the length of the string, the function returns a null string.

The following example uses the F$RIGHT function:

```
$ CHARS = F$RIGHT("ABCDEFGHI",6)
$ SHOW SYMBOL CHARS
CHARS = "FGHI"
```

In this example, the F$RIGHT function returns the string starting at position 6. DCL then assigns the resultant substring, "FGHI", to the symbol CHARS.

F$SEARCH

The F$SEARCH function searches a disk directory for the file you specify and returns its full RSTS/E file specification as a string.

Note

You cannot use the F$SEARCH function at the interactive level.

The format of the F$SEARCH function is:

```
F$SEARCH([file-spec])
```

where:

file-spec	is a string expression that specifies the file to locate. You must specify a file-spec the first time you use the F$SEARCH function. A complete RSTS/E file-spec consists of the fields: _dev:[proj,prog]filnam.typ You can use wildcard characters in the proj, prog, filnam, or .typ fields. If you specify an invalid file-spec or a device other than a disk, DCL displays an error message. DCL returns a null string when you do not include the file-spec argument in the initial use of the F$SEARCH function, or when the F$SEARCH function cannot find the specified file(s).

Use F$SEARCH to return information about one file, or about a series of files in your directory. You can also display information about files in other directories, if you have read access to those files.

The first example shows how to use the F$SEARCH function to obtain information about one file:

```
$ REPORT_FILE = F$SEARCH("REPORT.DAT")
```

After this command executes, the symbol REPORT_FILE is assigned the string value:

```
_SY:[4,214]REPORT.DAT
```

The second example uses the F$SEARCH function with a loop to obtain information about all files in your account that have the same file type:

```
$ NEXT_FILE = F$SEARCH("_SY:*.B2S")
$ LOOP:
$ IF NEXT_FILE .EQS.  "" THEN EXIT
$ WRITE 0 NEXT_FILE
$ NEXT_FILE = F$SEARCH()
$ GOTO LOOP
```

This example uses a wildcard and a loop to locate all occurrences of files with the file type .B2S. DCL returns a null string, after displaying all .B2S files in your directory.

If you have read access to other directories, you can also obtain information about files in those directories. For example:

```
$ SYSTEM_WIDE = F$SEARCH("_SY:[*,*]*.LST")
$ LOOP:
$ IF SYSTEM_WIDE .EQS.  "" THEN EXIT
$ WRITE 0 SYSTEM_WIDE
$ SYSTEM_WIDE = F$SEARCH()
$ GOTO LOOP
```

This command procedure displays all .LST files (in all directories on the system) to which you have read access.

F$STRING

The F$STRING function converts an integer expression to a string.

The format of the F$STRING function is:

F$STRING(expression)

where expression is an integer expression or symbol that equates to an integer expression.

The following example uses the F$STRING function:

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

In this example, the F$STRING function converts the result of the expression (-2 + A) to the numeric string "3". Note that the symbol A has the integer value 5.

F$TERMINAL

The F$TERMINAL function returns the keyboard (KB) number for the current job, as an integer.

The format of the F$TERMINAL function is:

F$TERMINAL( )

Use the F$TERMINAL function in command procedures that need to determine a job's keyboard number or determine if a job is currently detached. If the job is detached, this function returns a negative value, which is the two's complement of the terminal from which the job detached.

The following example shows the use of the F$TERMINAL function:

```
$ !Broadcast message to my terminal
$ TERM = F$TERMINAL
$ BROADCAST KB'TERM' "Assembly finished"
        .
        .
        .
```

F$TIME

The F$TIME function returns the current date and time.

The format of the F$TIME function is:

F$TIME()

This function returns a string of the form:

date time

where:

date	is the current date, in a format defined by your system manager. The date can have the format dd-mmm-yy or yy.mm.dd.
time	is the current time, in a format defined by your system manager. Time can be either 24-hour or AM/PM format.

The F$TIME function always returns a string. The length of the string depends on the format of the system date; no trailing blanks are returned. However, the time field is always returned starting at position 11 in the string, regardless of the format of the system date returned, because one or more spaces are inserted between the date and time fields to force the time string to begin at position 11.

The following example shows the use of the F$TIME function:

```
$ WRITE 0 "Program started on ", F$TIME
Program started on 12-Jun-85 07:36 AM
        .
        .
        .
```

F$TYPE

The F$TYPE function returns a string indicating the type of a symbol or expression.

The format of the F$TYPE function is:

```
F$TYPE(symbol)
F$TYPE(expression)
```

The F$TYPE function returns one of the following keyword strings:

Keyword	Meaning
INTEGER	The symbol has an integer value, or the expression result is an integer.
STRING	The symbol has a string value, or the expression result is a string.
null string	The symbol is not defined, or the expression contains one or more undefined symbols.

The following example uses the F$TYPE function:

```
$ !Initialize COUNT if not yet defined
$ IF F$TYPE(COUNT) .EQS.  "UNDEFINED" THEN COUNT == 0
```

In this example, the F$TYPE function determines the symbol's type. If COUNT is an undefined symbol, then the procedure starts counting at 0.

F$USER

The F$USER function returns the project-programmer number (PPN) for the current job.

The format of the F$USER function is:

F$USER()

The F$USER function returns a string of the form:

[proj,prog]

where:

proj	is the project number for the current job.
prog	is the programmer number for the current job.

Use this function in command procedures that need to use their own job's PPN as input to a task or compare and extract information from a list, such as a DIRECTORY or SYSTAT listing. For example:

```
$ ID = F$USER()
$ WRITE 0 "You are account ''ID'."
You are account [1,214]
        .
        .
        .
```

In this example, DCL assigns the PPN returned by F$USER to the symbol ID. The WRITE command then displays the PPN at the user's terminal because you specified channel 0.

F$VERIFY

The F$VERIFY function returns the current verification status as an integer, indicating 1 if verification is on, and 2 if verification is off. If you specify an argument, this function also turns verification on or off.

The format of the F$VERIFY function is:

```
F$VERIFY([value])
```

where:

value	is an optional argument that you can specify to turn verification on or off. DCL examines the low-order bit in the argument and turns verification off if the value is 0, or on if the value is 1.

Use the F$VERIFY function to enable or disable verification for a group of commands and then restore verification to its previous state. For example:

```
$ !Save verify state and set NOVERIFY
$ SAVE_VERIFY = F$VERIFY()
$ SET NOVERIFY
        .
        .
        .
$ !Restore verify state
$ IF SAVE_VERIFY THEN SET VERIFY
```

In this example, the assignment statement saves the current verification status in the symbol SAVE_VERIFY. Then the SET NOVERIFY command disables verification. The IF command tests the value of SAVE_VERIFY. If it is 1, showing that verification was previously on, the SET VERIFY command executes and verification is restored. If SAVE_VERIFY has a value other than 1, verification was initially off. In this case, the SET VERIFY command is not executed, so verification remains off.

When you use an argument, F$VERIFY still returns the current verification status. However, the command interpreter then examines 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:

```
$ !Save verify state and set NOVERIFY
$ SAVE_VERIFY = F$VERIFY(0)
        .
        .
        .
$ !Restore verify state
$ SAVE_VERIFY = F$VERIFY(SAVE_VERIFY)
```

In this example, the F$VERIFY function turns verification off, and then restores the previous setting at the end of the procedure.

Chapter 5

Interacting with Command Files

This chapter describes how to control input to and output from command procedures. It tells you how to:

- Pass data to a command procedure
- Return data from a command procedure
- Display data from a command procedure
- Supply data to a program in a command procedure

Passing Data

When you write a command procedure, you need to be able to pass input data to the procedure when it executes. DCL provides the following ways to do this:

- You can pass data to a command procedure using parameters
- You can issue a prompt to the user at the terminal, and the user can enter data interactively

The following sections describe these methods in greater detail.

Note

You can also read data from a file within a command procedure. For example, you could design a command procedure that displays informational records at a terminal, or that reads the records of an assembly listing file to search for possible errors.

See Chapter 6 for descriptions of the OPEN, READ, and CLOSE commands, which you use to read data from a file within a command procedure.

Passing Parameters

When you invoke a command procedure using the at (@) command, you can pass it up to eight parameters by placing the values of the parameters after the file specification of the command procedure. Separate the parameters with one or more spaces or tabs. You can specify a parameter value as one of the following:

- Literal -- Specify the literal with or without quotation marks ("), depending on whether you want to preserve spaces, tabs, and lowercase characters. The following example passes the values "JOHN" and "DOE" to DATA.COM:

  ```
  $ @DATA John Doe
  ```

  Use quotation marks to preserve spaces, tabs, or lowercase characters. The following example passes the single value "John Doe" to DATA.COM:

  ```
  $ @DATA "John Doe"
  ```

- Symbol -- To pass the value of a symbol, place an apostrophe (') before and after the symbol. The following example passes the values "JOHN" and "DOE" to DATA.COM:

  ```
  $ NAME = "John Doe"
  $ @DATA 'NAME'
  ```

  When DCL passes a symbol, it removes quotation marks that enclose a literal. To preserve spaces, tabs, and lowercase characters in a symbol value, specify the enclosing quotation marks as part of the symbol value. To include quotation marks as part of a literal value, double the quotation marks inside the literal string. The following example passes the single value "John "Mr. Average" Doe" to DATA.COM:

  ```
  $ @DATA "John ""Mr. Average"" Doe"
  ```

DCL treats all types of parameters as literal string values and assigns them to the local symbols P1 through P8: P1 is assigned the first parameter value; P2 the second; P3 the third; and so on. If you pass more than eight values, DCL returns an error message and does not execute the procedure. If you pass fewer than eight values, DCL assigns null values to the remaining symbols. In addition, you can pass a null parameter explicitly, by using double quotation marks in the list of parameters.

In the next example, you pass parameters to the procedure DATA.COM:

```
$ @DATA "John Doe" "" 24 "(603) 555-8003"
```

When DCL processes this command string, it assigns the following values to P1 through P8:

```
P1 = "John Doe"
P2 = ""
P3 = "24"
P4 = "(603) 555-8003"
P5 = ""
P6 = ""
P7 = ""
P8 = ""
```

When DCL enters a nested procedure, it assigns the local symbols P1 through P8 the new parameters passed by the invoking procedure. Note that the local symbols P1 through P8 in the nested procedure are not related to the local symbols P1 through P8 in the invoking procedure. For example, suppose DATA.COM invokes NEWDATA.COM with the command:

```
$ @NEWDATA 'P1'
```

In NEWDATA.COM, P1 through P8 are defined as follows:

```
P1 = "JOHN"
P2 = "DOE"
P3 = ""
P4 = ""
P5 = ""
P6 = ""
P7 = ""
P8 = ""
```

Prompting for Symbol Values

When you want a user to enter data interactively at a terminal, you can use the INQUIRE command to define a value for a symbol while the procedure is executing. INQUIRE prompts for a value at a user's terminal and waits for a response. After the user enters a response, INQUIRE reads the value from the terminal and assigns it to the specified symbol name.

Format

```
INQUIRE symbol-name [prompt-string]
```

Command Qualifiers	Defaults
/[NO]PUNCTUATION	/PUNCTUATION
/[NO]ECHO	/ECHO
/EXIT[=label]	See discussion
/TIME_OUT=seconds	See discussion
/GLOBAL	
/LOCAL	

Command Parameters

symbol-name

Is a name from 1 to 255 characters long. DCL assigns a string value to the symbol name.

prompt-string

Is the prompt to display at the terminal when the INQUIRE command executes. If you do not specify a prompt string enclosed in quotation marks, the command uses the symbol name to prompt for a value.

Command Qualifiers

/[NO]PUNCTUATION

Indicates whether to append a colon and a space to the prompt string when INQUIRE displays it. The default is /PUNCTUATION. To suppress the colon and space, use the /NOPUNCTUATION qualifier.

/[NO]ECHO

Indicates whether to display the user's response to the prompt at the terminal. The default is /ECHO. To suppress the display of sensitive information at the terminal, such as passwords, use the /NOECHO qualifier.

/EXIT[=label]

Specifies the label of the line in the command procedure to be given control when a CTRL/Z is encountered. If you do not specify /EXIT, or if you specify /EXIT without a label, the command procedure exits on a CTRL/Z.

/TIME_OUT=seconds

Specifies the number of seconds to wait for input. If no input is received within the specified time, DCL assigns a null string to the symbol name. The number of seconds must be in the range 1 to 32767.

/GLOBAL

Places the symbol in the global symbol table. /GLOBAL is the default at the interactive command level.

/LOCAL

Places the symbol in the local symbol table for the current command procedure. /LOCAL is the default for all command levels except the interactive level. DCL displays an error message when you specify /LOCAL at the interactive command level.

Examples showing how to use the INQUIRE command follow.

Use the /NOPUNCTUATION qualifier to suppress the colon and space that are normally added after the prompt. For example:

```
$ INQUIRE/NOPUNCTUATION IN "Input from?  "
$ INQUIRE/NOPUNCTUATION OUT "Output to?  "
```

When these commands execute, the following prompts appear at the terminal:

```
Input from?
Output to?
```

To define a global symbol name with the INQUIRE command, use the /GLOBAL qualifier. For example:

```
$ INQUIRE/GLOBAL FILE "Filename"
```

When this command executes, DCL enters the symbol name FILE and the response in the global symbol table.

To define a local symbol name with the INQUIRE command, use the /LOCAL qualifier. For example:

```
$ INQUIRE/LOCAL FILE "Filename"
```

When this command executes, the following prompt appears at the terminal:

```
Filename:
```

After you enter a response, DCL places the symbol name FILE and the value you enter in the local symbol table for the current command procedure.

If you press the RETURN key in response to an INQUIRE command, DCL assigns a null value to the specified symbol name. For example:

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

In this example, the INQUIRE command is followed by a test to determine whether a null value was entered. If a null value was entered, the procedure exits.

Returning Data

To return a value from a command procedure (either to a calling procedure or to the interactive level), you must assign the value to a global symbol. The global symbol can be read at any command level. Use comments to explain the use of any global symbols.

To create a global symbol, specify the value to be passed on the right side of a global assignment statement. For example:

```
$ @DATA "John Doe"

                        DATA.COM

$ !  P1 is a full name
$ !  NAME.COM returns the last name in the
$ !  global symbol LAST_NAME
$
$ @NAME 'P1'

                        NAME.COM

  $ !  P1 is a first name
  $ !  P2 is a last name
  $ !  return P2 in the global symbol LAST_NAME
  $ LAST_NAME == P2
  $ EXIT

$ !  write LAST_NAME to the terminal
$ WRITE 0 "LAST_NAME = ""'LAST_NAME'"""
LAST_NAME = "DOE"
```

In this example, DATA.COM passes a full name to NAME.COM. NAME.COM places the last name in the global symbol LAST_NAME. DATA.COM reads the value by referencing the global symbol.

Displaying Data

You can display data in a command procedure by using the WRITE command.

Use the WRITE command to display literal or symbol values, or a combination of both, at a terminal:

- Literal -- Enclose the text in quotation marks ("). The following example displays the text "Two files were written.":

```
$ WRITE 0 "Two files were written."
```

- Symbol Value -- The following example displays the text "STATUS.DAT":

```
$ FILE = "STATUS.DAT"
$ WRITE 0 FILE
```

- Combination of literals and symbol values -- Enclose the text in quotation marks. Where a symbol appears, precede it with two apostrophes and follow it with one apostrophe. The following example displays the text "STAT1.DAT and STAT2.DAT were written.":

```
$ AFILE = "STAT1.DAT"
$ BFILE = "STAT2.DAT"
$ WRITE 0 "''AFILE' and ''BFILE' were written."
```

See Chapter 6 for a complete description of the WRITE command.

Supplying Data to a Program

The following sections describe how to supply data to a program either from inside a command procedure or interactively at a terminal while the procedure executes.

Reading Data in a Program

When a program requiring terminal input runs directly under control of a keyboard monitor, the program receives its input directly from the user's keyboard. However, when the same program runs under control of a command procedure, all input normally received from the terminal is read from the current command file.

For example, the following program prompts a user for a list of numbers and prints the average value. In interactive mode, the program dialogue is:

```
$ RUN AVRAGE

How many numbers?  4
1st number?  86
2nd number?  91
3rd number?  89
4th number?  90
The average is 89.0

$
```

You can also run the AVRAGE program from a command procedure using a predefined set of numbers. The procedure would look like this:

```
$ RUN AVRAGE
4
86
91
89
90
$ EXIT
```

When the AVRAGE program requests data from the terminal, the monitor intercepts the request and reads the data lines from the command file instead. When you design this type of noninteractive command procedure, you must eliminate any unexpected prompting that could cause the procedure to read the wrong data.

You can also use the SET NODATA command (see the next section) to run a program from a command procedure that supplies the program with input from the terminal.

Supplying Data to a Program from a Terminal

By default, the SET DATA command is in effect when you invoke a command procedure. Thus, any data that a program or a keyboard monitor (other than DCL) requests must be supplied in the command file. However, you can redirect data input back to a terminal by issuing the SET NODATA command. The SET NODATA command tells DCL to read all data required by a program or a command from the terminal rather than from the command file.

Format	
SET [NO]DATA	
Command Qualifiers	**Defaults**
/END_OF_DATA="char"	/END_OF_DATA="$"

Command Qualifiers

/END_OF_DATA="char"

Identifies an alternate prefix character to signal the end of data. Whenever a line is read from the command file that begins with the alternate prefix character, control returns to the DCL keyboard monitor and any command following the alternate prefix executes.

The alternate prefix character stays in effect until the data list terminates. After control returns to DCL, the standard $ character is reenabled.

Note that you should never begin a data line with a $ character in the first space, even if you define an alternate prefix character with the SET DATA/END_OF_DATA command. In cases where a program requires its data to begin with a $ character, insert a leading space or a tab space before the $ character to distinguish it from a DCL command line. (Most programs routinely remove leading spaces and tabs from its data lines.)

For example, the following command file runs PIP to copy a file from your account to the system library ($) account:

```
$ SET DATA
$ RUN $PIP
<TAB>$FOO=BAR
$ EOD
```

Since PIP's data line begins with a tab space, RSTS/E correctly treats it as a data line. The tab space does not affect the PIP command, since PIP strips leading spaces and tab spaces. Note that if you did not include the tab space in this example, RSTS/E would handle the data line as a DCL command line -- in this case, by stopping PIP and executing the line as a DCL assignment statement.

Use the SET DATA command to supply data to a program from inside a command procedure, specifying an alternate prefix character that signals the end of data. For example:

```
$ SET DATA/END_OF_DATA="<"
$ CREATE LOGIN.COM
$RUN $SYSTAT
$SYSTAT.DAT
        .
     (data lines)
        .
<
$ EXIT
```

In this example, $RUN $SYSTAT and $SYSTAT.DAT are data lines for the CREATE command. When the alternate prefix character (<) is read, control returns to the DCL keyboard monitor. Note that if the prefix character were a $, the procedure would interpret the first $ in $RUN $SYSTAT as the end of data signal.

Use the SET NODATA command to supply a program with input from the terminal. The SET NODATA command stays in effect until you issue a SET DATA command or until the current command procedure exits.

In nested command procedures, each level maintains its own [NO]DATA setting and alternate prefix character. When a command procedure exits, DCL restores the [NO]DATA setting and alternate prefix character of the higher-level procedure.

The following example shows the command procedure AVRAGE.COM that runs the program AVRAGE.TSK, letting you supply data at your terminal:

AVRAGE.COM

```
$ SET NODATA
$ RUN AVRAGE
$ EXIT
```

Note that this procedure does not include data lines for the AVRAGE program. The SET NODATA command tells DCL to read data from the terminal rather than from the command file. For example, the dialogue at your terminal would appear as:

```
$ @AVRAGE

How many numbers?  4
1st number?  91
2nd number?  87
3rd number?  92
4th number?  90
The average is 90.0

$
```

By using the SET NODATA command, you can supply different sets of data to a program without modifying the command file.

Signaling the End of a Data List

Use the end-of-data ($EOD) command to signal the end of a data list and exit from a program within a command procedure.

Format

```
$EOD
```

The only function of the $EOD command is to terminate data lists to a program or to another DCL command. Although you can use any DCL command preceded by the $ character to exit from a program, the $EOD command provides a standard way to end a data list, and return to the DCL keyboard monitor.

Exiting from a Program in a Command File

The RSTS/E monitor takes the following steps to exit from a program and return to the DCL keyboard monitor. Whenever the monitor encounters a $ prefix character, a special prefix character, an $EOD command, or when it reaches the end of the current command file, it:

- Forces up to five CTRL/Z characters to the program, as needed. This step causes most RSTS/E programs to exit normally.

- Forces up to two CTRL/C exits to the program, as needed. This step aborts any program that does not perform its own CTRL/C handling.

- Forces a double CTRL/C exit to the program. This step aborts any program that uses CTRL/C as a normal way to exit.

- Aborts the program immediately, and returns control to the DCL keyboard monitor. This step is unconditional, and specifically guards against CTRL/C trapping.

Detaching Programs in a Command Procedure

When designing command files to run programs that detach, you generally want the command procedure to continue executing after a program detaches. For example, the START.COM start-up command file must continue processing after the OPSER program detaches.

RSTS/E allows a command procedure to continue executing when it runs programs that detach. If a program running within a command procedure detaches, RSTS/E detaches the current job, creates a new job at your terminal, and then transfers all of the DCL context from the detached job to the new job so that the command procedure can continue executing.

Therefore, you should consider the following points when writing command files that include programs that detach:

- RSTS/E does not create a new job if the detach request leaves the terminal either allocated to the job or open on any non-zero channels. In this case, the job detaches, and the command procedure detaches with it. You can set the "close" flag in the detach system call to deallocate the terminal and close any non-zero channels on which the terminal is open.

- RSTS/E does not detach a job if no job slots are available to create the new job.

- When a program detaches, the command procedure continues to execute, but in the newly created job; therefore, the F$JOB function will return a different job number than the number it returned before the program detached. See Chapter 4 for a complete description of the F$JOB function.

- If you later attach to a job that detached within a command procedure, it will not contain any DCL symbols, since all DCL syntax was moved to the newly created job.

Passing Symbol Values to a Program

Only applications that run under DCL can perform symbol substitution. Therefore, you must take a different approach when you need to pass a symbol value to a program running under control of another run-time system. For example, the following incorrect command procedure runs the PIP program under the RT11 run-time system:

MOVE.COM

```
$ RUN $PIP
LB:*.* = 'P1'
$ EOD
```

Suppose you execute this procedure with the command:

```
$ @MOVE TEST.FIL
```

You might expect the value TEST.FIL to be substituted for the symbol P1 during execution. However, this does not occur because the RT11 run-time system does not perform symbol substitution. Only DCL command lines can include symbol substitution.

To pass the value TEST.FIL to PIP, you must create a temporary command file and execute it as a nested procedure within MOVE.COM. For example:

MOVE.COM

```
$ OPEN/WRITE/REPLACE 1 MOVE1.COM
$ WRITE 1 "$RUN $PIP"
$ WRITE 1 "LB:*.* = ''P1'"
$ WRITE 1 "$EOD"
$ WRITE 1 "$EXIT"
$ CLOSE 1
$ @MOVE1
$ DELETE MOVE1.COM
```

You create a temporary command file using a series of OPEN and WRITE commands that are executed under DCL, which performs the symbol substitution. Now execute MOVE.COM with the command:

```
$
```

During execution, DCL substitutes the value TEST.FIL for the symbol P1, and PIP copies TEST.FIL from SY: to LB:.

Chapter 6

File Input and Output

This chapter describes how to combine DCL commands with the programming and symbolic capabilities of command procedures to manipulate disk files on RSTS/E systems. This chapter includes 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 you open

The basic steps in reading and writing to files from a command procedure are:

1. Open the file -- Use the OPEN command to open the file for reading or writing. You must also assign a channel number in the range 1 to 13 so that you can later refer to the file.
2. Read or write to the file -- Use the READ or WRITE command to read from or write to the file, specifying the channel number on which the file is open.
3. Close the file -- Use the CLOSE command to close the file, specifying the channel number on which the file is open. Note that a file stays open until you close it or until you log out of the system.

You can open, read, write, or close disk files that have the following file organizations and record formats:

- RSTS/E stream ASCII (native mode)
- RMS sequential

- RMS stream
- RMS variable
- RMS fixed
- RMS carriage control IMPLIED/NONE
- RMS span/nospan

Opening Files

The OPEN command opens a file either for reading or writing.

Format

OPEN channel-number file-spec

Command Qualifiers	Defaults
/APPEND	/READ
/READ	/READ
/[NO]REPLACE	See discussion
/WRITE	/READ

Prompts

Channel: channel-number
File: file-spec

Command Parameters

channel-number

Is a channel number in the range 1 to 13 on which to open the file. You can specify channel-number either as a literal or as a symbol that evaluates to a number. Subsequent READ, WRITE, and CLOSE commands refer to the file by using the channel-number. You get an error message if you specify a channel-number outside the range of 1 to 13, or when you specify a channel number already in use.

file-spec

Specifies the file to open for input or output. You can use any valid RSTS/E file specification; however, you cannot use wildcards in the file specification. If you omit the file type, it defaults to .DAT.

Command Qualifiers

/APPEND

Specifies that new records are to be appended to the end of an existing disk file. If the file does not exist, RSTS/E creates a new file.

Note

You can only use the /APPEND qualifier for RSTS/E stream ASCII (native mode) disk files. An error message displays if you use /APPEND for non-disk files or files with RMS attributes.

/READ

Opens a file for reading. /READ is the default when you do not specify a qualifier with the OPEN command.

/REPLACE
/NOREPLACE

Specifies whether to replace an existing file with a new file. If the file that you specify already exists:

- /REPLACE tells RSTS/E to delete the existing file and create a new file.
- /NOREPLACE tells RSTS/E not to replace the file if it exists; in this case, an error message displays.

If you specify neither qualifier, then RSTS/E issues a warning message and asks if you want to replace the file.

You should always specify this qualifier from within a command procedure if you are not sure the file exists. You can use the F$SEARCH function (see Chapter 4) to check the existence of a file within a command procedure.

Note that the /[NO]REPLACE qualifier requires the /WRITE qualifier and conflicts with the /APPEND qualifier.

/WRITE

Opens a file for writing. If the file already exists, then DCL deletes it and creates a new file (depending on the /[NO]REPLACE qualifier).

When you open a file, you specify whether it is to be read from or written to and assign it a channel number. Use the channel number in subsequent READ and WRITE commands to refer to the file. You cannot use READ or WRITE commands to access an open file either from within a program, or from keyboard monitors other than DCL. In addition, because channel 0 always specifies the user's terminal, you do not need to use an explicit OPEN command to write to the user's terminal.

The following example uses the OPEN command in a command procedure:

```
$ ON ERROR THEN GOTO OPEN_ERROR
$ FILE = "FILE1.DAT"
$ OPEN/READ 1 'FILE'
$ FILE = "FILE2.DAT"
$ OPEN/WRITE 2 'FILE'/REPLACE
$ ON ERROR THEN EXIT
        .
        .
        .
$ CLOSE 1
$ CLOSE 2
$ EXIT
$ OPEN_ERROR:
$ WRITE 0 "Error opening file ''FILE'"
$ STOP
```

In this example, the first assignment statement assigns the string value "FILE1.DAT" to the symbol FILE, and the OPEN/READ command opens the file for reading on channel 1. The second assignment statement assigns the string value "FILE2.DAT" to the symbol FILE, and the OPEN/WRITE command opens the file for writing on channel 2. If an error occurs while the system attempts to open either file, the procedure branches to the OPEN_ERROR routine, displays an error message, and returns to the interactive level. Otherwise, the procedure continues executing until it closes the open files and exits to either the calling command procedure or to the interactive level.

Reading Files

The READ command reads the next record from a file that you open for reading with the OPEN command and assigns the contents of the record to the symbol name you specify.

Format

READ channel-number symbol-name

Command Qualifiers	Defaults
/[NO]DELIMITER	/NODELIMITER
/END_OF_FILE=label	None
/GLOBAL	See discussion
/LOCAL	See discussion

Prompts

Channel: channel-number
Symbol: symbol-name

Command Parameters

channel-number

A channel-number in the range 1 to 13, that identifies the file to be read. You can specify channel-number either as a literal or as a symbol that evaluates to a number. You get an error message when:

- You specify a channel number outside the range 1 to 13.
- No file is open on the specified channel.
- The file on the specified channel is not open for reading.

symbol-name

Is the symbol name to which DCL assigns the contents of the record read in the READ command, as a string value. If you use the same symbol name for more than one READ command, each READ command redefines the value of the symbol name. The /GLOBAL and /LOCAL qualifiers determine whether the symbol is global or local.

Command Qualifiers

/[NO]DELIMITER

Tells DCL whether to include the record delimiter (such as CR/LF or FF) in the returned string. The default is /NODELIMITER. The /DELIMITER qualifier tells DCL to append the record delimiter to the string.

/END_OF_FILE=label

Specifies the label of the line in the command procedure to be given control when DCL detects an end-of-file (EOF) condition on the read. When this condition occurs, no record is returned. If you do not specify an /END_OF_FILE qualifier, the action taken depends on the current setting of the ON command. See Chapter 8 for a complete description of the ON command and error handling in command procedures.

/GLOBAL

Tells DCL to search the global symbol table for the symbol you specified. If DCL finds the symbol, it replaces its value. If DCL does not find the symbol, then it enters the symbol into the global symbol table. DCL searches the global symbol table by default when you execute a READ command at the interactive level.

/LOCAL

Tells DCL to search the local symbol table for the symbol you specified. If DCL finds the symbol, it replaces its value. If DCL does not find the symbol, then it enters the symbol into the local symbol table. DCL searches the local symbol table by default when you execute a READ command at command levels other than the interactive level. In addition, you get an error message if you specify /LOCAL at the interactive level.

Follow these steps to read data from a file:

1. Open the file -- The OPEN/READ command opens the file for read access and associates the file name with a channel number.

2. Begin the read loop -- File I/O is usually done in a loop unless you are reading or writing a single record.

3. Read the data from the file -- Use the READ command with the /END_OF_FILE qualifier to read the next record in the file and assign its contents to a symbol. The /END_OF_FILE qualifier causes DCL to pass control to the label you specify when you reach the end of the file. Generally, you specify the label that marks the end of the read loop.

4. Process the data -- Because you must read a file sequentially, process the current record before reading the next one.

5. Branch to the beginning of the loop -- You stay in the loop until you reach the end of the file.

6. End the loop and close the file -- The CLOSE command disassociates the file name from the channel number and closes the file.

When you issue a READ command, DCL returns the next record in the file as a string value and assigns it to the symbol name you specify. Note that the returned string does not include any trailing line delimiter characters.

You can read a record up to 255 characters long. You get an error message if DCL encounters a record that exceeds this limit.

DCL updates the $STATUS and $SEVERITY symbols whenever an error (other than an EOF trapped by the /END_OF_FILE qualifier) occurs on a read, and then bases control on the current ON setting. See Chapter 8 for more information.

The following example uses a READ command in a command procedure:

```
$ OPEN/READ 1 NOTICE.TXT
$ LOOP:
$ READ/END_OF_FILE=END 1 DATA
$ WRITE 0 DATA
$ GOTO LOOP
$ END:
$ CLOSE 1
$ EXIT
```

In this command procedure, you open the file NOTICE.TXT for reading on channel 1. The LOOP routine uses a READ command to read a record from the file opened on channel 1 and assign it, as a string value, to the symbol DATA. The WRITE command displays the symbol's value at the terminal. Each time the loop is executed, a new record is read and assigned to the symbol DATA, and written to the terminal, until the entire file is read and displayed at the terminal. When the procedure reaches EOF, control branches to the END routine, which closes the file before the procedure exits.

Writing Files

The WRITE command writes records to a file that you open for writing or appending with an OPEN command.

Format	
WRITE channel-number data[,...]	
Command Qualifiers	**Defaults**
/[NO]DELIMITER	/DELIMITER
Prompts	
Channel: channel-number	
Data: data[,...]	

Command Parameters

channel-number

A channel-number in the range 0 to 13 that identifies the file to be written. You can specify channel-number either as a literal or as a symbol that evaluates to a number. If you specify channel 0, the data is written to the terminal. You get an error message when:

- You specify a channel number outside the range 0 to 13
- No file is open on the specified channel
- The file on the specified channel is not open for writing or appending

data[,...]

The data to write to the file. The data list can contain one or more string expressions separated by commas. If you include integer expressions in the data list, DCL automatically converts them to string expressions before writing to the file.

DCL writes each data item to the file as a string with no intervening characters between each item. You must include spaces if you want to separate data strings written to the file.

Command Qualifiers

/[NO]DELIMITER

Tells DCL whether or not to write the string with trailing carriage return/line feed (CR/LF) characters. The default is /DELIMITER. The /NODELIMITER qualifier is useful when you need several WRITE commands to write a single record, or when you want a record delimiter other than CR/LF, such as form feed (FF).

Note that you specify an alternate record delimiter in the data list. For example:

```
$ FF = F$CHR(12)
$ WRITE/NODELIMITER 0 TEXT,FF
```

In this example, you define and use form feed as an alternate record delimiter.

DCL performs automatic symbol substitution when processing expressions; therefore, do not use apostrophes (') to enclose symbol names that you include in the data list of a WRITE command.

Whenever an error occurs on a write, DCL updates the $STATUS and $SEVERITY symbols accordingly, and then bases control on the current ON setting. See Chapter 8 for more information.

The following examples show how to use the WRITE command in command procedures:

```
$ LOOP:
$ INQUIRE/EXIT=END VAL1 "1st number"
$ VAL1 = F$INTEGER(VAL1)
$ INQUIRE VAL2 "2nd number"
$ VAL2 = F$INTEGER(VAL2)
$ WRITE 0 VAL1," + ",VAL2," = ",VAL1+VAL2
$ WRITE 0 VAL1," * ",VAL2," = ",VAL1*VAL2
$ GOTO LOOP
$ END:
$ EXIT
```

In this command procedure, the INQUIRE commands prompt for symbol values. The WRITE commands perform automatic symbol substitution when processing the expressions, displaying the result at the terminal. When the procedure executes, the following appears at the terminal:

```
1st number: 10
2nd number: 20
10 + 20 = 30
10 * 20 = 200
1st number: <CTRL/Z>
$
```

The next example shows how to read the records from one file and how to write those records selectively to a new file in a command procedure:

```
$ INQUIRE IN_FILE "Specify the file to read"
$ INQUIRE OUT_FILE "Specify the file to write"
$ OPEN 1 'IN_FILE'
$ OPEN/WRITE 2 'OUT_FILE'
$ LOOP:
$   READ/END_OF_FILE=DONE 1 RECORD
$   IF RECORD .EQS.  "" THEN GOTO LOOP
$   WRITE 2 RECORD
$ GOTO LOOP
$ DONE:
$ CLOSE 1
$ CLOSE 2
$ EXIT
```

This command procedure:

- Uses INQUIRE commands to prompt for a file to read (IN_FILE) and the new file to write (OUT_FILE)
- Uses OPEN commands to open both files
- Uses a read/write loop (specifying the label to branch to when the last record is read) to read the records from a file and write those records to a new file with blank lines suppressed
- After the last record is read, branches to the DONE routine and exits

Closing Files

The CLOSE command closes a file opened for reading, writing, or appending with the OPEN command and removes its channel number from the list of open channels.

Format
CLOSE channel-number
Command Qualifiers Defaults
/ALL
Prompts
Channel: channel-number

Command Parameters

channel-number

Is a channel-number in the range 1 to 13, on which to close the file. You can specify channel-number either as a literal or as a symbol that evaluates to a number. If you specify a channel number on which no file is currently opened, no error message appears. However, DCL does return an error if you specify a channel-number outside the range 1 to 13.

Command Qualifiers

/ALL

Tells DCL to close all open data files.

Any file that you open with an OPEN command stays open until you explicitly close it using the CLOSE command, or until you log out.

Note

If the system manager deletes your job, or if you delete a batch entry, and you have files open for writing or appending, the most recent data written to the file may be lost.

Note that, because channel 0 always specifies the user's terminal, any attempt to close channel 0 results in an error.

The following example uses the CLOSE command in a command procedure:

```
$ !Make sure that channels 1-5 are closed
$ CHANNEL = 1
$ LOOP:
$   CLOSE CHANNEL
$   IF CHANNEL .EQ.  5 THEN GOTO END
$   CHANNEL = CHANNEL + 1
$ GOTO LOOP
$ END:
$ EXIT
```

In this command procedure, you assign the value 1 to the symbol CHANNEL. Each time the loop is executed, DCL closes a channel, increments the channel number, and so forth, until all five channels are closed.

Chapter 7

Controlling Execution Flow in Command Procedures

Normal execution flow in a command procedure is sequential: DCL executes the commands in the procedure in order, until reaching the end-of-file (EOF). However, in some cases you may want to repeat a series of commands, skip a series of commands, or abort the command procedure.

The basic commands for controlling execution flow in a command procedure are:

- 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 label in the procedure
- The at (@) command -- Invokes another command procedure and begins execution at another command level
- The EXIT and STOP commands -- Terminate the current procedure and restore control to either the calling command procedure or the interactive command level

The IF Command

The IF command tests the value of an expression and executes the DCL command after the THEN keyword if the result of the expression is true. An expression is true if its integer result is odd, otherwise it is false. DCL converts string values to integer values before performing the test. See Chapter 3 for a description of how DCL converts strings to integers.

Format

```
IF expression THEN command
```

Command Parameters

expression

Defines the test to be performed. The test may consist of any valid DCL expression. An expression is true if the result is odd and false if the result is even.

See Chapter 3 for a summary of operators and details on how to specify expressions.

command

Defines the action to take if the result of the expression is true. You can specify any valid DCL or CCL command after the THEN keyword, optionally preceded by a $ character.

If the result of the expression is false, control resumes at the next line in the command procedure.

The following example uses the IF command:

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

In this example, the IF command sets up a loop in a command procedure. The IF command checks the value of the symbol COUNT and performs an EXIT command when the value of COUNT is greater than 10.

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 example, the IF command tests each expression 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 the IF command expressions are true, RESULT is assigned a value of 1; otherwise, the assignment statement is not executed.

You can use the @ command in an IF command to invoke another command procedure. For example:

```
$ IF A .EQ. B THEN @FILE2
```

If the result of the expression A .EQ. B is true, DCL executes the nested procedure FILE2.COM. After the nested procedure executes, control returns to the next line in the calling procedure.

Command Line Labels

Command line labels let you identify lines in a command procedure to which control passes when a GOTO command executes. 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 command line and must be preceded by a $ character
- A label must end with a colon (:)
- You can specify only one label on a command line

Keep the following in mind when you enter labels:

- Excessive use of labels can exhaust the space available for local symbols
- DCL returns an error when insufficient space exists for storing a label definition
- DCL does not allow symbol substitution (apostrophes) in labels because it performs label processing before it performs symbol substitution

When DCL finds a label at the beginning of a command line, it enters the label into a local label table that shares space in the local symbol table. If the label already exists in the table, DCL displays an error message and immediately ends the procedure with a STOP command. For example:

```
$ LABEL1:
        .
        .
        .
$ GOTO LABEL2
        .
        .
        .
$ LABEL1:
        .
        .
        .
$ LABEL2:
        .
        .
        .
```

In this command procedure, DCL enters LABEL1 in the local label table while processing the procedure. When DCL finds the GOTO command, LABEL2 is not yet in the local label table. DCL scans forward through the procedure to locate the label. While scanning forward, however, DCL finds a duplicate LABEL1, displays an error message, and terminates the procedure with a STOP command. Control immediately returns to the interactive level.

The GOTO Command

The GOTO command passes control to a labeled line in a command procedure.

Format

```
GOTO label
```

Command Parameters

label

Specifies a 1 to 255 character label that appears as the first item on a command line. When the GOTO command executes, control passes to the command following the specified label.

The label can precede or follow the GOTO command in the current command procedure.

The GOTO command is useful after a THEN clause to cause a procedure to branch forward or backward according to variable conditions or parameters that you pass to the procedure. For example:

```
$ IF P1 .EQS.  "" THEN GOTO ERROR
        .
        .  (normal processing)
        .
$ ERROR:
        .
        .  (error processing)
        .
```

In this example, DCL tests to determine if P1 is null. If it is, then control passes to the label ERROR, where error processing occurs. If P1 is not null, then control resumes at the next line after the GOTO command.

You can also use the GOTO command to set up loops in a command procedure. For example:

```
$ NUM = 0
$ LOOP:
$   NUM = NUM + 1
        .
        .
        .
$   IF NUM .LT.  11 THEN GOTO LOOP
        .
        .
        .
```

In this example, you use the GOTO command in a loop that executes a specified number of times. The first line in the procedure sets up a counter named NUM. The loop increases the value of the counter, does some processing and then tests the counter's value. The command procedure exits from the loop when the value of NUM is greater than or equal to 11.

Controlling Execution Flow in Command Procedures

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

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

In this example, the INQUIRE command requests a file name. If the user's response is a null value, the loop does not execute. Otherwise, the loop executes iteratively until the user enters a null value by pressing only the RETURN key.

Nesting Command Procedures

The GOTO command described in the previous section provides one way to divide command procedures into more easily read and understood sections. In a more complex procedure, however, you may want to separate different sections into several smaller procedures. Or you may find it convenient to develop small, generalized procedures that perform common functions and then invoke these procedures from other procedures that you write. You can call one command procedure from inside another by using the @ command. Using the @ command to invoke new levels of command execution is similar to using a CALL statement in a high-level programming language.

When you enter a procedure, the command level increases by one. For instance, if you invoke procedure SUB from interactive command level (level 0), SUB executes at command level 1. If SUB then calls SUB1, which calls SUBSB1, SUB1 executes at command level 2 and SUBSB1 at command level 3. The deepest permissible command level is 13.

By convention, the interactive level is the highest command level; and command level 13 the lowest. Therefore, if you move from command level 3 to command level 2, you move to the next higher command level.

Figure 7-1 shows the flow of control when you execute nested command procedures; the figure also shows how DCL creates new command levels when you execute nested command procedures.

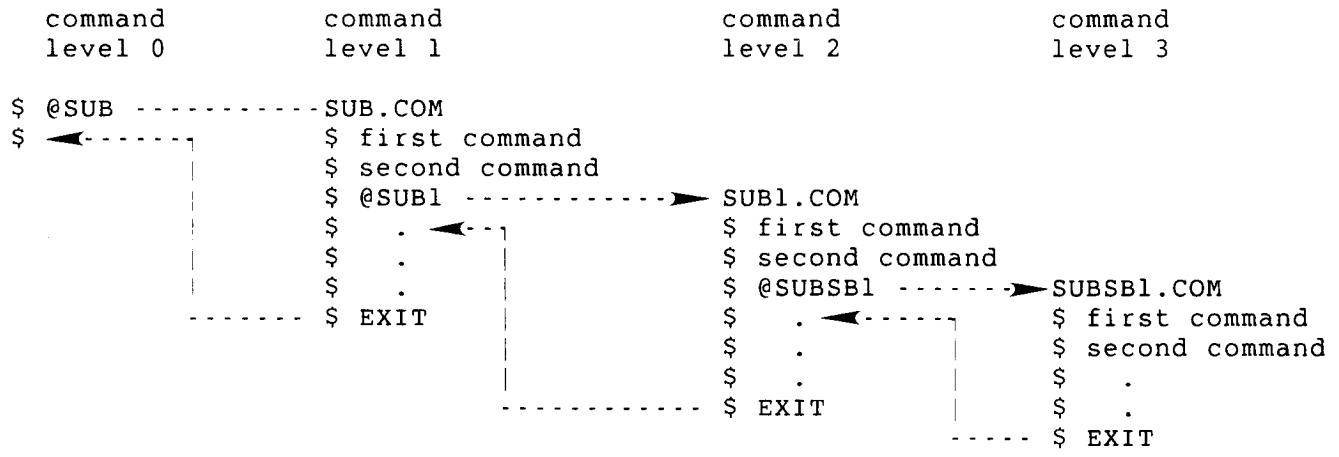

Figure 7-1: Command Levels in Nested Command Procedures

If you need to pass information from one command level to another, use one of the following techniques:

- Passing parameters -- You can pass up to eight parameters to a procedure you invoke using the @ command. See Chapter 5 for a description of techniques for passing parameters.

- Using global symbols -- You can use global symbols to pass data from one procedure to another; a global symbol defined in a nested command procedure can be referred to in all command procedures. See Chapter 2 for a description of global symbols.

- Using data files -- You can write data to a data file using the WRITE command, then read the data back in a nested command procedure using the READ command. See Chapter 6 for information on writing and reading files.

Exiting from a Command Procedure

A command procedure exits when it reaches the end of the procedure, an EXIT command, or a STOP command. If the exit is caused by the end of the procedure or an EXIT command, control returns to the next higher command level. For instance, if you invoke SUB at interactive command level, and SUB calls SUB1, then:

- Exiting from SUB1 returns you to SUB at the command line following the call to SUB1.
- Exiting from SUB returns you to interactive command level.

When you use the EXIT command to cause an exit, you can return a status value to the next higher command level by specifying the value as the parameter of the EXIT command. This status value is placed in the global symbols $STATUS and $SEVERITY. Upon returning from the lower level routine, the higher level procedure performs any error handling based on its ON...THEN setting and the value of $SEVERITY. See Chapter 8 for a description of error handling in command procedures.

If the STOP command causes the exit, control immediately returns to interactive command level.

The following sections describe the EXIT and STOP commands in greater detail.

The EXIT Command

The EXIT command ends a command procedure and returns control to the next higher command level.

Format

```
EXIT [status-code]
```

Command Parameters

status-code

A numeric expression that defines a value for the reserved global symbols $STATUS and $SEVERITY. Note that the reserved global symbol $SEVERITY is affected by the value placed in the $STATUS symbol. You can specify the status-code as an integer or an expression. If you do not specify a status-code, DCL does not change the values of $STATUS and $SEVERITY from the most recently executed command or program. See Chapter 8 for a complete description of the $STATUS and $SEVERITY symbols.

You can use the EXIT command to make sure that a procedure does not execute certain lines. For example, if you write an error-handling routine at the end of a procedure, you can place an EXIT command before the routine:

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

The EXIT command is also useful in 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 if either of these strings was entered. The GOTO command branches appropriately, using the parameter as the branch label. If P1 is not TAPE or DISK, the INQUIRE command prompts for a correct parameter; the GOTO START command establishes a loop.

The commands after 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 an implicit exit occurs when DCL detects the end-of-file (EOF). However, DIGITAL recommends that you use the EXIT command at the end of a command procedure.

When a command procedure has multiple levels of interaction, you can use the EXIT command to pass status values from nested levels back to the calling procedure. The exit code defines values for the reserved global symbols $STATUS and $SEVERITY.

For example, suppose the procedure A.COM contains these lines:

```
$ @B
$ IF $STATUS .EQ.  2 THEN GOTO CONTROL
        .
        .
        .
```

In addition, the procedure B.COM contains the line:

```
$ EXIT 2
```

This EXIT command places the value 2 in the global symbols $STATUS and $SEVERITY. The calling procedure, A.COM, tests $STATUS.

The STOP Command

The STOP command ends a command procedure and immediately returns control to the interactive level, thus terminating any intermediate command procedures.

Format

```
STOP
```

The following example uses the STOP command:

```
$ ON ERROR THEN GOTO OPEN_ERROR
$ OPEN 2 FILE/READ
        .
        .
        .
$ OPEN_ERROR:
$   WRITE 0 "Error opening file"
$   STOP
```

In this command procedure, if an error occurs while attempting to open a file, the procedure goes to the label OPEN_ERROR, prints a message, stops executing, and returns immediately to the interactive level.

Chapter 8

Controlling Error Conditions and CTRL/C Interrupts

This chapter describes how to control command procedure execution when an error condition or a CTRL/C interrupt occurs. An error condition occurs when a command does not terminate successfully. A CTRL/C interrupt is the result of pressing CTRL/C during command procedure execution.

Error Condition Handling

Error conditions are stored as codes in the reserved global symbol $STATUS. If an EXIT command does not explicitly set a value for $STATUS, then DCL returns its current value. This value is set implicitly by individual commands and programs that execute in a procedure. The value that is set, called a condition code, provides information about the execution of:

- The most recent command
- The most recent program
- The most recent procedure

The following sections describe how you can include action routines and error handling statements in your procedures based on values in $STATUS.

$STATUS and $SEVERITY Symbols

DCL reserves two global symbols, $STATUS and $SEVERITY, to maintain an error condition code called "exit status."

The lowest three bits of the $STATUS value represent the severity of the exit status. The remaining bits of the $STATUS value are reserved, and are currently undefined. The $SEVERITY symbol contains

the same value as the lowest three bits of the $STATUS value.

DIGITAL recommends that you test severity values using the $SEVERITY symbol instead of the $STATUS symbol. This lets you avoid unnecessary modifications in the future, if other bits in $STATUS are defined.

Table 8-1 lists the severity values and their meanings.

Table 8-1: Severity Values

Value	Severity	Meaning
0	WARNING	This condition indicates that an operation completed, but not necessarily as expected. For example, if you try to mount a disk initialized read-only using the /WRITE qualifier, the following warning message appears: %Disk is mounted read-only RSTS/E prefixes all WARNING messages with the percent sign (%) character.
1	SUCCESS	This condition indicates that an operation completed successfully, that is, with no errors or warnings. Informational or action messages also receive this status value.
2	ERROR	This condition indicates that an operation did not complete. For example, an ERROR message appears after you enter an illegal command or after you enter a command using invalid syntax. RSTS/E prefixes all ERROR messages with the question mark (?) character.
4	SEVERE_ERROR	This condition indicates that an operation did not complete, but for more serious reasons. RSTS/E prefixes all SEVERE_ERROR messages with double question marks (??).

Note that the success code has an odd numeric value, while warning and error codes have even numeric values. Because DCL evaluates all odd values as true (1) and all even values as false (0), you can use the IF command to test the severity codes and determine if an operation completed successfully. For example:

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

This IF command executes the command after the THEN clause when the severity value is odd, indicating success.

```
$ IF .NOT. $SEVERITY THEN ...
```

This IF command executes the command after the THEN clause if the severity value is even, indicating that a warning, error, or severe error occurred.

Note

The following DCL commands, if they complete successfully, do not affect the $STATUS or $SEVERITY setting:

- GOTO
- STOP
- $EOD
- IF
- SHOW SYMBOL
- ON (see the next section for more information)
- an assignment statement

The ON Command

By default, DCL performs an EXIT command when an error or severe error occurs and continues processing when a warning occurs. You can override this default with the ON command.

The ON command specifies an action to be performed whenever DCL detects a warning, error, or severe error in a command procedure. Whenever a command, program, or command procedure completes, DCL checks the exit status and takes action based on the current setting of the ON command.

Format

```
ON severity-level THEN command
```

Command Parameters

severity-level

Specifies the severity level at which to perform the command action. You can specify one of the following severity-level keywords:

- WARNING
- ERROR
- SEVERE_ERROR

command

Specifies the CCL or DCL command to be executed if the status code returned from the last command, program, or command procedure is greater than or equal to the severity level you specify. You can specify any valid CCL or DCL command after the THEN keyword, optionally prefixed with a $ character.

When DCL finds an ON command, it stores the command following the THEN keyword in a special area of the local symbol table and executes it whenever an error greater than or equal to the specified severity level occurs.

If you define an ON command action for a specific severity level, DCL performs the specified action when errors of the same or worse severity occur. When less severe errors occur, DCL continues to process the file.

Table 8-2 summarizes how the ON command controls error handling.

Table 8-2: ON Command Keywords and Actions

ON Keyword	Action Taken
WARNING	Command procedure performs the specified action if a warning, error, or severe error occurs.
ERROR	Command procedure performs the specified action if an error or severe error occurs; the procedure continues if a warning occurs.
SEVERE_ERROR	Command procedure performs the specified action if a severe error occurs; the procedure continues if a warning or error occurs.

For example, if you want a command procedure to exit whenever a warning, error, or severe error occurs, use the command:

```
$ ON WARNING THEN EXIT
```

If you want the command procedure to continue if a warning or an error occurs, but to branch to another part of the file if a severe error occurs, use a command such as:

```
$ ON SEVERE_ERROR THEN GOTO ERROR_TRAP
```

This ON command requests that the procedure branch to the label ERROR_TRAP only in the case of a severe error. If any command in the procedure causes a warning or error condition, execution continues with the next command in the procedure.

Note that severe errors always cause the command following the THEN keyword to be executed, regardless of the severity level you specify in the ON command. You must disable error checking with the SET NOON command for processing to continue when a severe error occurs.

When DCL executes the command following the THEN keyword, it resets the ON setting to its default (ON ERROR THEN EXIT). For example:

```
$ ON WARNING THEN GOTO ERROR_TRAP
```

In this example, if a warning occurs in the command procedure then control passes to the first command following the label ERROR_TRAP, and the ON setting is reset to ON ERROR THEN EXIT. If you want to restore the previous ON setting, you must reissue the ON WARNING THEN GOTO ERROR_TRAP command.

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 invokes another procedure, the ON command action does not apply to the nested procedure.

Enabling and Disabling Error Checking

The SET NOON command lets you override default error checking. You can use SET NOON to tell DCL not to check the status code returned from the last command, program, or command procedure executed.

Format
SET [NO]ON

During command procedure execution, DCL normally checks the status code returned when a command, program, or command procedure completes, and saves the numeric value of this code in the reserved global symbol $STATUS. The low-order three bits of this value are also saved in the reserved global symbol $SEVERITY.

When SET NOON is in effect, DCL continues to place the status code value in $STATUS and the severity level in $SEVERITY, but does not perform any action based on the value. As a result, the command procedure continues to execute regardless of how many errors are returned.

Like SET ON, the SET NOON command applies only at the current command level. If you use the SET NOON command in a command procedure that executes another procedure, DCL establishes the default, SET ON, while the second procedure executes.

The following example uses the SET NOON and SET ON commands:

```
$ ON WARNING THEN EXIT
$ SET NOON
$ MOUNT MM0:
        .
        .
        .
$ SET ON
        .
        .
        .
```

In this example, you temporarily disable error checking and attempt to mount a tape. If any error occurs, processing continues. The SET ON command restores the current setting of the ON command; that is, the procedure exits if later warnings, errors, or severe errors occur.

Note

> Whenever an input error other than EOF occurs when reading from a command file, DCL displays an error message, generates a SEVERE_ERROR, and terminates the command procedure. DCL takes this action regardless of the current ON setting or severity level.

CTRL/C Interrupt Handling

You can design a command procedure so that DCL takes a certain course of action when the user presses CTRL/C during execution of the procedure.

When the user presses CTRL/C at a logged-in terminal, the RSTS/E terminal service routine sets a CTRL/C event flag. DCL initially clears this flag when a user logs in to the system. Once the CTRL/C flag is set, it remains set until examined by a monitor directive, which clears the flag automatically. This design lets DCL read the CTRL/C event flag to determine if a CTRL/C interrupt occurs during command procedure execution.

If you execute a program within a command procedure that is designed to abort or to perform clean-up operations and then abort when you press CTRL/C, the CTRL/C event flag remains set and DCL takes action based on the ON CONTROL_C setting.

Some programs (such as MAIL), however, are designed to continue to operate properly when the user presses CTRL/C. In this case, DCL will take CTRL/C action when the program exits. Because this may not be desirable, make sure that the event flag remains clear by issuing the

SET NOCONTROL=C command to disable CTRL/C checking before you execute the program. Then, after the program executes, you can issue the SET CONTROL=C command to reenable checking.

In addition, you can completely disable a terminal's CTRL/C key by using the SET TERMINAL/NOCONTROL=C command.

Note

> You should use the SET NOCONTROL=C and SET TERMINAL/NOCONTROL=C commands for special applications that have been thoroughly tested. Generally, DIGITAL does not recommend that you disable CTRL/C interrupts.
>
> For example, if a procedure that disables CTRL/C 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 manager to terminate it for you.

The following sections describe how to establish a CTRL/C action routine, and how to enable or disable CTRL/C checking.

Setting a CTRL/C Action Routine

The ON CONTROL_C command specifies the action to take when a CTRL/C interrupt occurs during execution of a command procedure. The specified action applies only within the command procedure in which the command is executed.

Format

```
ON CONTROL_C THEN command
```

Command Parameters

command

> Specifies the DCL or CCL command to be executed if a CTRL/C condition occurs. You can specify any valid DCL or CCL command after the THEN keyword, optionally prefixed with a $ character.

Using the standard rules of substitution, DCL first performs a syntax check and substitutes any symbol value following the THEN clause. DCL

then stores the command in a special area of the local symbol table and executes it whenever a CTRL/C interrupt occurs.

The following example shows the use of ON CONTROL_C:

```
$ ON CONTROL_C THEN GOTO CTRL_EXIT
        .
        .
        .
$ CTRL_EXIT:
$ CLOSE 1
$ CLOSE 2
$ EXIT
```

When a CTRL/C interrupt occurs during execution of this procedure, DCL executes the GOTO command, which transfers control to the line labeled CTRL_EXIT:. At this label, the procedure performs clean-up operations. In this example, clean-up consists of closing files opened on channels 1 and 2, and exiting.

When you do not specify a CTRL/C action routine and a CTRL/C interrupt occurs, DCL executes an EXIT command by default and returns control to the next higher-level command procedure. Because control returns with the CTRL/C event flag still set, the higher-level procedure takes whatever action you specify for CTRL/C handling at that level.

When you do specify a CTRL/C action routine, anytime DCL takes a CTRL/C action, the default setting (ON CONTROL_C THEN EXIT) is reinitialized in the current procedure. Therefore, to establish a setting other than the default, the procedure must reexecute the ON CONTROL_C command after a CTRL/C interrupt occurs.

Disabling and Reenabling CTRL/C Interruptions

The SET NOCONTROL=C command disables CTRL/C checking by DCL within a command procedure. That is, if a command procedure executes the SET NOCONTROL=C command, pressing CTRL/C will not cause the ON CONTROL_C command to be executed. Pressing CTRL/C will still have its normal effect on running programs (unless a SET TERMINAL/NOCONTROL=C command is in effect).

On the other hand, the SET CONTROL=C command reenables CTRL/C checking. THE SET [NO]CONTROL=C setting applies only at the current command level.

Format
SET [NO]CONTROL=C

Use the SET NOCONTROL=C command in procedures where you do not want DCL to take special action if a CTRL/C interrupt occurs. For example:

```
          .
          .
          .
  $ SET NOCONTROL=C
  $ RUN PROG1
  $ SET CONTROL=C
          .
          .
          .
```

In this example, the command procedure executes a program, which could generate a CTRL/C condition. You issue the SET NOCONTROL=C command, execute the program, and then reenable CTRL/C checking with the SET CONTROL=C command.

Chapter 9

Controlling Terminal Output

RSTS/E provides the following ways to enable and disable command line display and program output to your terminal during its execution of a command procedure:

- SET ECHO command
- SET VERIFY command
- Terminal logging commands

You can use these commands interactively or within a command procedure. However, they only affect output to your terminal when a command procedure is executing.

By default, RSTS/E displays all output except DCL command lines during command procedure execution. However, you may sometimes want to disable all terminal output for "silent" command procedure processing, or to enable command line display to help debug a command procedure.

SET [NO]ECHO Command

You use the SET ECHO and SET NOECHO commands when you want to enable and disable the display of all output from a command procedure on your terminal.

Format	
SET [NO]ECHO	
Command Qualifiers	**Defaults**
/[NO]WARNINGS	/WARNINGS

Command Qualifiers

/[NO]WARNINGS

Specifies whether warnings and error messages display on the terminal. You can use this qualifier only with the SET NOECHO command.

When ECHO is in effect, RSTS/E displays the following during command procedure execution:

- All program data written to or echoed on the terminal
- Warnings and error messages
- DCL command lines if SET VERIFY is in effect (see the following section for a description of the SET [NO]VERIFY command)

When you first log in to the system, ECHO is in effect. You can issue the SET ECHO or SET NOECHO command either at the interactive level or within a command procedure to change its setting. The [NO]ECHO setting is global to all command procedures and remains in effect until you change it.

When NOECHO is in effect, RSTS/E displays only the following:

- Warnings and error messages (unless you specify the /NOWARNINGS qualifier)
- Messages broadcast to your terminal
- Prompts from an INQUIRE command
- Data output with the WRITE 0 command

RSTS/E suppresses all DCL command lines and other program data written to the terminal when NOECHO is in effect. If you specify the /NOWARNINGS qualifier with the SET NOECHO command, then warnings and error messages also do not display.

Note that NOECHO overrides the [NO]VERIFY setting, described in the next section.

SET [NO]VERIFY Command

You use the SET VERIFY and SET NOVERIFY commands when you want to enable and disable the display of all DCL command lines as they occur during command procedure execution. See Chapter 2 for a complete description of the SET [NO]VERIFY command.

SET [NO]VERIFY also controls whether RSTS/E writes the command lines to a log file. See the next section for a complete description of the terminal logging feature.

If SET VERIFY is in effect, command lines are displayed exactly as they appear in the procedure, before any substitution has been performed. You can specify the /DEBUG qualifier to display the command line as it appears after substitution.

When you first log in to the system, NOVERIFY is in effect. You can issue the SET VERIFY or SET NOVERIFY command either at the interactive level or within a command procedure to change its setting. The [NO]VERIFY setting is global to all command procedures and remains in effect until you change it.

When NOECHO is in effect (see the previous section), command lines are not displayed at the terminal, regardless of the VERIFY setting. However, if VERIFY is in effect and a log file is open (see the next section), RSTS/E continues to write the command lines to the log file.

Creating a Log File of a Terminal Session

The terminal logging feature lets you create and use a terminal log file to save a copy of the output that appears during the execution of a command procedure. See the *RSTS/E System User's Guide* for a complete description, with examples, of terminal logging.

Note that if you enable a log file during execution of a command procedure, the SET NOVERIFY command (see the previous section) disables output of command lines both to the terminal and to the log file.

The following sections describe the commands that you use to:

- Open a terminal log file
- Close a terminal log file
- Selectively disable and enable output to the log file

OPEN/LOG_FILE

The OPEN/LOG_FILE command opens a disk file for terminal logging. You can issue the OPEN/LOG_FILE command either interactively or within a command procedure.

You can only open one log file at any given time. If a log file is already open when you attempt to open a second log file, RSTS/E issues an error message and does not open the second file; instead, the current log file remains open.

Format

OPEN/LOG_FILE file-spec

Command Qualifiers	Defaults
/APPEND	See discussion
/DISABLE	/ENABLE
/ENABLE	/ENABLE
/[NO]REPLACE	See discussion
/[NO]TIME_STAMP	/NOTIME_STAMP

Prompts

File: file-spec

Command Parameters

file-spec

Specifies the disk file to open for logging terminal output. RSTS/E displays an error message if you specify a nondisk file, or if the file-spec is invalid. RSTS/E also displays an error message if a log file is already open. If you do not specify a file type, RSTS/E assumes a file type of .LOG. Unless you include the /APPEND qualifier, RSTS/E opens the file for output, and deletes any existing file of the same name.

Command Qualifiers

/APPEND

Tells RSTS/E to add data to the end of the file you specify. This qualifier lets you append terminal output to the end of an existing log file. If the file you specify does not exist, then RSTS/E ignores this qualifier and opens a new file. Note that

only RSTS/E stream ASCII files can be appended: if you specify a file that has RMS attributes, an error message displays.

/DISABLE

Indicates that terminal output should not be logged to the file until you issue the SET LOG_FILE/ENABLE command to enable terminal logging.

/ENABLE

Indicates that output should be logged to the file. This qualifier is the default when you issue the OPEN/LOG_FILE command.

/[NO]REPLACE

Specifies whether to replace an existing file with a new file. If the log file that you specify already exists:

- /REPLACE tells RSTS/E to delete the existing file and create a new file.
- /NOREPLACE tells RSTS/E not to replace the file if it exists; in this case, an error message displays.

If you specify neither qualifier, then RSTS/E issues a warning message and asks if you want to replace the file.

Note that the /[NO]REPLACE qualifier conflicts with the /APPEND qualifier.

/[NO]TIME_STAMP

Indicates whether to prefix each line in the log file with a date/time stamp. The default is /NOTIME_STAMP. However, if you specify /TIME_STAMP, RSTS/E prefixes each line in the log file with a date/time stamp specifying the date and time that the line was copied to the log file. Note that the date and time format is based on the system defaults that your system manager establishes. The date/time fields occupy the first 22 characters of each line.

CLOSE/LOG_FILE

The CLOSE/LOG_FILE command closes a log file that you opened with the OPEN/LOG_FILE command.

Format

CLOSE/LOG_FILE

Use the CLOSE/LOG_FILE command to close an open log file either during a session at your terminal or within a command procedure.

If logging is enabled when you issue the CLOSE/LOG_FILE command, then RSTS/E writes the command itself to the log file before closing it. If no log file is open when you issue this command, then RSTS/E displays an error message.

SET LOG_FILE

After you open a log file, you can use the SET LOG_FILE command to selectively enable and disable logging to the file. You can also use this command to enable or disable the time stamp feature in the log.

Format

SET LOG_FILE

Command Qualifiers	Defaults
/DISABLE	
/ENABLE	
/[NO]TIME_STAMP	

Command Qualifiers

/DISABLE

Indicates that terminal or command file output should not be logged to the current log file. If logging is currently disabled, then RSTS/E ignores this qualifier. This qualifier conflicts with /ENABLE.

/ENABLE

Indicates that terminal or command file output to the log file should be enabled. If logging is currently enabled, then RSTS/E ignores this qualifier. This qualifier conflicts with /DISABLE.

/[NO]TIME_STAMP

Enables or disables the time stamp feature, which prefixes each line in the log file with a date/time stamp in a format that the system manager has defined.

Appendix A

Sample Command Procedures

The following sample command procedures demonstrate some of the concepts and techniques that this manual discusses.

In the following example, the command procedure remembers the most recent file you edited, and automatically places it in the editor when you type the EDIT command:

```
$ ! EDIT.COM - Command file to "remember" last file edited
$ !
$ ! Parameters:
$ !       P1 - file to edit, or null to edit last file edited
$ !
$ ! To take full advantage of this command file, add the
$ ! following global command to your LOGIN.COM file
$ !
$ !         $ EDIT == "@EDIT"
$ !
$ ! Then, you edit a file in the usual way:
$ !
$ !         $ EDIT filespec
$ !
$ ! To edit the file you last edited, simply type:
$ !
$ !         $ EDIT
$ !
$ ! The last file edited is maintained in the global
$ ! symbol LAST_EDIT.

$           _IF F$TYPE(LAST_EDIT) .EQS. "UNDEFINED" THEN LAST_EDIT == ""
$           _IF F$LENGTH(P1) .EQ. 0 THEN P1 = LAST_EDIT

$GETFIL:
$           _IF F$LENGTH(P1) .EQ. 0 THEN _INQUIRE/EXIT=GETFIL P1 "File"
$           _IF F$LENGTH(P1) .EQ. 0 THEN _GOTO GETFIL

$           LAST_EDIT == P1
```

```
$           _SET NODATA
$           _WRITE 0 "Editing file ",LAST_EDIT,"..."
$           _EDIT 'LAST_EDIT'
$           _SET DATA
```

In the following example, the command procedure searches for and reads files specified with wildcards:

```
$ ! READ.COM - Command file to EDIT/READ wildcard file-specs
$ !
$ ! Parameters:
$ !       P1 - wildcard filespec (prompt if null)
$ !

$           _SET NODATA
$           _IF P1 .EQS. "" THEN INQUIRE/EXIT=END P1 "Filespec to read"
$           P1 = F$EDIT(P1,-1)
$           NEXT_FILE = F$SEARCH(P1)

$LOOP:
$           _IF NEXT_FILE .EQS. "" THEN _EXIT
$           _INQUIRE/EXIT=END P1 "Read ''NEXT_FILE' <yes>"
$           _IF P1 .EQS. "" THEN P1 = "YES"
$           P1 = F$EDIT(P1,-1)
$           _IF F$INSTR(1,"YES",P1) .EQ. 1 THEN _EDIT/READ 'NEXT_FILE'
$           NEXT_FILE = F$SEARCH()
$           _GOTO LOOP

$END:
```

In the following example, the command procedure remembers command arguments:

```
$!+
$! MEMCOM - Remember previous arguments for commands
$!
$! Parameters:
$!        P1  Name of command to use
$!        P2 - P8  Arguments of command to be remembered
$!
$! Description:
$!        This procedure remembers its arguments and allows them
$!        to be used again if desired. It also understands wild cards
$!        if they are the first argument to the command (i.e. P2).
$!
$!        Parameter P1 is the name of a DCL or CCL command such as EDT.
```

```
$!        This procedure uses a DCL global variable of the form
$!        'Command'_Memory where 'Command' is the name of the command
$!        to remember the arguments for each command.
$!
$!        If P2 is a wildcard filespec then MEMCOM will find the first
$!        (or next) occurrence and prompt with the file found as the
$!        default.  To accept the file type RETURN.  To see the next
$!        file type CONTROL/Z. To quit type CONTROL/C.
$!
$! Examples:
$!        $ E-DT :== @MEMCOM EDT
$!        $ EDT *.COM
$! or
$!        $ TY-PE := @MEMCOM TYPE
$!        $ TYPE *.B?S
$!-
$START:
$         WILD = 0 ! Initialize wild flag
$
$! Set command name. If no command is present then write an error
$! message (which will trigger default ON ERROR THEN EXIT action).
$
$         COMMAND = P1
$         _IF COMMAND .EQS. "" THEN WRITE 0 "?MEMCOM - No command present"
$
$! If command memory is undefined then set it to ""
$
$         _IF F$TYPE('COMMAND'_MEMORY) .EQS. "UNDEFINED" THEN -
            'COMMAND'_MEMORY == ""
$
$! Assume second argument is a file name and save it.  If a file name
$! was specified then go process it.
$
$         FILE = P2
$         _IF FILE .NES. "" THEN _GOTO PARSE_FILE
$
$! No file name was given.  If something was saved in command memory
$! then ask if it is to be used again; otherwise, just ask for
$!        a file name.
$
$         MEMORY = 'COMMAND'_MEMORY
$         _IF MEMORY .EQS. "" THEN _GOTO ASK
$         _INQUIRE/EXIT=NEXT FILE "FILE <''MEMORY'>"
$         _IF FILE .EQS. "" THEN FILE = MEMORY
$         _GOTO PARSE_FILE
$
$! Ask for a file spec if none was given on the command line and none
$! was saved in command memory
$
$ ASK:
$         _INQUIRE/EXIT=EXIT FILE "FILE"
$         _IF FILE .EQS. "" THEN _GOTO ASK
```

```
$
$! This section parses the file spec. It separates any switches,
$! constructs an argument list, and checks the file spec for wildcards.
$! After the argument list is constructed it is saved in the
$! command memory.
$
$ PARSE_FILE:
$         POS = F$INSTR(1,FILE,"/")
$         SWITCH = ""
$         _IF POS .NE. 0 THEN SWITCH = F$RIGHT(FILE,POS)
$         FILE = F$PARSE(FILE)
$         ARGS := 'FILE' 'SWITCH' 'P3' 'P4' 'P5' 'P6' 'P7' 'P8'
$         _IF ARGS .NES. "" THEN 'COMMAND'_MEMORY == ARGS
$         WILD = (F$PARSE(FILE,,"FLAGS") .AND. 870) .NE. 0
$         _IF .NOT. WILD THEN _GOTO DOIT
$         NEXT_FILE = F$SEARCH(FILE)
$
$! If the file spec was wildcard then ask for confirmation of any
$! file that matches the wild card.  If the user accepted the default
$! then construct the arg list with the next file.
$
$ ASK_WILD:
$         _INQUIRE/EXIT=NEXT FILE "FILE <''NEXT_FILE'>"
$         _IF FILE .NES. "" THEN _GOTO PARSE_FILE
$         FILE = NEXT_FILE
$         ARGS := 'FILE' 'SWITCH' 'P3' 'P4' 'P5' 'P6' 'P7' 'P8'
$
$! At last. Invoke the command on the constructed arg list
$
$ DOIT:
$         _SET NODATA
$         'COMMAND' 'ARGS'
$         _SET DATA
$
$! If it was a wildcard file spec then get the next file and
$! process it too.
$
$ NEXT:
$         NEXT_FILE = ""
$         _IF WILD THEN NEXT_FILE = F$SEARCH()
$         _IF NEXT_FILE .NES. "" THEN _GOTO ASK_WILD
$
$! All done. Clean up and exit
$
$ EXIT:
$         VERIFY = F$VERIFY(VERIFY)
$         _EXIT
```

Appendix B

RSTS/E and VAX/VMS Command Processor Differences

Chapter 2 notes that it is desirable, where possible, to provide compatibility between RSTS/E and VAX/VMS. This helps customers who use both operating systems, as well as those who will migrate to VAX systems in the future. Since RSTS/E and VMS have many inherent differences, complete compatibility between the two systems is neither desirable nor possible.

This appendix lists some of the major differences between the RSTS/E and VMS command processors:

- Character substring replacement -- RSTS/E does not support the VMS substring assignment feature:

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

- Arithmetic overlays -- RSTS/E does not support the VMS binary overlay feature:

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

- Exit status values -- RSTS/E recognizes only the following exit status values:

  ```
  0 (Warning)
  1 (Success)
  2 (Error)
  4 (Severe error)
  ```

  Note that RSTS/E does not support the VMS exit status value 3 (information).

- Symbol substitution -- RSTS/E only supports the apostrophe character as a substitution operator. RSTS/E does not support the VMS ampersand substitution operator. Users who need the 'multi-level' substitution feature provided by the ampersand operator can issue multiple assignments to produce the same result.

- Symbol name abbreviations -- In order to maintain compatibility with other commands, RSTS/E uses the hyphen (-) character to identify a symbol name's minimum abbreviation. RSTS/E also accepts the VMS abbreviation character (*).

- RSTS/E accepts abbreviations for all command synonyms and functions.

- String operators -- RSTS/E only supports the string concatenation operator, not the string reduction operator.

- Symbol tables -- RSTS/E does not define local symbols at the interactive level. If a local symbol assignment is attempted at the interactive level, DCL places the symbol in the global symbol table. If a command that accepts the /GLOBAL and /LOCAL qualifiers is issued with /LOCAL at the interactive level, DCL issues an error message.

- Symbol table searching -- VMS performs symbol searching by first searching the local symbol table at the current command level, then searching each higher command level's local symbol table, and then finally searching the global symbol table. RSTS/E first searches the local symbol table at the current command level, then the global symbol table. No higher-level local symbol tables are searched.

- VMS String Functions -- Since many RSTS/E users make use of BASIC-PLUS as a programming language, the DCL string handling functions on RSTS/E are compatible with BASIC-PLUS string functions. Use of the VMS string functions would cause unnecessary confusion for RSTS/E users, since the VMS functions use (0-based) offset arguments, rather than (1-based) position arguments.

  The following list outlines the RSTS/E string handling functions and the corresponding VMS function:

RSTS/E Function	VMS Function
F$ASCII	(none)
F$CHR	(none)
F$CVTIME	F$CVTIME
F$EDIT	(none)
F$INSTR	F$LOCATE
F$LEFT	(none)
F$LENGTH	F$LENGTH
F$MID	F$EXTRACT
F$RIGHT	(none)

- DCL Files -- VMS uses a logical name as the means for referencing an open file with the READ, WRITE and CLOSE commands. Furthermore, VMS makes use of standard logicals to refer to the input, command and output devices (SYS$INPUT, SYS$COMMAND, and SYS$OUTPUT). RSTS/E uses a channel number to reference files opened with the OPEN command. This technique lends itself more naturally to the file access mechanisms used in BASIC-PLUS.

Appendix C

RSTS/E Error Messages

Table C-1 lists RSTS/E error messages and their corresponding numeric values.

Table C-1: RSTS/E Error Messages

Decimal Value	Full Error Text
1	??Bad directory for device
2	?Illegal file name
3	?Account or device in use
4	?No room for user on device
5	?Can't find file or account
6	?Not a valid device
7	?I/O channel already open
8	?Device not available
9	?I/O channel not open
10	?Protection violation
11	?End of file on device
12	??Fatal system I/O failure
13	?Data error on device
14	?Device hung or write locked
15	?Keyboard wait exhausted
16	?Name or account now exists
17	?Too many open files on unit
18	?Illegal SYS () usage
19	?Disk block is interlocked
20	?Pack IDs don't match
21	?Disk pack is not mounted
22	?Disk pack is locked out
23	?Illegal cluster size
24	?Disk pack is private
25	%Disk pack needs REBUILDing

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
26	??Disk pack mount error
27	?I/O to detached keyboard
28	Programmable ^C trap
29	??Unused error message 29
30	?Device not file-structured
31	?Illegal byte count for I/O
32	?No buffer space available
33	??Odd address trap
34	??Reserved instruction trap
35	??Memory management trap
36	??SP stack overflow
37	??Disk error during swap
38	??Memory parity failure
39	?Maptape select error
40	?Maptape record length error
41	??Non-res run-time system
42	?Virtual buffer too large
43	?Virtual array not on disk
44	?Matrix or array too big
45	?Virtual array not yet open
46	?Illegal I/O Channel
47	?Line too long
48	%Floating point error
49	%Argument too large in EXP
50	%Data format error
51	%Integer error
52	?Illegal number
53	%Illegal argument in LOG
54	%Imaginary square roots
55	?Subscript out of range
56	?Can't invert matrix
57	?Out of data
58	?ON statement out of range
59	?Not enough data in record
60	?Integer overflow, FOR loop
61	%Division by 0
62	?No run-time system
63	?FIELD overflows buffer
64	?Not a random access device
65	?Illegal MAGTAPE( ) usage
66	?Missing special feature
67	?Illegal switch usage
68	?End of volume
69	?Quota exceeded
70	??Unused error message 70

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
71	?Statement not found
72	?RETURN without GOSUB
73	?FNEND without function call
74	?Undefined function called
75	?Illegal symbol
76	?Illegal verb
77	?Illegal expression
78	?Illegal mode mixing
79	?Illegal IF statement
80	?Illegal conditional clause
81	?Illegal function name
82	?Illegal dummy variable
83	?Illegal FN redefinition
84	?Illegal line number(s)
85	?Modifier error
86	??Unused error message 86
87	?Expression too complicated
88	?Arguments don't match
89	?Too many arguments
90	%Inconsistent function usage
91	?Illegal DEF nesting
92	?FOR without NEXT
93	?NEXT without FOR
94	?DEF without FNEND
95	?FNEND without DEF
96	?Literal string needed
97	?Too few arguments
98	?Syntax error
99	?String is needed
100	?Number is needed
101	?Data type error
102	?1 or 2 dimensions only
103	??Program lost-Sorry
104	?RESUME and no error
105	?Redimensioned array
106	%Inconsistent subscript use
107	?ON statement needs GOTO
108	?End of statement not seen
109	?What?
110	?Bad line number pair
111	?Not enough available memory
112	?Execute only file
113	?Please use the RUN command
114	?Can't CONTinue
115	?File exists-RENAME/REPLACE

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
116	?PRINT-USING format error
117	?Matrix or array without DIM
118	?Bad number in PRINT-USING
119	?Illegal in immediate mode
120	?PRINT-USING buffer overflow
121	?Illegal statement
122	?Illegal FIELD variable
123	Stop
124	?Matrix dimension error
125	?Wrong math package
126	??Maximum memory exceeded
127	%SCALE factor interlock
128	?Tape records not ANSI
129	?Tape BOT detected
130	?Key not changeable
131	?No current record
132	?Record has been deleted
133	?Illegal usage for device
134	?Duplicate key detected
135	?Illegal usage
136	?Illegal or illogical access
137	?Illegal key attributes
138	?File is locked
139	?Invalid file options
140	?Index not initialized
141	?Illegal operation
142	?Illegal record on file
143	?Bad record identifier
144	?Invalid key of reference
145	?Key size too large
146	?Tape not ansi labeled
147	?RECORD number exceeds max
148	?Bad RECORDSIZE on OPEN
149	?Not at end of file
150	?No primary key specified
151	?Key field beyond record end
152	?Illogical record accessing
153	?Record already exists
154	?Record/bucket locked
155	?Record not found
156	?Size of record invalid
157	?Record on file too big
158	?Primary key out of sequence
159	?Key larger than record
160	?File attributes not matched

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
161	?Move overflows buffer
162	?Cannot open file
163	?No file name
164	?Terminal fmt file required
165	?Cannot position to EOF
166	?Negative fill or string len
167	?Illegal record format
168	?Illegal ALLOW clause
169	??Unused ERROR message 169
170	?Indexed not fully optimized
171	?RRV not fully updated
172	?Record LOCK failed
173	?Invalid RFA field
174	?Unexpired file date
175	?Node name error
176	?Negative TAB not allowed
177	?Too much data in record
178	?OPEN Error - file corrupted
179	??Unused ERROR message 179
180	?No support for op in task
181	%Decimal overflow
182	?Network operation rejected
183	?REMAP overflows buffer
184	?Unaligned REMAP variable
185	%RECORDSIZE overflows MAP
186	?Improper error handling
187	?Illegal record lock clause
188	??Unused ERROR message 188
189	??Unused ERROR message 189
190	??Unused ERROR message 190
191	??Unused ERROR message 191
192	??Unused ERROR message 192
193	??Unused ERROR message 193
194	??Unused ERROR message 194
195	??Unused ERROR message 195
196	??Unused ERROR message 196
197	??Unused ERROR message 197
198	??Unused ERROR message 198
199	??Unused ERROR message 199
200	??Unused ERROR message 200
201	??Unused ERROR message 201
202	??Unused ERROR message 202
203	??Unused ERROR message 203
204	??Unused ERROR message 204
205	??Unused ERROR message 205

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
206	??Unused ERROR message 206
207	??Unused ERROR message 207
208	??Unused ERROR message 208
209	??Unused ERROR message 209
210	??Unused ERROR message 210
211	??Unused ERROR message 211
212	??Unused ERROR message 212
213	??Unused ERROR message 213
214	??Unused ERROR message 214
215	??Unused ERROR message 215
216	??Unused ERROR message 216
217	??Unused ERROR message 217
218	??Unused ERROR message 218
219	??Unused ERROR message 219
220	??Unused ERROR message 220
221	??Unused ERROR message 221
222	??Unused ERROR message 222
223	??Unused ERROR message 223
224	??Unused ERROR message 224
225	??Unused ERROR message 225
226	??Unused ERROR message 226
227	?String too long
228	?RECORDTYPES not matched
229	??Unused ERROR message 229
230	?No fields in image
231	?Illegal string image
232	?Null image
233	?Illegal numeric image
234	?Numeric image for string
235	?String image for numeric
236	?TIME limit exceeded
237	?1st arg to SEQ$ > 2nd
238	?Arrays must be same dim
239	?Arrays must be square
240	?Cannot change array dims
241	?Floating overflow
242	?Floating underflow
243	?CHAIN to non-existent line
244	?Exponentiation error
245	?Illegal exit from DEF*
246	?Error trap needs RESUME
247	?Illegal RESUME to SUBR
248	?Illegal subroutine return
249	?Argument out of bounds
250	?Not implemented

Table C-1: RSTS/E Error Messages (Cont.)

Decimal Value	Full Error Text
251	?Recursive subroutine call
252	?FILE ACP failure
253	?Directive error
254	??Unused ERROR message 254
255	??Unused ERROR message 255

Appendix D

ASCII Character Codes

Table D-1 lists all ASCII character codes.

Table D-1: ASCII Character Codes

ASCII			
Decimal	Octal	Character	Remarks
0	000	NUL	Null, FILL character
1	001	SOH	CTRL/A
2	002	STX	CTRL/B
3	003	ETX	CTRL/C
4	004	EOT	End of transmission, CTRL/D
5	005	ENQ	CTRL/E
6	006	ACK	CTRL/F
7	007	BEL	Bell, CTRL/G
8	010	BS	Backspace, CTRL/H
9	011	HT	Horizontal tab, CTRL/I
10	012	LF	Line feed, CTRL/J
11	013	VT	Vertical tab, CTRL/K
12	014	FF	Form feed, page, CTRL/L
13	015	CR	Carriage return, CTRL/M
14	016	SO	CTRL/N
15	017	SI	CTRL/O
16	020	DLE	CTRL/P
17	021	DC1	CTRL/Q*, XON
18	022	DC2	CTRL/R
19	023	DC3	CTRL/S**, XOFF
20	024	DC4	CTRL/T
21	025	NAK	CTRL/U
22	026	SYN	CTRL/V
23	027	ETB	CTRL/W

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
24	030	CAN	CTRL/X
25	031	EM	CTRL/Y
26	032	SUB	CTRL/Z, end of file
27	033	ESC	Escape***
28	034	FS	File Separator
29	035	GS	Group Separator
30	036	RS	Record Separator
31	037	US	Unit Separator
32	040	SP	Space or blank
33	041	!	Exclamation point
34	042	"	Quotation mark
35	043	#	Number sign
36	044	$	Dollar sign
37	045	%	Percent sign
38	046	&	Ampersand
39	047	'	Apostrophe
40	050	(	Left parenthesis
41	051	)	Right parenthesis
42	052	*	Asterisk
43	053	+	Plus
44	054	,	Comma
45	055	-	Hyphen or minus
46	056	.	Period or decimal point
47	057	/	Slash
48	060	0	Zero
49	061	1	One
50	062	2	Two
51	063	3	Three
52	064	4	Four
53	065	5	Five
54	066	6	Six
55	067	7	Seven
56	070	8	Eight
57	071	9	Nine
58	072	:	Colon
59	073	;	Semicolon
60	074	<	Left angle bracket, "less than" sign
61	075	=	Equal sign
62	076	>	Right angle bracket, "greater than" sign
63	077	?	Question mark
64	100	@	At sign

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
65	101	A	Uppercase A
66	102	B	Uppercase B
67	103	C	Uppercase C
68	104	D	Uppercase D
69	105	E	Uppercase E
70	106	F	Uppercase F
71	107	G	Uppercase G
72	110	H	Uppercase H
73	111	I	Uppercase I
74	112	J	Uppercase J
75	113	K	Uppercase K
76	114	L	Uppercase L
77	115	M	Uppercase M
78	116	N	Uppercase N
79	117	O	Uppercase O
80	120	P	Uppercase P
81	121	Q	Uppercase Q
82	122	R	Uppercase R
83	123	S	Uppercase S
84	124	T	Uppercase T
85	125	U	Uppercase U
86	126	V	Uppercase V
87	127	W	Uppercase W
88	130	X	Uppercase X
89	131	Y	Uppercase Y
90	132	Z	Uppercase Z
91	133	[	Left square bracket
92	134	\	Backslash
93	135	]	Right square bracket
94	136	^	Circumflex
95	137	_	Underscore
96	140	`	Grave accent
97	141	a	Lowercase a
98	142	b	Lowercase b
99	143	c	Lowercase c
100	144	d	Lowercase d
101	145	e	Lowercase e
102	146	f	Lowercase f
103	147	g	Lowercase g
104	150	h	Lowercase h
105	151	i	Lowercase i
106	152	j	Lowercase j
107	153	k	Lowercase k

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
108	154	l	Lowercase l
109	155	m	Lowercase m
110	156	n	Lowercase n
111	157	o	Lowercase o
112	160	p	Lowercase p
113	161	q	Lowercase q
114	162	r	Lowercase r
115	163	s	Lowercase s
116	164	t	Lowercase t
117	165	u	Lowercase u
118	166	v	Lowercase v
119	167	w	Lowercase w
120	170	x	Lowercase x
121	171	y	Lowercase y
122	172	z	Lowercase z
123	173	{	Left brace
124	174	|	Vertical line
125	175	}	Right brace ***
126	176	~	Tilde ***
127	177	DEL	Delete
128	200		Reserved
129	201		Reserved
130	202		Reserved
131	203		Reserved
132	204	IND	Index
133	205	NEL	New line
134	206	SSA	
135	207	ESA	
136	210	HTS	Horizontal tab set
137	211	HTJ	
138	212	VTS	Vertical tab set
139	213	PLD	Partial line down
140	214	PLU	Partial line up
141	215	RI	Reverse Index
142	216	SS2	Single shift 2
143	217	SS3	Single shift 3
144	220	DCS	Device control string
145	221	PU1	
146	222	PU2	
147	223	STS	
148	224	CCH	
149	225	MW	
150	226	SPA	

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
151	227	EPA	
152	230		Reserved
153	231		Reserved
154	232		Reserved
155	233	CSI	Control sequence introducer
156	234	ST	String terminator
157	235	OSC	
158	236	PM	
159	237	APC	
160	240		Reserved
161	241	¡	Inverted exclamation point
162	242	¢	Cent sign
163	243	£	Pound sign
164	244		Reserved
165	245	¥	Yen sign
166	246		Reserved
167	247	§	Section sign
168	250	¤	General currency sign
169	251	©	Copyright sign
170	252	ª	Feminine ordinal indicator
171	253	«	Angle quotation mark left
172	254		Reserved
173	255		Reserved
174	256		Reserved
175	257		Reserved
176	260	°	Degree sign
177	261	±	Plus/minus sign
178	262	²	Superscript 2
179	263	³	Superscript 3
180	264		Reserved
181	265	µ	Micro sign
182	266	¶	Paragraph sign, pilcrow
183	267	·	Middle dot
184	270		Reserved
185	271	¹	Superscript 1
186	272	º	Masculine ordinal indicator
187	273	»	Angle quotation mark right
188	274	¼	Fraction one quarter
189	275	½	Fraction one half
190	276		Reserved
191	277	¿	Inverted question mark
192	300	À	Uppercase A with grave accent
193	301	Á	Uppercase A with acute accent

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
194	302	Â	Uppercase A with circumflex accent
195	303	Ã	Uppercase A with tilde
196	304	Ä	Uppercase A with diaeresis or umlaut mark
197	305	Å	Uppercase A with ring
198	306	Æ	Uppercase A with dipthong
199	307	Ç	Uppercase C with cedilla
200	310	È	Uppercase E with grave accent
201	311	É	Uppercase E with acute accent
202	312	Ê	Uppercase E with circumflex accent
203	313	Ë	Uppercase E with diaeresis or umlaut mark
204	314	Ì	Uppercase I with grave accent
205	315	Í	Uppercase I with acute accent
206	316	Î	Uppercase I with circumflex accent
207	317	Ï	Uppercase I with diaeresis or umlaut mark
208	320		Reserved
209	321	Ñ	Uppercase N with tilde
210	322	Ò	Uppercase O with grave accent
211	323	Ó	Uppercase O with acute accent
212	324	Ô	Uppercase O with circumflex accent
213	325	Õ	Uppercase O with tilde
214	326	Ö	Uppercase O with diaeresis or umlaut mark
215	327	Œ	Uppercase OE ligature
216	330	Ø	Uppercase O with slash
217	331	Ù	Uppercase U with grave accent
218	332	Ú	Uppercase U with acute accent
219	333	Û	Uppercase U with circumflex accent
220	334	Ü	Uppercase U with diaeresis or umlaut mark
221	335	Ÿ	Uppercase Y with diaeresis or umlaut mark
222	336		Reserved
223	337	ß	German lowercase sharp s
224	340	à	Lowercase a with grave accent
225	341	á	Lowercase a with acute accent
226	342	â	Lowercase a with circumflex accent
227	343	ã	Lowercase a with tilde
228	344	ä	Lowercase a with diaeresis or umlaut mark
229	345	å	Lowercase a with ring

Table D-1: ASCII Character Codes (Cont.)

ASCII			
Decimal	Octal	Character	Remarks
230	346	æ	Lowercase ae dipthong
231	347	ç	Lowercase c with cedilla
232	350	è	Lowercase e with grave accent
233	351	é	Lowercase e with acute accent
234	352	ê	Lowercase e with circumflex accent
235	353	ë	Lowercase e with diaeresis or umlaut mark
236	354	ì	Lowercase i with grave accent
237	355	í	Lowercase i with acute accent
238	356	î	Lowercase i with circumflex accent
239	357	ï	Lowercase i with diaeresis or umlaut mark
240	360		Reserved
241	361	ñ	Lowercase n with tilde
242	362	ò	Lowercase o with grave accent
243	363	ó	Lowercase o with acute accent
244	364	ô	Lowercase o with circumflex accent
245	365	õ	Lowercase o with tilde
246	366	ö	Lowercase o with diaeresis or umlaut mark
247	367	œ	Lowercase oe ligature
248	370	ø	Lowercase o with slash
249	371	ù	Lowercase u with grave accent
250	372	ú	Lowercase u with acute accent
251	373	û	Lowercase u with circumflex accent
252	374	ü	Lowercase u with diaeresis or umlaut mark
253	375	ÿ	Lowercase y with diaeresis or umlaut mark
254	376		Reserved
255	377		Reserved

* CTRL/Q, or XON, resumes output if the TTSYNC terminal characteristic is set.

** CTRL/S, or XOFF, stops output if the TTSYNC terminal characteristic is set.

*** ALTMODE(ASCII 125) or PREFIX (ASCII 126) keys, which appear on some terminals, are translated internally into ESCAPE if the ALT MODE terminal characteristic is set.