

VAX/VMS

Writing Real Programs in DCL



digital

Paul C. Anagnostopoulos

VAX/VMS: Writing Real Programs in DCL

Digital Press VAX Users Series

Paul C. Anagnostopoulos

VAX/VMS: Writing Real Programs in DCL

Philip E. Bourne

UNIX for VMS Users

James F. Peters III and Patrick Holmay

An Introduction to VAX/VMS

Ronald M. Sawey and Troy T. Stokes

A Beginner's Guide to VAX/VMS Utilities and Applications

VAX/VMS:

Writing Real Programs in DCL

Paul C. Anagnostopoulos

digitalTM

Digital Press

Copyright © 1989 by Digital Equipment Corporation

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

9 8 7 6 5 4

Order number EY-C168E-DP

Printed in the United States of America.

Trademarks and trademarked products mentioned in this book include: American Mathematical Society, TEX; Bell Laboratories, UNIX, Shell; Digital Equipment Corporation, DCL, the Digital logo, DIGITAL, DEC/CMS, DEC/MMS, DECnet, DEC/shell, VAX, VAX LISP, VAX/TPU, VAX/VMS, VAXcluster, VAXstation, VMS, VT100, VT200, VT300; International Business Machines Corporation, IBM, IBM VM/SP, REXX.

La Vie Dansante, Jimmy Buffett / Michael Utley / Will Jennings © 1984 Coral Reefer Music BMI & Coconutly Music ASCAP & Warner Tamerlane Music & Blue Sky Rider Songs BMI.

Design: Sandra Calef

Copyediting: Alice Cheyer

Index: Howard Burrows, John Mann, Rosemary Simpson

Production: Editorial Inc. (Page makeup via \TeX with PostScript output)

Printing and binding: Hamilton Printing Company

Library of Congress Cataloging-in-Publication Data

Anagnostopoulos, Paul C., 1953—

VAX/VMS—Writing real programs in DCL / Paul C. Anagnostopoulos.

p. cm. — (Digital Press VAX users series)

Includes index.

ISBN 1-55558-023-8

1. VAX/VMS (Computer operating system) 2. DCL (Computer program language)

I. Title II. Title: Writing real programs in DCL. III. Series.

QA76.76.063A48 1989 005.4'44—dc19 89-1161 CIP

Contents

Preface **xiii**

1

Introduction **1**

- 1.1 Choosing to Use DCL 2
- 1.2 The Application Domain for DCL 3
- 1.3 A Word of Caution 6
- 1.4 How to Use This Book 6

2

A Review of VMS Concepts **10**

- 2.1 An Interactive Session 10
- 2.2 The DCL Command Interpreter 11
- 2.3 DCL Commands 11
- 2.4 Files and Directories 14
- 2.5 Logical Names 15
- 2.6 User Identification Codes 16
- 2.7 Privileges 17
- 2.8 DCL Procedures 17
- 2.9 Procedure Verification 20
- 2.10 The Login Procedure 20

3	Symbols, Data, and Expressions	21
3.1	Symbols	21
3.2	Types of Data	22
3.3	Assignment Commands	25
3.4	Expressions	31
3.5	Lexical Functions	36
3.6	Substring Assignment	42
3.7	Bit-Field Assignment	44
3.8	Summary of Assignment Commands	47
3.9	Deleting Symbols	47
4	Flow of Control	49
4.1	Sequential Execution	49
4.2	The GOTO Command	50
4.3	The IF Command	52
4.4	Loops	56
4.5	Invoking a Command Procedure	58
4.6	The CALL Command	62
4.7	The GOSUB Command	64
5	Substitution	66
5.1	Apostrophe Substitution	66
5.2	Implicit Substitution	70
5.3	Ampersand Substitution	73
6	Displaying Output	75
6.1	Displaying Text	75
6.2	Terminal Control Sequences	76
6.3	Formatting Output	78
6.4	Redirecting Program Output	79
6.5	Displaying a File	81
7	The DCL Environment	82
7.1	Aspects of the Environment	83
7.2	Putting It All Together	88

8	Error Handling	90
	8.1 Status Codes	90
	8.2 The Default Error Handler	94
	8.3 Changing Error-Handling Behavior	95
	8.4 Checking a Status Code Explicitly	96
	8.5 Handling Errors When Cleanup Is Required	97
	8.6 Procedure Call Unwinding	98
	8.7 Ignoring an Error	100
	8.8 Obtaining the Message for a Status Code	101
9	Interrupt Handling	104
	9.1 Interrupt Situations	104
	9.2 The Default Interrupt Handler	105
	9.3 Changing Interrupt Handler Behavior	106
	9.4 Using Interrupts for Other Purposes	109
	9.5 Tying It All Together	110
10	Application Structure	111
	10.1 Naming Conventions	111
	10.2 Invoking the Application	112
	10.3 The Main Procedure	113
	10.4 A Subprocedure	116
	10.5 The Subroutine Library	118
	10.6 The SIGNAL Subroutine	120
	10.7 Captive Accounts	122
11	Obtaining User Input	123
	11.1 The INQUIRE Command	124
	11.2 The READ Command	125
	11.3 Default Answers	126
	11.4 Editing Input	127
	11.5 Checking the Validity of Input	128
	11.6 Main Procedure Arguments	131
	11.7 A Sophisticated Input Subroutine	132
	11.8 Redirecting Program Input	133
	11.9 Displaying Large Amounts of Text	134
12	Debugging	136
	12.1 Verification	136
	12.2 Temporary Debugging Output	140
	12.3 Permanent Debugging Output	140
	12.4 Capturing Procedure Output	141

13	Files and Directories	143
	13.1 File Specifications	143
	13.2 Parsing	148
	13.3 Searching Directories	151
	13.4 File Protection	156
	13.5 File Attributes	157
	13.6 File Operations	158
14	Logical Names	161
	14.1 Defining and Using Logical Names	162
	14.2 Complete File Spec Parsing Algorithm	165
	14.3 Displaying Logical Names	166
	14.4 Access Modes	167
	14.5 Standard Logical Names	169
	14.6 Other Uses for Logical Names	169
	14.7 Obtaining Logical Name Information	174
	14.8 Deleting Logical Names	176
	14.9 Overview of VMS Directory Organization	177
	14.10 Overview of Logical Name Tables	179
15	Sequential File Operations	183
	15.1 Reading an Existing File	184
	15.2 Creating and Writing a New File	187
	15.3 Appending to an Existing File	189
	15.4 Temporary Files	190
	15.5 Displaying a File	192
	15.6 Searching Files	193
16	Data Manipulation	196
	16.1 Arithmetic Techniques	196
	16.2 Lists	198
	16.3 Keywords	201
	16.4 Dates and Times	202
	16.5 Record Structures	205

17	Indexed File Operations	210
	17.1 Sample File	212
	17.2 Reading an Existing File	213
	17.3 Creating a File	216
	17.4 Writing a File	219
	17.5 Updating a File	220
	17.6 Deleting Records	220
	17.7 File Sharing	221
18	File Protection	225
	18.1 UIC-Based Protection	226
	18.2 Access Control Lists	230
	18.3 Default File Protection	237
	18.4 The Access Algorithm	239
19	Devices	241
	19.1 Device Names	241
	19.2 Device Information	242
	19.3 Obtaining a List of the Devices on a System	244
	19.4 Terminal Characteristics	246
	19.5 Device Allocation	247
	19.6 Volume Initialization	249
	19.7 Volume Mounting	250
20	Processes	253
	20.1 Information about the Current Process	254
	20.2 Information about Other Processes	257
	20.3 Setting Process Information	259
	20.4 Subprocesses	260
	20.5 Process-Permanent Files	265
	20.6 Jobs	266
	20.7 An Example with SPAWN	267
21	Batch Jobs	270
	21.1 The Batch Environment	270
	21.2 Submitting a Batch Job	271
	21.3 Sending Mail about Job Status	274
	21.4 Restartable Batch Jobs	276
	21.5 Periodic Batch Jobs	279
	21.6 Preventing Duplicate Jobs	282

22	Arrays	284
	22.1 Arrays with Integer Indexes	284
	22.2 Arrays with String Indexes	288
	22.3 The DCL Symbol Table	290
	Appendixes	
A	Hexadecimal Notation	291
B	DEC Multinational Character Set	293
C	Subroutine Library	296
D	Sample Application	304
	D.1 Using MODL	305
	D.2 MODL Commands	306
	D.3 File Layout	313
	D.4 Listings	315
	Glossary	352
	Index	367

List of Tables

3.1	<i>DCL's Interpretation of Data as Boolean Values</i>	25
3.2	<i>Integer Operators</i>	32
3.3	<i>Character String Operators</i>	33
3.4	<i>Boolean Operators</i>	34
3.5	<i>Operator Precedence</i>	35
3.6	<i>Summary of Assignment Commands</i>	47
4.1	<i>Summary of Subroutine Facilities</i>	64
6.1	<i>F\$FAO Directives</i>	78
8.1	<i>Status Code Fields</i>	91
8.2	<i>Status Code Severity Levels</i>	92
8.3	<i>VMS Message Files</i>	102
13.1	<i>F\$FILE_ATTRIBUTES Items</i>	157
13.2	<i>VMS File Utilities</i>	159
13.3	<i>File Selection Qualifiers</i>	160
14.1	<i>Defining Logical Names in Different Modes</i>	168
14.2	<i>Standard Process and Job Logical Names</i>	169
14.3	<i>F\$TRNLNM Items</i>	176
14.4	<i>System Logical Names</i>	178

15.1	How /MATCH Affects Searching	194
16.1	Fields in a Record Structure	206
17.1	Fields in the Sample Indexed File	212
18.1	Access Needed for Directory Operations	227
18.2	Access Needed for File Operations	228
18.3	SET ACL Command Qualifiers	235
19.1	F\$GETDVI Items	243
19.2	MOUNT Command Qualifiers	251
20.1	F\$GETJPI Items	255
20.2	F\$ENVIRONMENT Items	256
20.3	Commands to Set Process Environment	259
20.4	SPAWN Command Qualifiers	261
20.5	Standard PPFs: Interactive Procedure	266
20.6	Standard PPFs: Batch Procedure	267
21.1	SUBMIT Command Qualifiers (Part 1)	272
21.2	SUBMIT Command Qualifiers (Part 2)	273
B.1	Standard ASCII Character Set	294
B.2	DEC Multinational Extension	295
D.1	MODL File: Header Record	314
D.2	MODL File: Keyword Record	314
D.3	MODL File: User Record	314

Preface

During 1983, while working in the VMS Development Group at Digital Equipment Corporation, I was given the task of developing a new software installation procedure for VMS. The purpose of this procedure was to provide a uniform environment and methodology for installing software products on a VMS system. There were no preconceived notions about how this procedure should be structured, other than its required name: `VMSINSTAL`. I was annoyed by the missing “L” in the name, but I nevertheless accepted the task with enthusiasm. After giving the problem some careful thought, I decided to implement the required software in the Digital Command Language, DCL.

My decision raised a few eyebrows. Some people said, “It’s too big and complicated, DCL will just get in your way.” Others said, “DCL isn’t a programming language!” I was convinced it was the correct decision, however, given the requirements of the project. Most of the actions taken by an installation procedure involve the manipulation of files: restoring from backup, copying, renaming, and so on. DCL is certainly a language that makes file manipulation easy. `VMSINSTAL` has to interact with a program written by the developers of the product being installed. The product’s program drives the installation process in conjunction with `VMSINSTAL`. DCL makes this kind of interaction simple: one procedure simply invokes the other, and vice versa. One need not worry about compiling or linking the programs. Finally, I felt that the user interface to `VMSINSTAL` would be simple and not require any fancy programming.

After a couple of months' work I had written a 1,300-line DCL procedure that performed software installations in a consistent and extensible fashion. In the process, I was forced to develop stylistic and organizational guidelines for programming in DCL.

Since that time I have developed many complex DCL procedures and lent a hand to other people who were developing them. I have also read many procedures and remain overwhelmed by their lack of style and organization. I have three goals in writing this book. The first is to help people make an intelligent choice between DCL and more conventional programming languages. You will only be comfortable with the decision to write an application in DCL if you understand DCL's capabilities and limitations.

My second goal is to offer a programming language to computer users who are not conversant with conventional languages. Secretaries, administrative assistants, word processing folks, database administrators, system managers: all will be more challenged and productive if they can create new programs and solve problems on their own. Never forget what Commander Grace Hopper said: It's easier to apologize later than it is to get permission.

My final goal is to make the DCL programming process enjoyable for everyone. Programming is satisfying when you understand the features of the language, know how to apply them, and have the ability to build upon previous work to meet new situations. The entire software development process is more rewarding when the resulting software is correct and the programs are easy to understand.

If you are going to write an application in DCL, you might as well take it seriously. Then you can have fun.

Paul C. Anagnostopoulos
March 1989

Acknowledgments

Very special appreciation goes to my wife, Cynthia L. Sorn, for her constant support and patience during the past year. She read all the chapters and gave me suggestions and criticisms that directed the book toward a broad audience with a wide range of background knowledge. The project was a lot more fun with her participation. She spurred me on when my enthusiasm was lagging and gave me encouragement when I grumped about some difficult section. I wonder what she'll think if I start another book?

I particularly appreciate the help of the folks at UIS, Inc. in Lexington, Massachusetts. They let me use their VMS Version 5 system to develop all the examples in this book. Samir Bhatia and Charles Strauss reviewed some chapters for me.

Lots of friends at Digital Equipment Corporation reviewed part or all of the book: Ruth Goldenberg, Joel Magid, Brian Mahoney, Rich Robbins, Rachel Ross, Hal Shubin. Thanks to all of them. Extra thanks to Rich and Hal, who read the entire book twice. As Rich said when he reviewed the chapter on debugging: "Debugging can be avoided if programmers refrain from making errors."

I appreciate my friend Bill Rothman spending time on some of the introductory chapters. He was not a VMS user back then, so his comments were invaluable for clarifying fundamental concepts.

The official technical reviewers from among the VMS user community were helpful and supportive: Ted Maryan of LTV Missiles and Electronics, James F. Peters of Kansas State University; Ronald Sawey of Southwest Texas State University; Joy Veronneau of Cornell University.

Particular recognition goes to Donald E. Knuth and Leslie Lamport. This book was composed with the \TeX typesetting system running the \LaTeX macro package. It sure was fun to typeset the whole thing myself!

Additional recognition goes to everyone who helped produce the book: Howard Burrows, Sandra Calef, Alice Cheyer, Stephani Colby, Chase Duffy, Timothy Evans, Hartley Ferguson, Amy Hendrickson, John Mann, Mike Meehan, Marilyn Rash, Rosemary Simpson, Joe Snowden.

A final thank you to Jimmy Buffett, an artist whose homemade music kept me going through all those long hours staring at the terminal screen.

Miss the beat if you close your eyes
Every night wears a new disguise
And I live when a new surprise surrenders
Feel it all with a willing heart
Every stop is a place to start
If you know how to play the part with feeling
I play with feeling

That's why I wander and follow *la vie dansante* . . .

— Jimmy Buffett
Will Jennings
Michael Utley

VAX/VMS: Writing Real Programs in DCL

Chapter 1



Introduction

The Digital Command Language, DCL, is the vehicle by which the interactive user communicates with the VAX/VMS operating system. The user enters commands, which are analyzed and executed by the DCL command language interpreter to perform actions or display information. Commands can have dozens or even hundreds of variations, many of which the user must understand intimately. A command language layer is present on all interactive operating systems, manifested as the Shell on Unix or REXX on IBM VM/SP. There are other command language interpreters available for VMS, including DEC/shell, which creates an environment similar to the Unix Shell.

VMS users soon find that they want to capture a series of DCL commands in a file so they can use the same series of commands quickly and accurately in the future. In this way, the file acts as a script, specifying actions to be taken automatically when the script is called up. Often the script can be significantly more useful if its commands can be varied from one use to the next. Therefore, most command languages inevitably evolve into programming languages. In the case of VMS, DCL is the name given to both the interactive command language and the command programming language, which are really one and the same thing. A file containing a DCL program is called a **command procedure**.

The goal of this book is to help you write better DCL procedures. The book assumes a familiarity with VMS; it does not attempt to teach the operating system from the ground up. However, even if you have written only a few rudimentary

procedures, you can become skilled in the art of DCL programming by the time the last chapter is read.

1.1

Choosing to Use DCL

DCL is by no means the solution to all programming problems. However, there are programming tasks for which it is perfectly suited, and there are even some aspects of DCL that are refreshing when compared to conventional languages. It is necessary to analyze the task at hand carefully when considering DCL as the programming language. The following is a list of DCL's strong points:

- DCL is available on every VMS system.
- DCL procedures can be developed rapidly. The programmer is freed from the edit/compile/link/test cycle because DCL is an interpreted language.
- DCL procedures can be modified quickly when requirements change or expand.
- DCL procedures can employ VMS utilities and other software products in a simple and natural manner.
- Some tricky aspects of VMS programming, such as error and interrupt processing, are significantly simplified by DCL (as long as the necessary processing is straightforward).
- Unsophisticated programmers can use DCL procedures as building blocks to create sophisticated programs. So, for that matter, can sophisticated programmers.

If one or more of these points stand out as important to the application being planned, then DCL may be the programming language of choice. Nevertheless, DCL's weak points must also be considered:

- A DCL procedure runs slower than a program written in a conventional language because the procedure is executed by an interpreter. This is particularly true if the procedure is computationally intensive (i.e., if it performs many arithmetic calculations), less true if the procedure spends most of its time doing file manipulations.
- DCL is missing some important arithmetic facilities such as floating-point numbers.
- Data-structuring capabilities, important to many software algorithms, are virtually nonexistent in DCL.

- DCL is missing some of the modern structured programming constructs such as WHILE and FOR loops.
- Only the simplest user interface can be programmed in DCL. DCL has no windowing or graphics capabilities.
- A program can be invoked directly from a DCL procedure only if it has a command interface. In particular, shareable images cannot be called directly from DCL procedures.

DCL is a programming language that allows rapid development of programming of VMS utilities and software products. DCL is also a language whose original design did not foresee the wide range of applications for which it would be used, so parts of DCL are not as well-structured as other modern programming languages.

1.2 *The Application Domain for DCL*

In light of these pros and cons, DCL is an appropriate choice of implementation language for some applications and not for others. The following sections describe a few of the application domains for which it is well suited. This is in no way an exhaustive list.

1.2.1 *Environment Extension*

One of the most common uses for DCL is to extend a user's personal VMS environment. Any sequence of commands that is frequently needed can be embodied in a DCL procedure. Such a procedure reduces the amount of typing required and, if properly written, can make decisions and alter its actions based upon the current state of the environment.

Simple procedures can be used to submit batch and print jobs. It is often helpful to use a procedure to run an application consisting of two or three programs executed in sequence. A procedure can be written to replace the LOGOUT command: before logging out it can delete scratch files and purge the user's directories in order to save disk space.

A moderately sophisticated but extremely handy procedure can be written to enhance the SET DEFAULT command, which is the VMS way of establishing the current working directory. This procedure can accept commands (e.g., UP, DOWN, SIDEWAYS) that reset the working directory by moving around in the directory

tree. An additional feature can allow the user to record the working directory, switch to a new working directory, and then pop back to the original one.

Extremely complex procedure libraries have been written to add a layer of features “on top of” DCL. These procedures provide an extended set of facilities, including a calendar and a telephone directory. The calendar interacts with the VMS batch facility to submit and run batch jobs at predetermined times. The telephone directory interacts with VMS electronic mail to provide powerful distribution list capabilities. These procedure libraries even allow users to write their own procedures, assign them a command name, and integrate them into the library.

1.2.2 System Management

DCL is perfect for extending the system management facilities included in VMS, particularly since many of these facilities are missing essential features. A procedure can tie together the many different system management tools, allowing complex tasks to be performed easily. Such a procedure also reduces the chance of doing things in the wrong order or of leaving out a critical step (a mistake made in system management can leave the VAX unusable).

A simple procedure can be used to run the AUTHORIZE utility, the VMS program that is used to authorize new system users. Because AUTHORIZE has no corresponding DCL command, and because it must be invoked from a particular system directory, the utility must be invoked in a special way. This is easily accomplished by a short procedure, whose only real purpose is to save typing.

A moderately complex procedure can be used to add new system roots to a system disk. This is particularly useful in a VAXcluster environment, where each node has its own system-specific root and all nodes share a common root. A procedure to perform just this task is included with VMS. It is provided in the file MAKEROOT.COM residing in the SYS\$MANAGER directory.

A more complex procedure can be written to perform daily disk backups. It would use the BACKUP utility to create incremental backups on tape. It might also include a simple tape library facility, which would automatically label the tapes and keep track of backup cycles. This backup procedure could be extended to deal with weekly backups, monthly backups, disk restoration, and so on.

1.2.3

Complex File Manipulation

Many applications require relatively complicated file manipulation. Files must be created, copied, backed up, and reorganized. VMS provides a host of utilities to help with such operations, but many of them cannot be called from conventional programs; they have no callable interface. DCL procedures are one way to sequence these utilities and control their subtler aspects.

In a simple application, a frequently updated file may need to be archived periodically. A batch job would run every four hours and make a copy of the file, purging all but the last five versions.

A distributed application might require that certain files be copied over a network every evening. A batch job is run that copies the files from the remote node to the local node. Relatively sophisticated error handling is necessary in case the network link is lost during a file transfer. It is essential to inform the application manager of the problem and to ensure that a consistent set of files is maintained on the local node.

The VMS software installation procedure, `VMSINSTAL`, is an example of a very complex DCL procedure that essentially solves a file manipulation problem. The files making up a software product are distributed on magnetic tape. These files, along with additional files created at the customer's site, must be placed in the appropriate VMS directories. Some files, such as command definitions and help text files, must be placed in specific libraries. All of these file operations must be performed in such a way as to minimize the confusion caused by a system failure during the installation.

1.2.4

Software Development

The software development process often requires complex procedures to create, maintain, and assemble the components of the software system. Many of the common procedures are addressed by products already on the market. For example, DEC/CMS is a source code management system that keeps track of program modules and prevents two developers from altering a module in an inconsistent fashion. It is always best to consider available products before embarking on a project to create one from scratch. However, development methodologies are constantly changing and a new idea may necessitate the development of in-house tools.

DCL procedures can tie together existing development tools to improve communication between members of a development group. For example, when a

developer replaces a module after modification, electronic mail can be sent to other members of the group so that they know the module is available.

Most software systems must be “built” before they can be tested. Building a system from its myriad components is a time-consuming process that must take into account the many interdependencies between the components. A set of DCL procedures, perhaps constructed around a system-building product such as DEC/MMS, can drive the building process from start to finish. The procedures can maintain and use a description file that lists all the software components and their relations. The procedures can make use of multiple nodes in a VAXcluster or network to perform the build in parallel.

DCL procedures can act as the backbone of an automated regression test system. The purpose of a regression test system is to verify a software product from one release to the next, to ensure that everything that worked in the previous release works in the new release. A critical aspect of such systems is the maintenance of a test case library and the execution of the tests within it. The results of the tests must be carefully recorded and compared with known correct results. Either or both of these requirements can be met with a set of DCL procedures.

1.3 *A Word of Caution*

A DCL procedure often runs with a high level of system privilege, particularly if the procedure performs system management functions. Beware of accepting “black box” procedures from people you do not know, or from a public bulletin board. There have been instances of “viruses” and “worms”—procedures that have deleted system files or destroyed the system in some other fashion.

1.4 *How to Use This Book*

This book can teach you how to write DCL procedures even if you have never written one before. However, the author assumes that you have a basic knowledge of VMS concepts and facilities. Do not attempt to learn all about VMS by reading this book; choose an introductory VMS text, like *Introduction to VMS* in the VMS Documentation Set, to read instead. Chapter 2 is a review of the VMS concepts required to understand the rest of the book. If you are a novice VMS user, read it carefully; otherwise a quick review is all that is necessary. Chapters 3–5 provide a detailed description of the facilities DCL provides for data manipulation and decision making. The author recommends that everyone read these chapters, because even experienced DCL programmers will discover new features and ideas.

The remaining chapters of the book address various DCL topics in detail. They have been arranged to allow each chapter to rely almost exclusively on features introduced in previous chapters. The goal of this book is to teach you a set of techniques for building applications in DCL. This is by no means a DCL reference manual; every command and option is not described in full. When you want the complete description of a particular feature, consult the VMS Documentation Set. References to the appropriate documents are given throughout the text. A list of the most important documents is presented in Section 1.4.2.

This book presents some VMS features that are only available in VMS Version 5. Whenever there is an important difference between the Version 5 and the Version 4 systems, the difference is explicitly noted.

Program examples are best presented within the context of a particular application. Throughout the book, examples are written as if they were part of a fictitious DCL application called “Example DCL Application,” or XDA. The XDA application does not really exist, but you can pretend it does while reading the examples. This lends some coherence and consistency to the many DCL examples illustrated in the book. In order to tie together all the information presented in the book, the procedures for a real application are presented in Appendix D.

A glossary of the terms introduced in this book is provided following the appendixes.

1.4.1 **Typographic Conventions**

The following typographic conventions are used throughout this book:

Introduction of term. A new term is introduced in boldface.

Emphasis of word. A word is emphasized in italics.

Document Name. The name of a manual in the VMS Documentation Set is specified in italics.

VMS KEYWORD or NAME. A keyword or name that is provided by VMS and entered as shown is specified in an uppercase “typewriter” typeface, as in `SHOW TIME`. This command is composed of two words, which must be spelled and punctuated as shown.

<RETURN> key. The name of a key on the terminal keyboard is specified in angle brackets.

Metalinguistic *symbol*. A **metalinguistic symbol** is shown in italics. A metalinguistic symbol is a symbolic name that stands for some information to be included in a DCL command. For example: DELETE *file-spec*. The metalinguistic symbol *file-spec* stands for a file specification, which must be included in the command in place of the symbol.

DCL programs and examples of the interaction between the user and VMS are illustrated as follows:

```
$! This is a DCL procedure.  
$  
$ show time  
.  
.do other things  
.  
$ exit
```

The program is set off from the text in a smaller typeface. It uses the same “typewriter” typeface as the text but is shown in lowercase letters. A program example may include metalinguistic symbols or ellipses to indicate portions of the program that are missing from the example. In examples of interaction between the user and VMS, the user’s input is shown in color.

▷ Ch. 1

When a complicated or tricky DCL feature is being used, a reference to the chapter or appendix in which it is described is called out in the margin.

1.4.2 **VMS Documentation Set**

The definitive description of the Digital Command Language is provided in the VMS Documentation Set. Five of the manuals in the set are particularly important for the DCL programmer:

Introduction to VMS. An introduction to VMS for the novice user.

VMS DCL Concepts Manual. A thorough discussion of DCL concepts and facilities.

VMS DCL Dictionary. A detailed description of the DCL commands and facilities. This manual defers to other manuals to describe some of the more complicated commands, such as BACKUP.

Guide to Using VMS Command Procedures. A description of some of the DCL programming techniques covered in this book.

Guide to Creating VMS Modular Procedures. A summary of the VMS naming standards, which allow various DIGITAL products and user applications to coexist on a VMS system.

This book does not describe every feature of DCL, nor does it list every option of every DCL command. You must consult the *VMS DCL Dictionary* when you need the full details of any particular DCL feature. This book is not a replacement for the *VMS DCL Dictionary*.

So you want to learn a new programming language? There are only four things you need to find out:

- What does a semicolon do?
- Are vector indexes zero- or one-based?
- How is a compound statement formed?
- Does it have macros?

If the answer to the last question is no, forget it.

— Technical Folklore

Chapter 2



A Review of VMS Concepts

This chapter reviews some basic VMS concepts, which any user of VMS must understand in order to use the operating system effectively. You may already understand all the concepts presented here, but a review is certainly helpful. If you find yourself stumbling over these ideas, it is best to study them in more detail by reading the volume *Introduction to VMS* in the VMS Documentation Set. All the concepts presented in this chapter are described in greater detail in subsequent chapters of this book.

2.1

An Interactive Session

An interactive VMS session is initiated by logging in to VMS from a terminal. VMS creates an **interactive process**, which can perform work on your behalf. The process executes the programs making up the applications that you choose to run on the VAX. These programs may be supplied by DIGITAL, purchased from third-party software vendors, or written at your own site.

A VMS process has many items of information associated with it. A few of the more interesting items are

- The **user name** with which you logged in to VMS. This may be your actual name or some other identifier for your account.
- The name of the VMS process. For your first interactive session, the process name is the same as your user name.

- The name of the terminal at which you logged in. You use this terminal to tell VMS what to do, and VMS uses it to display information.

2.2 **The DCL Command Interpreter**

Once you have established an interactive session, there must be some way to tell VMS what you want to do. To instruct VMS to perform an action, you issue a **command** at the terminal. This command is carried out by a special program called a **command language interpreter**. The standard command language interpreter for VMS is called DCL, which stands for Digital Command Language. DCL accepts commands from the keyboard, analyzes them, and then performs them.

DCL has control of your terminal during an interactive session. It signals that it is ready to accept a command by displaying the **DCL prompt**, which is a dollar sign (\$) unless you explicitly change it. A command is entered by typing it at the dollar sign prompt and pressing the <RETURN> key.

Most commands cause DCL to run a program in your process. The program is then responsible for further analyzing the command and eventually carrying it out. Once the program has finished, DCL prompts for another command from the keyboard. A few commands do not run any program but are instead carried out directly by the DCL interpreter. These are simple commands that do not warrant their own complete program.

The DCL command repertoire is extremely rich in features, as the size of the *VMS DCL Dictionary* attests. The *VMS DCL Dictionary* is the manual in the VMS Documentation Set that describes most of the commands and other features of DCL. Whenever you are in doubt about the features of a command, consult the *Dictionary*.

2.3 **DCL Commands**

Every DCL command adheres to the same basic command format. The first item in a command is called the **command verb**. The command verb is a single English word, or combination of words, that identifies the command and summarizes what it does. For example, the command LOGOUT is used to terminate an interactive session. Command verbs may be entered in an abbreviated form as long as the abbreviation is unambiguous. The LOGOUT command may be abbreviated to LOG but not to L, because there is also a LINK command. A verb is guaranteed to be unique if four or more letters are specified.

Many commands accept one or more **parameters** following the verb. A parameter is an item of information used by the command to further refine the action it will take. Often a parameter names some entity upon which the command will operate, such as a data file to be copied. The command verb specifies the action, while the parameter specifies an object to be acted upon.

Some command parameters are **required parameters**; they must always be specified. Others are **optional parameters** and are specified only when needed. Each command description points out which parameters are required and which are optional.

The following command deletes all the files called NAMES .DAT:

```
$ delete names.dat;*
```

The command verb is DELETE and the first parameter is the file specification NAMES .DAT;*. The format for file specifications is described in a later section.

The operation of DCL commands is modified and augmented by the presence of command **qualifiers**. Almost all qualifiers are optional. When specified, they alter certain aspects of the operation performed by the command. A qualifier consists of the slash character (/) followed by a word that describes the qualifier's effect on the command. In addition, some qualifiers accept values that further determine the operation. As with the command verb, qualifiers may be abbreviated as long as the abbreviation is unique (four letters guarantee uniqueness).

Here is the DELETE command with two qualifiers:

```
$ delete /confirm /log names.dat;*
```

The /CONFIRM qualifier specifies that the DELETE command should ask the user to confirm each file deletion before it is performed. The /LOG qualifier requests that a message be displayed naming each file after it is deleted. Without the qualifiers the command neither confirms nor displays messages about files as it deletes them. The particular action taken when a qualifier is omitted is called the default action. The default action for the /CONFIRM qualifier is to not confirm; the default for the /LOG qualifier is to not display log information.

When a qualifier requires a value, the qualifier name is followed by an equal sign (=) and then the value. For example:

```
$ delete /before=1-jan-1988 names.dat;*
```

The /BEFORE qualifier specifies that only files created before 1 January 1988 are to be deleted.

The overall format of DCL commands and the rules that govern their format are together called **command syntax**. The complete syntax rules are given in the VMS manual entitled *VMS DCL Concepts Manual*. You will learn about many DCL commands and their syntax as this book proceeds. Here is a summary of the syntax rules:

- A command begins with a verb, which may be preceded by one or more spaces. The verb may be abbreviated as long as the abbreviation is unambiguous. Verbs are guaranteed unique if at least four letters are specified.
- A command may take parameters, some of which are required and some optional. The verb and parameters are separated from one another by one or more spaces.
 - When a parameter contains spaces, commas, slashes, or lowercase letters, it must be enclosed in double quotes (").
 - Some parameters may include a list of items. The items in the list are separated by commas (,).
- Most commands allow qualifiers, which consist of a slash (/) followed by a name. The name may be abbreviated as long as the abbreviation is unambiguous. As with verbs, qualifiers are guaranteed unique if at least four letters are specified.
 - Some qualifiers require a value, others accept an optional one. The value is separated from the qualifier name by an equal sign (=).
 - When a qualifier value contains spaces, commas, slashes, or lowercase letters, it must be enclosed in double quotes (").
 - Some qualifiers may include lists of values. The items in the list are enclosed in parentheses [()] and separated by commas (,).
- A command can be continued on additional lines by ending each line (except the last) with a hyphen (-). The command can be split at any point where a space or a comma appears.

Files and Directories

A **file** is a collection of data stored on a permanent medium such as disk or magnetic tape. The information in a file is organized into **records**, each one containing a portion of the total collection of data. One of the primary purposes of any operating system is to support the creation and manipulation of data files. On VMS, a file is designated by a sequence of characters called a **file specification**, or **file spec** for short. The complete format for a file spec is given here, and each component of the file spec is summarized:

node : : device : [directory...] name . type ; version

Node. The node component identifies a particular node in a DECnet network.

The name of the node is included in the spec, followed by a double colon to distinguish it from a device name. If the node is omitted, the local node is assumed.

Device. The device component specifies the disk or tape on which the file resides. If it is omitted, the device containing the working directory is assumed (see below for a description of the working directory).

Directory. The directory component specifies a file catalog on the device, the catalog containing the desired file. Multiple catalogs are allowed on some devices. If the directory component is omitted, the working directory is assumed.

Name. The name component, taken together with the type component, identifies a particular file in the chosen directory. The purpose of the file name is to assign a meaningful identifier to the file.

Type. The type component is used with the file name to uniquely identify a file in a directory. This two-level name/type scheme allows the file name to identify a family of related files, while the file type identifies particular kinds of files within the family.

Version. The version component allows multiple generations of the same file to exist simultaneously. More than one generation can exist in the same directory. If the version component is omitted, the latest version is assumed.

Certain classes of storage media, most notably disks, can contain hundreds or thousands of data files. The concept of a **directory** is introduced to allow a collection of files to be cataloged together under one name, while another collection

is cataloged under a different name. A directory can catalog not only data files but other directories as well. This permits a hierarchy of directories to be formed, with data files cataloged at all levels of the hierarchy.

Two directories are of notable importance to the VMS user. When a user logs in to VMS, one particular directory is established as the user's **login directory**, or **home directory**. This directory acts as the repository for the user's files, and the user can create subdirectories under the login directory for additional files. At any time during a VMS session, one directory, not necessarily the login directory, is chosen as the **default directory**. When a file is accessed using a file spec that does not include a device and directory, the file is assumed to reside in the default directory. Thus, if the directory in which you are currently working is established as the default directory, most file specs need not include an explicit device or directory. When you log in, the login directory is automatically established as the default directory. You can change the default directory with the SET DEFAULT command.

Files and directories are discussed in detail in Chapters 13 through 18.

2.5 **Logical Names**

A **logical name** is a named entity that stands for part or all of a file specification. The primary purpose of a logical name is to relieve the VMS user from having to remember the disk and directory location of a file. To this end, VMS defines many logical names such as SYS\$HELP, which you can use to locate system files.

A logical name is created and assigned a value with the DEFINE command. The DEFINE command requires a logical name and its corresponding value:

```
$ define data_files user_disk:[smith.data]
```

Once the logical name DATA_FILES is defined, a file in Smith's data directory can be referred to without knowing either the disk or the directory:

```
$ type data_files:october.dat
```

This command will display the contents of the OCTOBER.DAT file in the data directory. Logical names defined in the manner of DATA_FILES are only available to your process. Other logical names defined by VMS or the system manager can be made available to all processes.

Chapter 14 describes logical names in detail.

2.6

User Identification Codes

The **user identification code** (UIC) is a number assigned to certain objects to identify the **owner** of the object. Files and disk volumes are two kinds of objects that have an owner. A file is assigned an owner when it is created, a disk volume when it is initialized. A UIC is also assigned to each authorized user of a VMS system. When you log in to VMS, your UIC is located and becomes associated with the interactive process VMS creates for you. Every VMS process has a UIC associated with it.

VMS uses the UIC as part of its object protection scheme. An object can be protected so that only certain users can read or modify it. For example, a file can be protected so that many users can read it, fewer users can modify it, and only one user can delete it. File protection is described in detail in Chapter 18. The protection scheme is based upon comparing the UIC of the process accessing the object with the owner UIC of the object itself.

A UIC is an integer that is divided into two parts. The first part is called the **group number** and the second part the **member number**. The system manager determines how the group/member hierarchy is used when assigning UIC numbers to users. A typical scheme is that all the users in one department are assigned the same group number, with separate member numbers assigned to each person in the group.

▷ Ch. 18

A numeric UIC is displayed in the format [123,456], where 123 is the group number and 456 is the member number. The group and member numbers are specified and displayed in the octal number system, which means that their digits must range from 0 through 7 (8 and 9 are not allowed). VMS Version 4 introduced the idea of UIC identifiers, which are referred to as **rights identifiers**. Each group is assigned a mnemonic identifier, such as DEVELOPMENT, and each group member is also assigned an identifier, such as ROBERTSON. If the development group is group 123 and Robertson is member 456, then Robertson's UIC is displayed as [DEVELOPMENT,ROBERTSON] instead of [123,456]. This identifier format conveys much more information than the numeric format. A member identifier is almost always identical to the member's user name.

▷ Ch. 18

Certain group numbers are reserved for **system groups**. A user in a system group can generally access system files that are not available to regular users. The system group numbers usually range from 1 through 10 octal (1 through 8 decimal), although the system manager can alter the range.

2.7

Privileges

The VMS privilege scheme is the means by which VMS restricts the use of those system functions and resources that can have a negative impact on the system when used indiscriminately. For example, the capability to terminate another user's interactive session has the potential to be misused. Another example is the ability to delete files that normally cannot be deleted: only a few authorized users should be able to do so.

Every critical system function has one or more privileges associated with it. Each privilege has a name, so that it can be specified in DCL commands. The OPER privilege, for example, is required to perform various operator activities, such as broadcasting a message to all system users. Only users with OPER privilege can broadcast messages to all users.

There are four sets of privileges associated with each VMS process, two of which are important to DCL. The first set is the **authorized privileges**, which are assigned to a user by the system manager. The authorized privileges are those the user is allowed to enable. As long as OPER privilege is authorized, a user can enable it and then broadcast a message to all system users. The second privilege set is the **process privileges**. These are the privileges the user has in fact enabled. Again, the system manager determines which privileges will be enabled when a user logs in, but the user can change the enabled privileges at any time with the SET PROCESS/PRIVILEGES command. A user cannot enable privileges that are not in the authorized set unless the user has SETPRV privilege. The SETPRV privilege allows a user to enable any privileges whatsoever, and therefore it should be authorized with extreme caution.

Some DCL commands can only be performed when you have certain privileges enabled. Such restrictions will be stated in this book whenever they apply.

2.8

DCL Procedures

After using VMS for a while, you may find yourself entering the same sequence of DCL commands over and over. This quickly becomes tedious, so users look for a way to capture the command sequence, assign it a name, and request the command sequence by name. A command sequence can be stored in a text file using an editor, and the text file can be "played back" later by DCL, just as if the commands were reentered at the terminal. The command text file is called a **DCL command procedure**. Command procedures are what this book is all about.

A command procedure is a file of text that you can create with your favorite editor. In its simplest form, a procedure file consists of a sequence of DCL commands, one per line:

```
$ show default
$ show process
$ show quota
```

Each command in the procedure must begin with a dollar sign, which tells DCL that the line contains a command for it to interpret. One or more spaces can follow the dollar sign. Then the DCL command is specified exactly as it would be typed at the DCL prompt.

By convention, a DCL command procedure file is given the file type COM, for “command.” Assume that the previous procedure is contained in a file named SIMPLE.COM. Once a procedure file is created with an editor, the commands in the procedure can be played back using the at-sign (@) command:

```
$ @simple.com
```

The at-sign tells DCL that the file spec for a command procedure follows. DCL looks for the file SIMPLE.COM and interprets the commands contained in it, one at a time, until it runs out of commands. When the commands have all been interpreted, DCL requests another command from the terminal. The at-sign command is said to **invoke** or **call** the procedure, and DCL subsequently **executes** the commands contained within it. The commands in a DCL procedure are sometimes called **DCL code**.

DCL assumes that the file type of a procedure is COM, so the .COM can be omitted from the preceding command:

```
$ @simple
```

The procedure displays the following information at the terminal:

```
$DISK3: [GREEK.PERSONAL.BOOK]

9-MAY-1988 19:45:00.12 LTA5:           User: GREEK
Pid: 0000015A   Proc. name: GREEK       UIC: [AMCDEV,GREEK]
Priority: 4     Default file spec: $DISK3:[GREEK.PERSONAL.BOOK]

Devices allocated: LTA5:
User [AMCDEV,GREEK] has 6956 blocks used, 33044 available,
of 40000 authorized and permitted overdraft of 100 blocks on $DISK3
```

If a procedure line contains nothing but a dollar sign, DCL ignores it and goes on to the next line. Such a line can be used for visual separation of commands:

```
$ show default
$ show process
$
$ show quota
```

2.8.1 **Comments**

A **comment** in a command procedure is a phrase or sentence that describes the procedure to the human reader. Comments are completely ignored by DCL; they have absolutely no effect on the execution of the procedure, the actions it takes, or the output it displays. The beginning of a comment is marked with an exclamation point (!) and the rest of the line contains the comment. DCL ignores the exclamation point and all other characters through the end of the line.

```
$! This is the procedure SIMPLE.COM
$! It displays information about the current process.
$
$ show default
$ show process        ! The process name and other info.
$
$ show quota         ! Disk space used by this user.
```

This procedure contains four kinds of lines:

- Lines beginning with \$!, which contain only comments.
- Separator lines containing only the dollar sign.
- Command lines with no comments.
- Command lines that also include a comment at the end.

Comments allow a complex procedure to be annotated so that future readers and maintainers of the procedure can better understand how the procedure works.

2.8.2 **Abbreviation**

DCL allows command verbs and qualifiers to be abbreviated. In a command procedure, however, it is best to avoid abbreviation. Abbreviations are harder to read, and searching for particular command verbs or qualifiers is difficult

when spellings are inconsistent. If you insist on abbreviating a verb or qualifier, *never* abbreviate to fewer than four letters. The names of verbs and qualifiers are guaranteed to be unique within the first four letters, but not in three or fewer.

Take the SET PROTECTION command, for example. In VMS Version 2, you could include the following line in a procedure:

```
$ set pro=(o:rwd) *.dat;* ! Abbreviate "protection".
```

However, the SET PROMPT command was added in a later version of VMS. Suddenly the abbreviation PRO was not unique, because it matched both PROTECTION and PROMPT.

2.9 Procedure Verification

The following command will cause DCL to **verify** the lines in a command procedure as it performs them:

```
$ set verify
```

As DCL reads the command procedure and interprets its commands, each command line is displayed at your terminal. In this way you can trace the procedure and see what it is doing. The SET VERIFY command can be useful as you read this book and try out examples.

Procedure verification is described in Chapter 12.

2.10 The Login Procedure

After you log in to VMS, DCL checks your login directory for a file named LOGIN.COM. If such a file exists, it is automatically invoked as if specified with an at-sign command. All the commands in this login procedure are executed before DCL issues the first command prompt. The login procedure can be used to establish a personal working environment: it can define logical names, enable privileges, or change the DCL prompt.

As you read further, you may discover interesting things to put in your login procedure.

Chapter 3



Symbols, Data, and Expressions

Chapter 2 introduced basic DCL concepts, which are important for all VMS users. This chapter begins an examination of DCL facilities that make it useful as a general-purpose programming language. The focus of this chapter is on data and the manipulation of data.

3.1

Symbols

A **symbol** is the DCL equivalent of what most programming languages call a variable. A symbol is a named entity with which you can associate an item of data. Later on, the data can be retrieved and manipulated by using the name to refer to it. The item of data is called the symbol's **value**. The same symbol may have different values at different points in the program. It is the ability of a symbol to take on different values that makes it such a powerful programming tool. A DCL symbol has three items of information associated with it:

Name. Each symbol has a name, which is used to refer to it. The name of a symbol, together with its level, uniquely distinguishes it from all other existing symbols.

Level. Each symbol has a level, which is determined by the context in which the symbol was originally created. The various symbol contexts are described in Section 3.3. A symbol with the same name may exist in two different contexts, but duplicate names may not exist in the same context. Therefore, a symbol's name and level uniquely distinguish it from all other symbols.

Value. Each symbol has a value associated with it. The value is an item of data, which can be manipulated by referring to it by way of the symbol's name. In other words, the symbol name acts as a "handle" for the data item.

A symbol name is composed of letters, digits, dollar sign (\$), and underscore (_), but its first character cannot be a digit. You may type a symbol name in lowercase or uppercase letters; DCL converts the name to uppercase before doing anything with it. A symbol name is limited to 255 characters in length. The symbol value is an item of data, associated with the symbol, that can be accessed using the symbol name. In DCL, data items can be integers or character strings.

Here is a simple example to illustrate the creation of a symbol:

```
$ sym1 = 42
```

This is an **assignment command**, which creates a new symbol. The symbol has the four-character name `SYM1`, a level determined by the context of the assignment command, and the value 42, which is an integer. Once a symbol is created, its value can be replaced with a new value by performing another assignment command:

```
$ sym1 = "I'd rather be sailing."
```

This command does not create a new symbol but rather replaces the existing symbol's value with the character string "I'd rather be sailing.". An assignment command creates the symbol if it does not already exist and then sets it to the value specified on the right-hand side of the command.

It is difficult to fully appreciate the power of symbols without some further background. The rest of this chapter describes symbols, data items, and assignment commands in detail.

3.2 **Types of Data**

The ultimate purpose of every program is to create and manipulate data. The collective term **data** refers to all the information available to, or generated by, a program. This data is composed of individual **data items**, each of which can be manipulated separately from the rest. In DCL, a data item has an associated **data type**, which signifies what kind of data it is. Strictly speaking, DCL supports two types of data: integers and character strings. However, a particularly useful third type of data, boolean data, can be implemented with the first two. The following sections discuss these data types.

Every data item has two representations, an internal one and an external one. The internal representation is determined by the host computer, which for DCL is the VAX. Almost all computers operate in binary, representing data items as numbers in base 2. Each binary digit is called a **bit**, so we speak in terms of data occupying a certain number of bits of computer memory.

Computer users are much happier when data can be represented in natural forms such as decimal numbers or letters of the alphabet. These familiar forms are the external representations of the data, and all programming languages provide for them. When the external representation of a data item appears in a program, it is called a **literal**. The following DCL command contains two literals:

```
$ write sys$output "The answer is: ", 13
```

The first literal is the character string "The answer is: " and the second is the decimal integer 13.

3.2.1 **Integers**

An integer is a whole number, which can be negative, zero, or positive. DCL supports integers that occupy 32 bits of memory, thus restricting them to the range from -2, 147, 483, 648 to +2, 147, 483, 647. In the VAX architecture, a 32-bit quantity is referred to as a **longword**. Although the internal representation of an integer is in binary, the external representation is made up of digits and other characters, which allows a "natural" specification of the number.

Integer literals can be represented in base 10 (decimal), base 16 (hexadecimal), or base 8 (octal). Decimal integers are discussed in this section, hexadecimal integers in Appendix A. Octal integers are rarely used and will not be discussed in this book. A decimal integer is composed of the ten decimal digits 0–9, optionally preceded by a plus sign (+) or minus sign (-). Punctuation marks such as commas are not allowed in integers. The following are examples of decimal integers:

```
0 7 07 42 +3286 -1 -39840938
```

3.2.2

Character Strings

A character string is a sequence of individual characters. A character can be an actual glyph, such as an uppercase A or a plus sign (+). Glyphs are usually called **printable characters**. A character can also be a **control character**, which does not represent a glyph but rather a formatting operation, such as tab or line feed. Each distinct character is assigned a code number, and the complete collection of characters and their code numbers is called a **character set**. The VAX architecture employs the **ASCII character set** (ASCII stands for American Standard Code for Information Interchange), in which characters occupy eight bits and the code numbers range from 0 to 255. A complete ASCII character table is included in Appendix B.

The number of individual characters in a character string is called its **length**. A character string can be composed of any number of characters from one to approximately 900. The upper limit depends on the context in which the string is used. A character string can also contain no characters at all, in which case it is called the **null string**.

A character string literal is represented by enclosing its constituent characters in a set of double quotes ("). Here are a few examples:

```
" " "x" "X" "*" "a short string"  
"This is a sentence complete with punctuation."
```

The first example is the null string. You can represent any printable character in a string literal, but there is no way to represent control characters. A special case is made of the double quote character: you must represent each double quote character in the literal with an adjacent pair of double quotes:

```
"This string contains ""a quoted phrase""."
```

The string literal shown here consists of six words, the last three of which are inside double quotes. Each double quote that is part of the literal, but not the ones that enclose the literal, must be paired. It is a common mistake to forget to pair the double quotes inside a string:

```
"This string contains "a quoted phrase"."
```

It appears to DCL that the string literal ends with the space following the word contains and that the final three words are not part of the string literal.

Table 3.1 DCL's Interpretation of Data as Boolean Values

Boolean Value	Data So Interpreted
TRUE	Odd integers; character strings beginning with t, T, y, or Y; character strings representing odd integers (e.g., "381")
FALSE	Even integers; all other character strings

3.2.3

Booleans

The boolean data type encompasses the two logic values true and false. Logical values are important in programming because true/false, yes/no decisions are constantly being made by a program in order to guide its flow of execution. Many conventional programming languages distinguish the boolean data type from other data types, but DCL does not. However, the boolean data type can be simulated using the integer and character string data types. Whenever DCL needs a true/false value to use with a logical operation, it accepts an integer or character string and interprets it as true or false depending on its value. Table 3.1 specifies how this interpretation is made.

The following data items are considered true:

```
1 3 "True" "Y" "YES" "yupsters"
```

The following are considered false:

```
0 2 "False" "N" "NO" "nope" "random!nonsense" "380"
```

Symbolic literals for the two boolean values true and false round out the simulation of the boolean data type. DCL provides no such literals, so the author chooses the two symbols TRUE and FALSE to represent them. These symbols can be established with the assignment command described in the next section.

3.3

Assignment Commands

The assignment command is the means by which DCL symbols are created and assigned values. Because most languages refer to their commands as **statements**, the assignment command is often called the assignment statement. The assignment statement has the following general format:

```
$ symbol = expression
```

The symbol named on the left-hand side of the equal sign is assigned the value determined by the **expression** on the right-hand side. For example:

```
$ my_name = "Fred Shubin"
```

Here the symbol `MY_NAME` is assigned the character string "Fred Shubin". We read the assignment statement as "MY_NAME gets "Fred Shubin"" and say that the symbol `MY_NAME` is set to the value of the expression "Fred Shubin". The word *equals* is not used when discussing the action of an assignment statement because it is too easily confused with the idea of equality for comparison purposes, as in "1 does not equal 2." Integer and string literals are examples of simple expressions; a literal can appear by itself on the right-hand side of an assignment statement. Section 3.4 describes expressions in detail.

A symbol is created the first time a command is used to assign it a value. Subsequent assignments to the same symbol discard the current value and replace it with the new value. A symbol can have an integer value at one point in the program and a character string value at another point. The data type is associated with the value, not with the symbol, as it is in many languages such as Pascal. Because the data type is associated with the value and not with the symbol, a symbol can be set to any type of data. The following assignment statements illustrate these rules:

```
$ days_per_year = 366
$ days_per_leap_year = days_per_year
$ days_per_year = 365
$ days_per_leap_year = "three hundred sixty-six"
```

In the second assignment statement, the symbol `DAYS_PER_LEAP_YEAR` is set to the value of the previously created symbol `DAYS_PER_YEAR`. A symbol by itself is another example of a simple expression on the right-hand side of an assignment command. Note that `DAYS_PER_LEAP_YEAR` is assigned both types of data at different times.

In addition to its name and value, each symbol has a level. The name and the level together uniquely identify the symbol from among all existing symbols. A symbol's level is determined by the context in which it is created and by the kind of assignment statement used to create it. The following sections describe the different levels: prompt level, procedure level, and global level.

3.3.1 **DCL Prompt Level**

Symbols created at the DCL prompt are associated with the DCL prompt level. You might create such a symbol to remember something to do later:

```
$ note = "Remember to call Jim before leaving."
```

This symbol is at the prompt level because it was created with an assignment statement at the DCL prompt. You can display the value of the symbol with the `SHOW SYMBOL` command (you type the first line, and VMS responds with the second line):

```
$ show symbol note
NOTE = "Remember to call Jim before leaving."
```

The symbol can be used to create another symbol at the prompt level:

```
$ old_note = note
$ note = "Don't forget to pick up a video tape."
```

Now two prompt-level symbols exist, `NOTE` and `OLD_NOTE`.

Symbols at the prompt level remain in existence during your entire login period unless explicitly deleted. Section 3.9 describes how to delete symbols.

3.3.2 **Procedure Levels**

Each DCL command procedure has its own symbol level. When you invoke a procedure with the at-sign (`@`) command, a new level is established. Any symbols created by the procedure are associated with the procedure's level. If the procedure invokes another procedure, the second procedure has its own level and its symbols are associated with its level. In this way, each procedure has a set of symbols that "belong" to it.

Here is a simple procedure named `SIMPLE-PROC`:

```
$ name = "Moon Unit"
$ show symbol name
```

When invoked, this procedure creates the procedure-level symbol `NAME` and assigns a character string to it. The procedure then displays the value of the symbol:

```
$ @simple-proc
NAME = "Moon Unit"
```

When a procedure terminates, all the symbols created at its level are deleted. Since the symbols are associated with the procedure's level, it makes no sense for them to exist once the procedure has terminated. Should the procedure be invoked again, its symbols will be recreated from scratch by DCL; they will have lost their old values.

A procedure can refer to a symbol created at the prompt level as long as the symbol does not have the same name as a symbol created by the procedure. If symbols with the same name are created at the prompt and procedure levels, the one at procedure level "shadows" or "hides" the one at prompt level. Similarly, a subprocedure invoked from a main procedure can refer to symbols created by the main procedure. For example, if procedure A invokes procedure B, B can refer to the symbols created by A. The ability to use symbols created at an outer level allows you to write a procedure that displays the NOTE symbol created at the prompt level:

```
$! Procedure to display the note.  
$  
$ write sys$output "Your note is:"  
$ show symbol note
```

▷ Ch. 6

The WRITE command displays the text in double quotes. The SHOW SYMBOL command refers to the NOTE symbol created at the prompt level. Because this little procedure does not create its own symbol named NOTE, the reference to NOTE gets the symbol at prompt level. The display produced by this procedure is shown here:

```
Your note is:  
Don't forget to pick up a video tape.
```

When DCL is executing a procedure and needs the value of a symbol, it performs a simple search process. As soon as it finds a symbol with the given name, it terminates the search and uses the value of the symbol. Here is the process as described so far:

1. DCL looks for the symbol among the symbols created by the currently executing procedure.
2. If the procedure was invoked by another procedure, DCL looks for the symbol among the invoking procedure's symbols.

3. Step 2 is repeated for each additional level of command procedure.
4. DCL looks for the symbol among the symbols created at DCL prompt level.

Although a procedure can use the values of symbols at outer levels, it cannot create or change symbols at outer levels. This is somewhat restrictive, so a special global level exists to solve the problem.

3.3.3 **Global Level**

Sophisticated DCL applications are composed of multiple procedures, which invoke one another in various combinations. Sometimes it is necessary for a procedure B to calculate a value and pass it back to another procedure A, which invoked it. With what you know so far, there is no way to accomplish this. If B stores the value in a symbol at its own level, that symbol will be deleted when B terminates. The alternative is for B to store the value in a symbol at A's level, but DCL provides no means of doing so. The global symbol level exists to solve this dilemma.

Symbols created at the global level are called **global symbols**. You must explicitly request that a global symbol be created by using a variant of the assignment statement:

```
$ xda_answer == 42
```

Note that two equal signs are used in the example. The double equal sign requests that the symbol XDA_ANSWER be created at the global level and assigned the value 42. Global symbols are only created when you use the double equal sign form of the assignment command.

▷ Ch. 10 By convention, global symbol names begin with the application facility code and a single underscore. In the preceding example, the facility code is XDA.

Global symbols can always be created, whether you are at the DCL prompt or executing a command procedure. This capability is what distinguishes global symbols from prompt-level symbols and is the only major difference between the two kinds of symbols. The double equal sign forces a global symbol to be created regardless of the level at which the assignment statement is executed. Subsequent global assignment to the same symbol changes the global symbol's value. The value of a global symbol can be obtained at any level as long as there are no symbols of the same name at a procedure level or prompt level. To accommodate global symbols, the symbol search process is extended to its final form:

1. DCL looks for the symbol among the symbols created by the currently executing procedure.
2. If the procedure was invoked by another procedure, DCL looks for the symbol among the invoking procedure's symbols.
3. Step 2 is repeated for each additional level of command procedure.
4. DCL looks for the symbol among the symbols created at prompt level.
5. DCL looks for the symbol among the global symbols.

The following simple example illustrates a subprocedure that creates a global symbol in order to pass a value back to its calling procedure:

```
$! This is procedure A.
$
$ @b ! B will set global symbol XDA_ANSWER.
$ show symbol xda_answer

$! This is procedure B.
$
$ xda_answer == 42
$ exit
```

Procedure A invokes procedure B to establish the global value. Procedure B creates the global symbol XDA_ANSWER using a double equal sign assignment statement. It then exits, allowing procedure A to continue. Procedure A displays the value of the global symbol. Because XDA_ANSWER is global, it is not deleted when procedure B exits, thus allowing A to obtain its value.

It is quite easy to omit the second equal sign when you mean to perform a global assignment. Look at the following code:

```
$ xda_answer == 42
.
.
$ xda_answer = 43 ! Meant to use ==
$ show symbol xda_answer
```

The first assignment command creates the global symbol XDA_ANSWER and assigns it the value 42. The second assignment command was intended to change the value of the global symbol, but only one equal sign is present. Therefore, the assignment command creates a *procedure-level* symbol with the same name, XDA_ANSWER. This procedure-level symbol hides the global symbol. The SHOW command displays 43 quite nicely, but the global symbol still has the value 42. Be careful always to use two equal signs when performing global assignments.

Global symbols remain in existence during your entire login period, unless explicitly deleted. Section 3.9 describes how to delete symbols.

3.4 Expressions

▷ Ch. 4

An assignment statement assigns a value to a symbol. The value is determined by an expression on the right-hand side of the equal sign. Expressions are also used in other DCL commands, such as the IF command. Literals and symbols have already been used as simple expressions. When used in an expression, a literal stands for itself and a symbol stands for its current value. These simple expressions are useful but are not powerful enough to compute new values, such as the sum of two integers.

New values are computed using expressions composed of **operators** and **operands**. An operator is a character or sequence of characters that stands for some mathematical operation, such as multiplication, or for a string operation, such as concatenation. The operands associated with an operator determine the values that are to participate in the operation. Here is a simple expression:

```
a * b
```

This denotes that the value of the symbol A is to be multiplied by the value of the symbol B to produce a new value. The fate of the new value is determined by the context in which the expression appears. So far, the only context in which an expression can appear is the assignment statement:

```
$ product = a * b
```

Here, the product of the values of A and B is assigned to the symbol PRODUCT. When DCL encounters an expression, it applies the operators to their operands in a certain predetermined order, producing a final result, which is assigned to a symbol or used for some other purpose. When DCL processes an expression in this manner, we say that DCL **evaluates** the expression.

In order to completely understand expressions, you must become familiar with the available operators, the operands they expect, and the order in which the operators are applied to their operands. Tables 3.2, 3.3, and 3.4 describe the operators provided by DCL for use with integer, character string, and boolean values, respectively. Table 3.5 illustrates the order in which operators are applied.

Not all operators require two operands as multiplication does. Some require only one operand. (In the C language there is an operator that requires three operands.) The number of operands required by an operator is called its **arity**.

Table 3.2 Integer Operators

Operator	Arity	Result Type	Result Value
+	Unary	Integer	Integer operand unchanged.
-	Unary	Integer	Negative of integer operand.
+	Binary	Integer	Sum of integer operands.
-	Binary	Integer	Difference of integer operands.
*	Binary	Integer	Product of integer operands.
/	Binary	Integer	Quotient of integer operands, truncated towards zero.
.EQ.	Binary	Boolean	True if integer operands are equal, false otherwise.
.NE.	Binary	Boolean	True if integer operands are unequal, false otherwise.
.GT.	Binary	Boolean	True if first integer operand is greater than second, false otherwise.
.GE.	Binary	Boolean	True if first integer operand is greater than or equal to second, false otherwise.
.LT.	Binary	Boolean	True if first integer operand is less than second, false otherwise.
.LE.	Binary	Boolean	True if first integer operand is less than or equal to second, false otherwise.
.NOT.	Unary	Integer	Bitwise boolean NOT of integer operand. A bit in the result is 1 if the corresponding bit in the operand is zero, and vice versa.
.AND.	Binary	Integer	Bitwise boolean AND of integer operands. A bit in the result is 1 if both of the corresponding bits in the operands are 1.
.OR.	Binary	Integer	Bitwise boolean inclusive-OR of integer operands. A bit in the result is 1 if either or both of the corresponding bits in the operands are 1.

Operators with an arity of 2 are called **binary operators**. Those with an arity of 1 are called **unary operators**. A few examples:

```
$ sum = a + b - c
$ sum = -sum
$ positive = sum .gt. 0
$ negative = .not. positive
```

Table 3.3 Character String Operators

Operator	Arity	Result Type	Result Value
+	Binary	String	A copy of the first string operand with a copy of the second one concatenated to it (e.g., "Hello-" + "there." produces "Hello-there.").
-	Binary	String	A copy of the first string operand with the leftmost occurrence of the second one removed from it (e.g., "oh-why-oh-why" - "why" produces "oh--oh-why").
.EQS.	Binary	Boolean	True if string operands contain the same character sequence, false otherwise.
.NES.	Binary	Boolean	True if string operands contain different character sequences, false otherwise.
.GTS.	Binary	Boolean	True if first string operand is alphabetically greater than second, false otherwise. The collating sequence is based on the ASCII character set.
.GES.	Binary	Boolean	True if first string operand is greater than or equal to second, false otherwise.
.LTS.	Binary	Boolean	True if first string operand is less than second, false otherwise.
.LES.	Binary	Boolean	True if first string operand is less than or equal to second, false otherwise.

The first example contains an expression composed of two binary operators. The operands for the plus operator are A and B; the operands for the minus operator are the resulting sum and C. The second example uses the unary minus operator to negate its operand. Notice how the hyphen character is used as two different operators with different arities, its meaning determined by context. The third example uses the binary “greater than” operator to compare two numbers. The final example uses the unary “not” operator to invert its boolean operand. The operator tables specify the arity and meaning of every DCL operator.

The order in which operators are applied to operands is determined by operator **precedence**. Table 3.5 lists the precedence of the DCL operators. An operator with a high precedence is applied before an operator with a lower precedence, regardless of the order of their appearance in the expression. Every operator is assigned a precedence so that the order of application can be determined without ambiguity. Here are a few expressions to illustrate operator precedence:

Table 3.4 Boolean Operators

Operator	Arity	Result Type	Result Value
.NOT.	Unary	Boolean	True if boolean operand is false, false if it is true.
.AND.	Binary	Boolean	True if both boolean operands are true, false otherwise. There is no guarantee about which operand is evaluated first.
.OR.	Binary	Boolean	True if either or both boolean operands are true, false otherwise. There is no guarantee about which operand is evaluated first.

```
$ value = a * b + c
$ value = c + a * b
$ value = a * b - c * d
$ value = -x + y
```

Because multiplication has a higher precedence than addition, the first two examples both multiply A by B before adding C. This is true even though, in the second example, the multiply operator appears after the add operator. The order of evaluation is determined by operator precedence, not merely by order of appearance. The third example calculates the product of A and B, and then the product of C and D, and finally subtracts one product from the other. The fourth example negates X and then adds Y; the precedence of unary minus is higher than that of addition. If there are two or more operators of equal precedence in an expression, such as the multiply operators in the third example above, the operators are evaluated from left to right. In the third example, $A * B$ is evaluated before $C * D$.

Sometimes the order of evaluation determined by operator precedence is not what you want. Parentheses are used to force operators to be evaluated in a certain order regardless of their precedence. When parentheses surround a portion of an expression, that portion is evaluated before the surrounding expression, regardless of precedence. Here are the preceding examples with parentheses added:

```
$ value = a * (b + c)
$ value = (c + a) * b
$ value = a * (b - c) * d
$ value = -(x + y)
```

Table 3.5 Operator Precedence

Precedence	Operators
8 (highest)	()
7	Unary + -
6	* /
5	Binary + -
4	.EQ. .NE. .GT. .GE. .LT. .LE. .EQS. .NES. .GTS. .GES. .LTS. .LES.
3	.NOT.
2	.AND.
1 (lowest)	.OR.

The first example now calculates the sum of B and C and then multiplies it by A. The sum appears in parentheses, so it is evaluated first, even though the precedence of multiplication is higher. The second one adds C and A and then multiplies the sum by B. The third example subtracts C from B, multiplies the difference by A, and then multiplies that result by D. The final example adds X and Y and negates the resulting sum. In each case, the final value is different when parentheses are used.

As DCL evaluates an expression, it must decide whether each operand represents an integer, string, or boolean value. In some cases, the type of the operands actually affects the meaning of the operator. Such a case is the plus (+) operator, which performs addition when its operands are integers, but performs string concatenation when its operands are character strings. DCL uses the following rules to match operators and operand types:

- If the operator accepts only integer operands (e.g., * for multiply), then any string operands are first converted to integers.
- If the operator accepts only string operands (e.g., .EQS. for string compare equal), then any integer operands are first converted to strings.
- If the operator accepts either integers or strings (e.g., + for add or concatenate) and its operands are of different types, then integers win over strings and the string operand is first converted to an integer. For example, if you attempt to add an integer and a string, the string is first converted to an integer.

- If the operator is a boolean operator (e.g., .AND. for logical and), then the operands are interpreted as boolean values according to the rules given in Table 3.1.

A string can be converted to an integer as long as it contains a valid external representation of an integer (e.g., "-372" can be converted to -372). If it does not, it is converted to the integer 0. An integer can always be converted to a string by simply creating a string containing its external representation.

Because these operator/operand matching rules are complicated, it is best to avoid using operators with mixed operand types. You can explicitly request that a string be converted to an integer, or vice versa, using the lexical functions F\$INTEGER and F\$STRING presented in the next section.

3.5

Lexical Functions

So far, we have worked with operands that can be integer literals, character string literals, symbols, or expressions. There is another form of operand called the **lexical function**. A lexical function, or simply function, is a built-in DCL subroutine that can perform complex operations related to character strings, files, processes, and other VMS entities. A lexical function may be used in an expression wherever an operand is allowed. The general format of a lexical function is as follows:

`f$name(argument, . . .)`

Each lexical function has a name beginning with F\$ and includes a meaningful word or phrase that describes what the function does. Following the name is a pair of parentheses enclosing the **arguments** to the function. Each argument is itself an expression whose value is needed by the function in order to perform its intended operation. A lexical function may require one or more arguments, each of which is separated from the others by a comma. A few lexical functions require no arguments, but the parentheses must be included anyway.

A lexical function uses its argument values to perform its intended operation. The operation always results in a new value, which is returned by the function. The **return value** is used *in place of* the lexical function as DCL continues to evaluate the expression in which the function appears. Lexical functions are similar to symbols in this regard: the value of the symbol or lexical function is used in place of the symbol or lexical function itself.

A simple lexical function is F\$LENGTH. It requires one argument, which must be a character string. The function determines the length of the string and returns it as an integer value. Assume that the symbol NAME contains a character string:

```
$ name_length = f$length(name)
$ name_too_long = f$length(name) .gt. 31
```

In the first example, F\$LENGTH is used by itself to determine the length of the name and assign it to the symbol NAME_LENGTH. In the second example, the name length is compared to 31 to decide whether it is too long. The symbol NAME_TOO_LONG is set to true if the length is greater than 31, false otherwise. Lexical functions may be used by themselves or in combination with operators and other operands.

A lexical function that requires no arguments is F\$TIME. It returns the current system time as a character string:

```
$ current_time = f$time()
```

Note that the parentheses are required even though they enclose no arguments. In this example, the system time is assigned to the symbol CURRENT_TIME.

Each argument to a lexical function is an expression. This means that an argument to a lexical function can be another lexical function, as in this example:

```
$ time_length = f$length(f$time())
```

The argument to the F\$LENGTH function is the F\$TIME function. The F\$TIME function returns the current system time, which is then handed to the F\$LENGTH function to determine its length. The F\$LENGTH function returns the length, which is assigned to TIME_LENGTH.

Some lexical functions accept **optional arguments**. An optional argument may be included or omitted according to the requirements of each use of the lexical function. If an optional argument is omitted, the function provides a **default value**, that is, a standard value that the designer of the function felt was the one most commonly needed. The F\$GETSYI lexical function is one that takes an optional argument. The function obtains information about a VMS system (“get system information”). Its first argument is required and specifies the desired item of information. The second argument is optional and specifies the VAXcluster node from which the information is to be obtained. If the second argument is omitted, the information is obtained from the local node, the one on which you are running. This default represents the most common use of the function,

particularly since the local node is the only available node on a system that is not a member of a cluster. Here are some examples:

```
$ vms_version = f$getsysi("NODE_SWVERS")
$ node_version = f$getsysi("NODE_SWVERS",node)
```

The first example obtains the VMS version for the local node and assigns it to the symbol `VMS_VERSION`. Note how the second argument is not specified, so the local node is assumed. The second example obtains the VMS version for the node whose name is the value of the symbol `NODE`. The symbol can contain the name of any node in the VAXcluster, including the local one.

When a lexical function accepts three or more arguments, and some of the arguments in the middle are not included, you must still specify the correct number of commas preceding the arguments you do include. For example, the `F$PARSE` function accepts up to five arguments. If only the first and fourth arguments are desired, three consecutive commas are required to signify that the second and third arguments are missing:

```
$ file_name = f$parse(file_spec,,, "NAME")
```

The `F$GETSYI` lexical function is one of many functions that accept **keyword** arguments. A keyword argument is a character string selected from a fixed set of strings, each of which specifies an item of information or a particular operation to be performed by the function. The first argument to `F$GETSYI` is a keyword that names the item of system information to be returned. The keyword cannot be an arbitrary character string but must be chosen from a predetermined repertoire of strings, each of which denotes a particular item. Keyword strings can be specified in uppercase or lowercase letters. In this book, all keywords appear in uppercase to emphasize that they are keywords and not arbitrary strings.

The following sections describe lexical functions that determine the type of a data item and convert between the integer and string types.

▷ Ch. 13

3.5.1

A Type-Checking Function

The `F$TYPE` lexical function provides a type-checking capability for DCL. The `F$TYPE` function requires a single argument, which must be a symbol name; an arbitrary expression is not allowed. The function inspects the value of the symbol and returns one of three strings:

"INTEGER". This string is returned if the value of the symbol is an integer or a string whose value is the external representation of an integer (e.g., "0", "42", "-4872").

"STRING". This string is returned if the value of the symbol is a string (unless the string represents a valid integer).

". The null string is returned if the symbol has not been created.

So `F$TYPE` is useful not only to determine the type of a symbol's value but also to determine whether the symbol exists at all. If the symbol has not been created, the null string is returned. Here are a few examples:

```
$ number = 42
$ type = f$type(number)
$ show symbol type
    TYPE = "INTEGER"
```

-or-

```
$ string = "Margaritaville!"
$ type = f$type(string)
$ show symbol type
    TYPE = "STRING"
```

-or-

```
$ string = " -987"
$ type = f$type(string)
$ show symbol type
    TYPE = "INTEGER"
```

-or-

```
$ type = f$type(nonexistent_symbol)
$ show symbol type
    TYPE = ""
```

3.5.2

Conversion Functions

DCL provides two lexical functions to perform data conversions: `F$INTEGER` and `F$STRING`. The `F$INTEGER` function requires a single argument, which can be any expression producing a character string result. The string is converted to

an integer, and the integer is returned. If the string does not contain the external representation of an integer, zero is returned. The `F$STRING` function requires a single argument, which can be any expression producing an integer result. As you might expect, the integer is converted to a string, and the string is returned. If the argument to one of these functions is already of the correct type, it is simply returned by the function.

Here are a few examples:

```
$ string = "108"
$ integer = f$integer("-" + string)
$ show symbol integer
    INTEGER = -108   Hex = FFFFFFF94   Octal = 37777777624
```

-or-

```
$ string = "gobble-dee-gook"
$ integer = f$integer(string)
$ show symbol integer
    INTEGER = 0   Hex = 00000000   Octal = 00000000000
```

-or-

```
$ integer = 384726
$ string = f$string(integer/2)
$ show symbol string
    STRING = "192363"
```

3.5.3 **Character String Manipulation**

The `F$LENGTH` lexical function allows you to determine the length of a character string. There are two more lexical functions that are fundamental to the manipulation of strings: `F$LOCATE` and `F$EXTRACT`.

The `F$LOCATE` function searches one string (the target) for an occurrence of another string (the pattern). It returns the **index** in the target of the first (leftmost) occurrence of the pattern. The index of the first character in a string is zero, the second character is 1, and so forth. If the pattern does not occur in the target, the length of the target is returned. The pattern is the first argument to the function, the target the second. For the following examples, assume that the symbol `PHRASE` contains an English phrase and the symbol `WORD` contains a single word:

```
$!   Find the position of a question mark in the phrase.
$
$    qm_index = f$locate("?",phrase)
```

```

$! Determine whether the phrase contains the word "who".
$
$ who_index = f$locate("who",phrase)
$
$! Find the position of the word in the phrase.
$
$ word_index = f$locate(word,phrase)

```

The F\$EXTRACT lexical function makes a copy of a portion of a character string and returns a new string containing the copy. The portion of the string to be copied is called the **substring**. A substring can be any part of a string: perhaps the first few characters, a section in the middle, the final few characters, or even the entire string. The F\$EXTRACT function requires three arguments: the starting index of the substring, the length of the substring, and the string from which the substring is to be extracted. The portion of the string beginning at the starting index and continuing for the length is copied and returned by the function. The following examples expand upon the previous ones:

```

$! Chop off the phrase at the question mark.
$
$ phrase = f$extract(0, qm_index, phrase)
$
$! Eliminate the word "who" from the phrase.
$! (Note that the hyphen is a continuation character.)
$
$ wo_who = f$extract(0, who_index, phrase) + -
           f$extract(who_index+3, 9999, phrase)
$
$! Get a copy of the word plus its surrounding characters.
$
$ context = f$extract(word_index-1, f$length(word)+2, phrase)

```

Assume that the value of PHRASE is "So who is the winner?" and the value of WORD is "the". After the above commands are executed, the values of the resulting symbols are as follows:

```

QM_INDEX: 20
WHO_INDEX: 3
WORD_INDEX: 10
PHRASE: "So who is the winner"
WO_WHO: "So is the winner"
CONTEXT: " the "

```

When working with character strings and the lexical functions that manipulate them, there are a few things to keep in mind:

- The characters in a string are indexed beginning with zero.
- Therefore, the index of the last character in a string is 1 less than the length of the string.
- The F\$LOCATE function returns the length of the target if the pattern does not occur in it.
- The first argument to the F\$EXTRACT function is an index, but the second argument is a length.
- If the starting index for an F\$EXTRACT function is past the end of the string, the null string is returned. If the length extends past the end of the string, only the existing characters are returned.
- Therefore, to extract a substring beginning at a certain point and extending for the rest of the string, specify a length greater than the longest possible string (e.g., 9999).

3.6 **Substring Assignment**

DCL provides a facility that lets you alter a substring of an existing character string. When a substring assignment is specified, you choose the portion of the string to be replaced.

The general form of a substring assignment is as follows:

```
$ symbol[index,length] := "replacement"
```

The *symbol* is the name of a symbol containing the character string to be altered; *index* and *length* specify the boundaries of the substring. The substring begins with the character specified by the index. The length determines the number of characters in the substring; it must be zero or positive. The assignment statement will replace the characters beginning at the index and extending for the length. Both the index and length can be arbitrary expressions. The replacement text is specified as a string literal following the substring assignment indicator (:=). The characters in the substring are replaced by the characters in the replacement string.

If the replacement string is shorter than the substring, it is padded on the right with spaces to make it the required length. If it is longer than the substring, it is truncated to the required length.

Here are a few examples:

```
$ string = ".This is a string."
$ string[0,1] := "("
$ string[f$length(string)-1,1] := ")"
$
$ string[0,9] := " "
$
$ i = 10
$ string[i,3] := "---"
```

The goal of the first example is to replace the first and last characters of the string with parentheses. Replacing the first character is easy: the index is zero and the length is 1. The index of the last character is determined by obtaining the length of the string and subtracting 1. This index, coupled with a length of 1, is used to replace the last character. Note that the length of a string is 1 greater than the index of its final character because the indexes begin at zero. The result of the first example is "(This is a string)".

The second example replaces the first nine characters of the string with a single space. Because the replacement string is shorter than the substring, it is padded on the right with spaces to the length of the substring being replaced. This results in a string of nine spaces that overlays the substring. It would work just as well to specify the null string ("") as the replacement value, but the single space emphasizes that the substring is being blanked. The result of the second example is " a string)".

The third example replaces three characters of the string with dashes. The starting index is specified by the symbol I, which in this case has the value 10. Therefore, the characters with indexes 10 through 12 are replaced with dashes. The result of the third example is " a---ring)".

If the starting index of the substring is beyond the end of the existing string, the string is padded with spaces to extend it to the required length. After the padding is performed, the substring is replaced. For example:

```
$ buffer = ""
$ buffer[4,4] := "abcd"
```

The BUFFER symbol is first initialized to the null string. The starting index of the substring assignment is 4, so the buffer is padded with four spaces, and then the fourth through seventh characters are set to "abcd". The resulting string is " abcd".

The substring assignment indicator := specifies that the assignment is done to a symbol at the current procedure level. The indicator == specifies that the assignment is done to a global symbol.

You must be wary of a few idiosyncracies exhibited by substring assignment:

- No space can appear between the symbol and the left square bracket ([).
- The length of the symbol name plus the length of the replacement string cannot exceed 1,024 characters.
- The maximum index value is 768. The sum of the index and length cannot exceed 769.
- In an obsolete variation of substring assignment, the replacement text need not be enclosed in double quotes. In this case, all the characters following the := operator are taken as the replacement text. The behavior of this form of assignment is confusing, because you expect an expression on the right-hand side of the assignment operator.
- The previous point makes it clear that the replacement text cannot be an expression or even a simple symbol. However, there is a way to replace a substring with the value of a symbol using a substitution operation.

▷ Ch. 5

3.7 **Bit-Field Assignment**

The bit-field assignment facility allows you to alter the sequence of bits that make up the value of a symbol. Bits can be altered in both integer and character string values. The general form of a bit-field assignment is as follows:

\$ *symbol*[*position*,*size*] = *expression*

The use of bit-field assignment to alter bits in an integer value is rare and will not be described in this book. It is more common to create or alter character strings. In order to effectively use bit-field assignment on character strings, you must first understand how the bits in a character string are numbered. The following picture illustrates the numbering scheme for the bits in the character string "ABC":

A

 0 index
7 0

B

 1 index
15 8

C

 2 index
23 16

Every character is composed of eight bits. The first character has bits numbered 0–7, the second has bits numbered 8–15, and so on. In other words, the position of the least significant bit of each character is equal to the character’s index in the string times eight. The bit-field assignment specifies both the *position* of the least significant bit and the *size* of the field to be altered. The position is limited to 6,151 and the size to 32 bits.

The assignment indicator = specifies that the bit-field assignment is done to a symbol at the current procedure level. The indicator == specifies that the assignment is done to a global symbol.

The *replacement* expression must be an integer expression. When the *symbol* specified in a bit-field assignment does not exist, DCL creates a new symbol with a character string value. The characters in the string are derived from the integer value of the replacement expression. Each bit in the integer value becomes the corresponding bit in the character string value. So bits 0–7 of the integer become the first character, bits 8–15 become the second character, and so on. A character string of up to four characters can be created in this manner. For example, here is an assignment statement that creates a character string containing a form feed:

```
$ formfeed[0,8] = %x0C
```

Assuming that the symbol FORMFEED does not exist, this assignment creates the symbol and assigns it a character string value. The bit position is 0 and the size 8, so the symbol’s value will be a one-character string derived from the replacement expression. The replacement expression has the hexadecimal value 0C, which is the ASCII code for a form feed.

There is a trick to creating a string with more than one character. The following assignment statement creates a string containing a carriage return/line feed pair:

```

$   crlf[0,16] = %x0A0D
$!           ^~--- first character
$!           ^~----- second character

```

The bit position is 0 and the size 16, so a two-character string is created. The hexadecimal literal contains two characters, a carriage return (0D) and a line feed (0A). They appear to be backwards, but remember that the first character is in bits 0–7 and the second in bits 8–15. So the carriage return will be deposited as the first character in the string, and the line feed as the second one.

Bit-field assignment is the best way to create character strings containing control characters. Do not include a control character directly in a DCL procedure, because strange things may happen when you display the procedure at a terminal or print it.

If the symbol in a bit-field assignment already exists and has a character string value, then the assignment replaces characters in the string with new ones. As always, the bit position and size specify the particular characters that are replaced, and the replacement expression gives the ASCII values of the new characters. If the position specifies bits beyond the end of the string, it is extended with zero bits to the required length. Here is an example of replacing a character in a string:

```

$   message = "Hello there.XXGoodbye."
$   message[12*8,16] = %x0A0D

```

The symbol MESSAGE is assigned a character string. The bit-field assignment then replaces the XX sequence with the carriage return/line feed combination. The bit position of the first X is specified as the product of 12, the character index of the X, and 8, the number of bits per character.

The above assignment will work, but it is certainly not perspicuous. It is better to create the message with simple concatenation:

```

$   message = "Hello there." + crlf + "Goodbye."

```

This method of creating the message does not use bit-field assignment at all. However, it does use the value of the CRLF symbol created above to obtain the carriage return/line feed characters.

Table 3.6 Summary of Assignment Commands

Command	Operation
$A = \textit{expression}$	Set procedure-level symbol to value of expression.
$A == \textit{expression}$	Set global symbol to value of expression.
$A[i, l] := \textit{text}$	Replace substring of procedure-level symbol with new text.
$A[i, l] ::= \textit{text}$	Replace substring of global symbol with new text.
$A[p, s] = \textit{expression}$	Replace bit field of procedure-level symbol with value of expression.
$A[p, s] == \textit{expression}$	Replace bit field of global symbol with value of expression.

If the symbol in a bit-field assignment exists and has an integer value, then the assignment alters bits in the integer value. This form of bit-field assignment is rarely used and will not be described in this book.

3.8 **Summary of Assignment Commands**

Table 3.6 summarizes the various forms of the DCL assignment command.

3.9 **Deleting Symbols**

Symbols at the prompt and global levels are never deleted automatically by DCL; they remain in existence as long as your process does. All the symbols created at a procedure level are deleted when the procedure exits. You can explicitly request that a symbol be deleted using the DELETE/SYMBOL command. There are two variations of the command, the first being used to delete prompt- or procedure-level symbols:

```
$ delete/symbol note
```

When used at the DCL prompt, the command will delete the prompt-level symbol named NOTE. When used in a procedure, the command will delete the procedure-level symbol NOTE.

The second variation of the command is used to delete global symbols. A global symbol can be deleted at the DCL prompt or in a procedure, as follows:

```
$ delete/symbol/global xda_answer
```

The /GLOBAL qualifier specifies that a global symbol is to be deleted.

There is rarely any reason to delete procedure-level symbols. Because they are deleted automatically when the procedure exits, you don't have to worry about lots of symbols cluttering up procedure levels or conflicting with other procedures.

Chapter 4



Flow of Control

The previous chapter presented symbols and expressions, the tools DCL provides for creating and manipulating data. Without some means for making decisions about data, however, there is not much a procedure can do with data other than display it. This chapter presents DCL commands that allow a procedure to make decisions and take different actions based on those decisions. These commands are called **flow-of-control commands** because they are used to control and alter the flow of execution through the procedure. Each time the procedure is run, DCL can execute distinct portions of the procedure based on data obtained or calculated by it.

4.1 Sequential Execution

When you run a DCL procedure with the at-sign (@) command, DCL executes the commands in the procedure one at a time, in order, from top to bottom. This is called **sequential execution** of commands. Simple procedures can be written using only sequential execution:

```
$! A procedure named SIMPLE-PROC.  
$  
$ show time  
$ directory  
$ show users
```

This procedure first displays the current date and time. It then displays a listing of the files in the working directory. Finally it displays a list of the users logged in to the system.

In order to run this procedure, you use the at-sign command. The name of the procedure file is specified following the at-sign, and DCL runs the procedure:

```
$ @simple-proc
18-MAR-1988 17:54:39:17

Directory LISPW$: [GREEK]

ARCHREQ.TXT;10    BACK-UP-VAXSTATION.COM;1          BASE-DEVO.DIS;3
BASE-DEVOS.DIS;2  BINDECLIB.DIS;4    CHECKSIZE.COM;1  CLCS.DIR;1
.
.
.
```

We say that the at-sign command **runs** or **executes** a procedure. When DCL reaches the end of a procedure file, the procedure is terminated and the DCL prompt appears again. DCL is ready for the next command.

4.2 **The GOTO Command**

Simple procedures can be written using only sequential execution, but flow-of-control commands are needed to construct the majority of useful procedures. The GOTO command is a fundamental flow-of-control command that allows a procedure to alter the normal sequential execution of commands. A GOTO command breaks sequential execution and redirects DCL to another part of the procedure.

Here is a procedure containing a GOTO command:

```
$ show time
$ directory
$ goto another_place
$ show users
$
$another_place:
$ show memory
```

This procedure begins by displaying the time and directory listing. Then DCL encounters the GOTO command, which specifies ANOTHER_PLACE as its destination. The name of the destination of a GOTO command is called a **label** because it labels a particular point in the procedure. When DCL encounters a GOTO, it ceases sequential execution, ignoring the commands immediately following the

GOTO, and takes up execution at the line containing the label. Sequential execution is resumed at the line containing the label. In this example, DCL continues execution at the line labeled ANOTHER_PLACE after displaying the directory. It then displays information about system memory and finally terminates the procedure. The SHOW USERS command is *never* executed.

A GOTO command can specify a label preceding it in the procedure as well as one following it. Here is an example:

```
$ show time
$again:
$ show users
$ goto again
$ show memory
```

After displaying the time, DCL encounters the label AGAIN. DCL ignores labels that it finds as it executes commands sequentially. DCL executes the SHOW USERS command. It then comes to the GOTO command, which specifies the label that appeared earlier in the procedure. DCL suspends sequential execution, locates the label, and resumes execution at the line containing the label. This causes the SHOW USERS command to be executed once more, followed by the GOTO, followed again by SHOW USERS, and so on indefinitely. The procedure is stuck in an **infinite loop** and never terminates. The SHOW MEMORY command is never executed.

A procedure should not contain duplicate labels. When a duplicate label appears in a procedure, it “replaces” the previous label of the same name. DCL remembers the position of the second label and forgets the first one without issuing a warning message. Your procedure may then behave in a strange manner, because GOTO commands might not alter the flow of execution to the expected point in the procedure.

The GOTO command is of little use by itself. All you can do with it is skip commands or cause infinite loops. Some kind of decision-making commands are needed to vary the action of the GOTO command.

4.3

The IF Command

The IF command is the means by which procedures can make decisions. Once a decision is made, a procedure can choose one of two alternative actions. The actions are disjoint; DCL executes one or the other but not both. The ability to make decisions, coupled with the GOTO command's ability to alter the sequential flow of execution, provides all the power you need to solve arbitrarily complex problems with a computer.

In VMS Version 5, there are two forms of the IF command: simple and compound. The compound form is not available in Version 4, so Section 4.3.1 describes what to do in the event you are running VMS Version 4.

The first form of the IF command, the simple form, consists of an expression and a command, as follows:

```
$ if expression then command
```

▷ Ch. 3

The expression must result in a boolean value. If the value is true, then the command following the word THEN is executed. If the value is false, the command is ignored. Nothing else is done with the expression value; its sole purpose is to determine whether or not the command is executed. The expression is called the **condition**, because its true/false condition determines whether the command is executed.

Here is a simple example:

```
$ if f$getsysi("PAGEFILE_FREE") .lt. 50000 then -  
    write sys$output "WARNING: paging space is getting low."  
$ show time
```

The expression in this IF command uses the F\$GETSYSI lexical function to determine whether the amount of free paging file space is less than 50,000 blocks. The expression's value is true if there are fewer than 50,000 blocks, false if there are 50,000 or more. When the value is true, the WRITE command is executed to display the message. When the value is false, the WRITE command is ignored. In either case, sequential execution continues and the time is displayed.

The real power of the IF command emerges when it is used in conjunction with the GOTO command. The following code uses the IF command to skip a group of commands in the event they should not be executed:

```

$   if f$getsysi("PAGEFILE_FREE") .ge. 50000 then goto skip_stuff
$   write sys$output "WARNING: paging space is getting low."
$   show memory
$skip_stuff:
$   show time

```

The expression in the IF command tests the number of free paging file blocks, just as in the previous example. However, instead of testing whether the number is less than 50,000, the condition is inverted to test whether the number is greater than or equal to 50,000. If so, the GOTO command redirects execution to the SKIP_STUFF label. If not, the GOTO command is ignored and *both* the WRITE and SHOW MEMORY commands are executed. This is how a group of commands can be executed or skipped based on an IF test. In either case, sequential execution continues at the SKIP_STUFF label and the SHOW TIME command is executed.

Now that you understand how to use IF and GOTO to conditionally execute a group of commands, it is time to learn how the GOTO can be avoided. Modern programming practice dictates that commands such as GOTO, which allow arbitrary changes to a program's control flow, should be shunned because a program with a lot of GOTOs can easily deteriorate into a complex "bowl of spaghetti," which is difficult to follow. The GOTO command also requires that you constantly devise labels that serve no purpose other than to group statements together. The compound form of the IF command in VMS Version 5 allows a group of commands to be executed when the condition expression is true, without the use of the GOTO command. The previous example can be rewritten as follows:

```

$   if f$getsysi("PAGEFILE_FREE") .lt. 50000
$   then
$       write sys$output "WARNING: paging space is getting low."
$       show memory
$   endif
$   show time

```

When the condition in the IF command is true, all the commands between the THEN and the ENDIF are executed. When the condition is false, the commands are skipped. In either case, execution continues with the command following the ENDIF. Notice how the IF command has reverted to testing for free blocks less than 50,000, a more natural test because the critical condition is "less than 50,000 blocks," not "greater than or equal to 50,000 blocks." Notice also that there are no GOTO commands with their clutter of command labels.

The first command in the group (WRITE) can appear on the same line as the THEN keyword, although the author feels it is better style to use a separate line for the first command. The ENDIF keyword must appear on a line by itself. Note also

that the commands between the THEN and the ENDIF are indented two columns to emphasize that they are grouped together. This is not required, but it helps a reader of the procedure to see where the group begins and ends.

It is often the case that one group of commands is to be executed when a condition is true and an alternative group is to be executed when it is false. The second form of the IF command accommodates this requirement with the ELSE keyword:

```
$ if f$getsysi("PAGEFILE_FREE") .lt. 50000
$ then
$   write sys$output "WARNING: paging space is getting low."
$   show memory
$ else
$   write sys$output "Paging space is fine."
$ endif
$ show time
```

The block of commands between the ELSE and ENDIF keywords is only executed if the condition is false. So if the condition is true, the WRITE and SHOW MEMORY commands are executed, and if it is false, the single WRITE command is executed to display "Paging space is fine.". As usual, regardless of whether the condition is true or false, execution continues with the SHOW TIME command following the ENDIF.

The two forms of the IF command provide the capability to conditionally execute single commands or groups without the need for any GOTO commands.

4.3.1

The IF Command Prior to VMS Version 5

The compound form of the IF command, which allows groups of commands to be executed as one unit, was not available until VMS Version 5. If a version prior to Version 5 is being used, the compound IF command must be constructed with simple IF and GOTO commands. Here is the example of an IF with a group of commands to be executed when the boolean expression is true:

```
$ if f$getsysi("PAGEFILE_FREE") .ge. 50000 then goto 19
$   write sys$output "WARNING: paging space is getting low."
$   show memory
$19:
$   show time
```

The WRITE and SHOW MEMORY commands are to be executed when there are fewer than 50,000 free paging file blocks. The IF expression is arranged so that the GOTO will skip the commands when there are enough blocks. The label

19 marks the end of the commands that are conditionally executed, and it serves as the target of the GOTO.

Here is the example with one group of commands to be executed when the condition is true, another when it is false:

```
$   if f$getsysi("PAGEFILE_FREE") .ge. 50000 then goto 15
$     write sys$output "WARNING: paging space is getting low."
$     show memory
$     goto 19
$15:
$     write sys$output "Paging space is fine."
$19:
$     show time
```

In the previous example, the GOTO command specified label 19, the end of the entire IF construct. In this example, it specifies label 15, which marks the beginning of the commands to be executed when the IF expression is false. In addition, a new GOTO command is required after the SHOW MEMORY command in order to skip around the second WRITE command. Here is the overall flow of control depending upon the number of free paging file blocks:

Less than 50,000. The condition is false, so the GOTO is ignored. The WRITE and SHOW MEMORY commands are executed. The GOTO 19 command skips to the end of the IF construct.

Greater than or equal to 50,000. The condition is true, so the GOTO is performed. Execution continues at label 15 and the "Paging space is fine." message is displayed. Execution falls through to the end of the construct.

The author used the following labeling convention for an IF construct prior to VMS Version 5: The end of the construct is marked with a numeric label equal to a multiple of ten plus nine. If the construct has an "else" alternative, it is assigned a label equal to the same multiple of ten plus five. The example uses 15 and 19. This labeling scheme is consistent with the one presented for loops.

4.4

Loops

A **loop** is a sequence of commands that is repeatedly executed until some termination condition arises. Section 4.2 described how to set up an infinite loop, one that never terminates. This is rarely useful because the command procedure executes forever. A loop that eventually terminates must check for a termination condition and stop looping when the condition is met. Each execution of the commands in a loop is called an **iteration**.

Here is a loop that iterates exactly ten times:

```
$    count = 0
$10: count = count + 1
$    if count .gt. 10 then goto 19
$    show users
$    goto 10
$19:
```

The symbol COUNT is used to maintain a count of the number of iterations. It is initialized to zero and incremented at the beginning of each iteration. The loop is terminated when the counter exceeds ten. The SHOW USERS command is therefore executed exactly ten times. Here is what happens on the first iteration:

1. The value of COUNT is incremented from 0 to 1.
2. The value of COUNT is compared to 10. Since it is less, the condition is false and the GOTO 19 command is ignored.
3. The SHOW USERS command is executed.
4. The GOTO 10 command redirects execution back to the beginning of the loop.

In contrast, this is what happens on the eleventh iteration:

1. The value of COUNT is incremented from 10 to 11.
2. The value of COUNT is compared to 10. Since it is greater, the condition is true and the GOTO 19 command is executed. It redirects execution to the line labeled 19, thus terminating the loop.

The author uses the following labeling convention for a loop: The beginning of the loop has a numeric label that is a multiple of ten. The line following the loop has a numeric label equal to the beginning label plus nine. The example uses 10 and 19. In addition to the labeling convention, the commands within the loop are indented two columns. These commands are called the **loop body**.

There are many ways to control loop termination other than with a simple counter. Assume we have a character string in the symbol `LINE` and we want to remove all the spaces from the string. The following loop accomplishes this task:

```
$10:  if f$locate(" ",line) .eq. f$length(line) then goto 19
$      line = line - " "
$      goto 10
$19:
```

▷ Ch. 3

The `F$LOCATE` lexical function takes two arguments, a pattern string and a target string. It scans the target string for an occurrence of the pattern and returns the index of the pattern in the target. If the pattern does not occur in the target, the length of the target string is returned. In this example, the `F$LOCATE` function is used to determine whether the line contains any spaces. If not, the loop is terminated. If so, the leftmost space is removed from the line with the string reduction operator (`-`) and the loop is repeated. Eventually, all spaces will be removed and the loop will terminate.

▷ Ch. 3

The preceding loop could also be written as follows:

```
$10:  line = line - " "
$      if f$locate(" ",line) .eq. f$length(line) then goto 19
$      goto 10
$19:
```

It is perfectly fine to write the loop this way, because the string reduction operator is harmless if there is no space in the line. The loop removes a space if there is one and then checks the line to see if it is free of spaces. If not, the loop is repeated. By reorganizing the loop this way, we push the termination test to the bottom of the loop. This allows us to combine the terminating `IF` command with the final `GOTO` command:

```
$10:  line = line - " "
$      if f$locate(" ",line) .ne. f$length(line) then goto 10
$19:
```

In this case, the `IF` command determines whether there are any more spaces in the line and redirects execution to the beginning of the loop if there are. When all spaces have been removed, the `IF` command “falls through” to the label following the loop and the loop is terminated. The label `19` is no longer the target of any `GOTO` commands and is therefore technically unnecessary, but it is left as a visual indicator of the end of the loop.

A loop may require more than one termination test. The following loop finds the first parenthesis in a string:

```
$ i = -1
$10: i = i + 1
$ if i .ge. f$length(line) then goto 19
$ if f$extract(i,1,line) .eqs. "(" .or. -
    f$extract(i,1,line) .eqs. ")" then goto 19
$ goto 10
$19:
```

▷ Ch. 3

The symbol `I` is used as an index into the string. It is initialized to 1 and incremented each time through the loop. Remember that the characters in a string are indexed beginning with zero. Two termination tests are required. The first one checks to see if `I` has become too large to index the string. If the first check does not terminate the loop, then `I` can be used as a string index. The second `IF` checks to see if the character indexed by `I` is an open or a close parenthesis and terminates the loop if so. If neither test terminates the loop, another iteration is started.

The condition of the second `IF` command in the preceding example is more complicated than in previous `IF` commands. The boolean expression contains the `.OR.` operator. Because of the relative precedence of the `.EQS.` and `.OR.` operators, the string comparisons are performed first. The first comparison produces a true value if the character is an open parenthesis, false if not. The second comparison produces a true value if the character is a close parenthesis, false if not. The result of the `.OR.` operation is true if either comparison was true, false if both were false. In other words, the final result is true if the character is a parenthesis, false if not.

4.5 **Invoking a Command Procedure**

Chapter 2 presented a cursory overview of the `at-sign` command and how it is used to run a command procedure. This section describes the `at-sign` command in greater detail. When you enter a command at the `DCL` prompt, `DCL` reads the command from the keyboard and executes it. If the command is an `at-sign` command, `DCL` stops reading commands from the keyboard and instead reads them from the command procedure file specified following the `at-sign`. Once it has read all the commands from the procedure, `DCL` resumes reading commands from the keyboard. At first, this may seem like a trivial concept, but it is really quite a powerful one.

We use the word *level* to refer to DCL's source of commands. The DCL prompt is at **prompt level**, and a procedure invoked from prompt level is said to run at **procedure level 1**. It is perfectly permissible for a procedure running at level 1 to invoke another procedure, which would then be running at **procedure level 2**. Here is an example:

```
$! This is procedure A.  
$  
$ show time  
$ @b  
$ show time  
  
$! This is procedure B.  
$  
$ show users
```

If you invoke procedure A from the DCL prompt, it runs at procedure level 1. After displaying the current time, it invokes procedure B with the at-sign command. Procedure B then runs at procedure level 2, displays a list of the current users, and terminates. When B terminates, execution continues in procedure A with the line following the at-sign command (the second `SHOW TIME`). The time is displayed again, A terminates, and the DCL prompt reappears. The important point is that DCL remembers the line in procedure A where execution must continue after B terminates. This place is called the **return point**. DCL can keep track of the return points for up to 32 procedure levels, more than most applications ever require.

In the jargon of programming languages, a procedure invoked by another procedure is called a **subprocedure** or **subroutine**. The at-sign command provides the DCL programmer with a straightforward subroutine capability.

4.5.1 **Parameters**

In order to create a procedure that can vary its actions from one use to the next, there must be a way for the user of the procedure to provide it with data and control information. For example, a procedure that deletes or purges all the files with the file type `TMP` must be told whether it should delete or purge. There are many ways to provide a procedure with information; one of the most common is called the procedure **parameter**. A procedure parameter is an item of information passed to the procedure when it is invoked with the at-sign command. You can pass as many as eight parameters to a procedure.

Here is a procedure that can delete or purge all of the TMP files in the working directory:

```
$! Procedure CLEAN-UP.  
$! First parameter is the word DELETE or PURGE.  
$  
$ if p1 .eqs. "DELETE" then delete *.tmp;*  
$ if p1 .eqs. "PURGE" then purge *.tmp
```

When this procedure is invoked, it is provided with a parameter that specifies whether temporary files should be deleted or purged. There are two ways to invoke it:

```
$ @clean-up delete
```

— or —

```
$ @clean-up purge
```

The parameter is specified after the procedure name and must be separated from it by one or more spaces. Before DCL begins to execute the procedure, it creates a special procedure-level symbol named P1 whose value is the specified parameter. The parameter is converted to uppercase letters unless it is enclosed in double quotes. The value of the P1 symbol is always a character string, even if the parameter is an integer. When CLEAN-UP is invoked with the first command, P1 has the value "DELETE". When it is invoked with the second command, P1 has the value "PURGE". The IF commands in the procedure compare the value of P1 against two literal strings to decide which function to perform.

You can specify up to eight parameters to a procedure. They are stored in the symbols P1, P2, and so on, up through P8. There are always eight symbols regardless of the number of parameters specified. Those symbols that have no corresponding parameter are set to the null string. Do not use these special symbols for any purpose other than to access the parameters. The symbols are procedure-level symbols and so are deleted when the procedure exits.

▷ Ch. 3

Spaces are used to separate the parameters in an at-sign command. If a single parameter contains spaces or slashes (/), it must be enclosed in quotation marks. The quotation marks group all the enclosed characters into one parameter. Here is a procedure that accepts two messages, displays them, and remembers them in two global symbols:

```

$! Procedure REMEMBER
$! The first parameter is one message.
$! The second parameter is another message.
$
$ write sys$output p1
$ write sys$output p2
$ save_message1 == p1
$ save_message2 == p2

```

The two messages are passed as parameters 1 and 2 and so are obtained by the procedure using symbols P1 and P2. Both messages are displayed and then saved in the symbols SAVE_MESSAGE1 and SAVE_MESSAGE2. The procedure is invoked with a command such as

```
$ @remember "go to the grocery store" "get a video tape"
```

Because the messages contains spaces, they must be enclosed in quotation marks. This makes it clear that the first message is "go to the grocery store" and the second one is "get a video tape". If the procedure is invoked without the quotation marks:

```
$ @remember go to the grocery store get a video tape
```

then each word appears to be a separate parameter, because parameters are separated by spaces. In this case, there would be nine parameters, more than are allowed by DCL.

A procedure parameter must be enclosed in quotation marks in the following circumstances:

- When the parameter contains spaces or slash characters.
- When the parameter contains lowercase letters that must not be converted to uppercase.

4.5.2

The EXIT Command

When DCL runs out of commands in a procedure file, it automatically terminates the procedure. If the procedure was invoked from the DCL prompt, the prompt reappears and you can enter another command. If the procedure was invoked from another procedure, the original procedure resumes execution with the command following the at-sign command. The termination of a procedure is called **procedure exit**. The EXIT command can be used to cause procedure exit before DCL gets to the bottom of the procedure.

Here is the CLEAN-UP procedure with an EXIT command:

```
$! Procedure CLEAN-UP.
$! First parameter is the word DELETE or PURGE.
$
$ if p1 .nes. "DELETE" .and. p1 .nes. "PURGE"
$ then
$     write sys$output "The parameter must be DELETE or PURGE."
$     exit
$ endif
$
$ if p1 .eqs. "DELETE" then delete *.tmp;*
$ if p1 .eqs. "PURGE" then purge *.tmp
```

The new IF command checks the parameter to make sure it is either "DELETE" or "PURGE". If not, an error message is displayed and an EXIT command is executed: the procedure terminates immediately. If the parameter is valid, the procedure continues with its task.

The preceding example illustrates a common use for the EXIT command. Procedure parameters are checked at the beginning of the procedure to ensure they are valid. If not, the EXIT command is employed to terminate the procedure before it begins its real work. Be careful not to overuse the EXIT command; a person reading your procedure has to be particularly aware of each EXIT command because it stops the natural flow of execution from top to bottom.

4.6 **The CALL Command**

The CALL command provides a subroutine capability similar to the at-sign command, except that the subroutine resides in the same procedure file as the command that invokes it. The CALL command was introduced in VMS Version 5. Here is a procedure that uses the CALL command:

```
$ call show_disks "Before the program:"
$ run sys$system:application-pgm
$ call show_disks "After the program:"
$ exit
$
$show_disks:
$ subroutine
$ write sys$output p1
$ show time
$ show devices/mounted
$ exit
$ endsubroutine
```

The CALL commands are used to invoke a subroutine named SHOW_DISKS. Parameters can be passed to the subroutine just as if it were a separate procedure invoked with the at-sign command. In this example, the subroutine is called twice, the first time with the parameter "Before the program:" and the second time with the parameter "After the program:".

The subroutine begins with a label specifying its name, followed by the DCL command SUBROUTINE. This command marks the start of the subroutine, which includes all the commands down through the corresponding ENDSUBROUTINE command. When the subroutine is invoked with CALL, a new procedure level is established, new procedure-level symbols P1–P8 are defined, and the commands in the subroutine are executed. When the ENDSUBROUTINE is reached, the subroutine is terminated and execution continues after the CALL command. As with procedures, an EXIT command can be used to terminate the subroutine at any point during its execution.

If DCL runs into a SUBROUTINE command as it executes a procedure sequentially, all the subroutine's commands are skipped and execution continues after the ENDSUBROUTINE. The only way to execute the commands in a subroutine is to call it. Therefore, the first EXIT command in the preceding procedure is technically unnecessary. DCL would skip over the subroutine as it executed the main portion of the procedure, reach the end of the procedure file, and terminate the procedure. However, the use of the EXIT command in this circumstance makes the procedure easier to read and follows our convention that an explicit EXIT command is used at the end of every procedure.

Table 4.1 summarizes the similarities and differences between the at-sign and CALL commands.

A disclaimer: when the term *subroutine* is used in this book, it does not necessarily refer to a subroutine invoked with the CALL command. The term is used in a general way to signify any sequence of commands invoked by a DCL procedure, whether they are in the same procedure file or another one, and regardless of how they are invoked. Subroutines therefore encompass procedures invoked with the at-sign command, subroutines invoked with CALL, and those invoked with GOSUB.

Table 4.1 Summary of Subroutine Facilities

Feature	At-sign	CALL	GOSUB
Location of subroutine	Separate procedure	Same procedure	Same procedure
Creates new procedure level?	Yes	Yes	No
Pass P1–P8 parameters?	Yes	Yes	No
Implicit termination	End of procedure	ENDSUBROUTINE	End of procedure
Explicit termination	EXIT	EXIT	RETURN
Can alter caller's local symbols?	No	No	Yes
Can return values in global symbols?	Yes	Yes	Yes

4.7

The GOSUB Command

The GOSUB command is similar to the CALL command. It invokes a subroutine that resides in the same procedure as the caller. However, there are some important differences, summarized in Table 4.1. In particular, GOSUB does not create a new procedure level and there is no way to pass the P1–P8 parameters to the subroutine. The subroutine is limited to performing relatively simple operations and must receive its input data in symbols created by the caller. Here is the CALL example modified to use GOSUB:

```

$ heading = "Before the program:"
$ gosub show_disks
$ run sys$system:application-pgm
$ heading = "After the program:"
$ gosub show_disks
$ exit
$
$show_disks:
$ write sys$output heading
$ show time
$ show devices/mounted
$ return

```

The GOSUB command requires one parameter, the label of the subroutine to be invoked. The subroutine itself contains no special surrounding commands like SUBROUTINE and ENDSUBROUTINE. It performs a series of commands and then

terminates with the RETURN command. Because there is no way to pass parameters to the subroutine, it must receive the heading string in the symbol HEADING. The symbol acts as a parameter for the subroutine, so the main procedure must set it before invoking the subroutine. The EXIT command is required in the main procedure to prevent DCL from falling through and executing the subroutine's commands. Because the subroutine is not delineated by commands such as SUBROUTINE and ENDSUBROUTINE, DCL does not know to skip the subroutine when it is encountered during sequential execution.

In effect, a subroutine invoked with GOSUB is executed exactly as if it appeared in its entirety in the place of the GOSUB command. A subroutine invoked with GOSUB can itself invoke another subroutine with GOSUB. For each procedure level, DCL can remember the return points for up to 16 GOSUB levels.

Chapter 5



Substitution

Up to this point, the book has focused on DCL language features that are similar to those found in conventional languages. The similarity fades, however, with the introduction of a powerful feature called **substitution**. The substitution facility is akin to macros in assembly language or preprocessor definitions in the C language. However, substitution in DCL is unrelated to the syntax of commands: it is purely textual replacement. Any part of a command, from a single character to a portion of a file specification to an entire command line, can be created on the fly by the appropriate use of substitution. This is an extremely potent feature, which allows you to create DCL commands “customized” for particular circumstances. It also grants you the power to create malformed or completely garbled commands.

DCL provides three kinds of substitution: apostrophe substitution, implicit substitution, and ampersand substitution. The principle underlying all three kinds is that a symbol or lexical function appearing in a command is removed and replaced with its value. The differences among the three kinds of substitution lie in how they are specified and when they occur.

5.1 Apostrophe Substitution

This section describes **apostrophe substitution**, which is one form of explicit substitution. Using an apostrophe ('), you explicitly request that DCL perform a substitution where it would not otherwise do so. (In contrast, the next section de-

scribes substitution that DCL performs automatically.) Apostrophe substitution occurs after a command is read but before it is analyzed and executed. Therefore, DCL performs simple textual replacement, independent of the format of the command. Once a substitution has been performed, DCL rescans the entire command, including the new portion, for additional substitutions of any type.

An apostrophe substitution is requested by enclosing a symbol or lexical function in apostrophes, as follows:

```
$ type 'file_spec'
```

If the value of FILE_SPEC is the string "FOO.BAR;1", the following command results:

```
$ TYPE FOO.BAR;1
```

The apostrophes and the symbol they enclose have been removed and replaced with the value of the symbol. Here is an example with a lexical function:

```
$ type 'f$environment("PROCEDURE")'
```

The F\$ENVIRONMENT lexical function returns the file specification of the current command procedure. If the procedure is named TEST.COM;1 and resides in directory \$DISK1:[ROBBINS], the following command results:

```
$ TYPE $DISK1:[ROBBINS]TEST.COM;1
```

The apostrophe was chosen as the substitution indicator because it serves no other purpose in DCL. However, apostrophes are common inside string literals, particularly in contractions or quotations. For this reason, *two* apostrophes are required to perform substitution inside a string literal:

```
$ @log_line "The file ''file_spec' was typed."
```

After substitution:

```
$ @LOG_LINE "The file $DISK1:[ROBBINS]TEST.COM;1 was typed."
```

Notice that substitution requires only a single trailing apostrophe regardless of whether it is specified outside or inside a string. This trailing apostrophe is quite interesting. The “official” syntax for substitution calls for one trailing apostrophe; however, it is unnecessary in many cases. In fact, it is only required in the following situations:

- When the character following the symbol could be construed as part of the symbol (i.e., the character is alphanumeric, a dollar sign, or an underscore). This situation does not arise with the substitution of lexical functions, because the lexical function ends with a close parenthesis.
- When another apostrophe substitution immediately follows. If the trailing apostrophe were not present, the leading apostrophe of the second substitution would be mistaken for the missing trailing one.
- When the substitution is specified as the last thing in a string literal. In this situation, a trailing apostrophe is required for both symbol and lexical function substitution.
- When the character following the symbol is a grave accent (`), tilde (~), percent sign (%), ampersand (&), curly brace ({}), backslash (\), or vertical bar (|). This is because DCL “accidentally” treats these characters as part of the symbol (the author believes this was unintentional).
- When the character following the symbol is a space, and the next character is a sharp sign (#), circumflex (^), close parenthesis [)], plus sign (+), equal sign (=), close square bracket (]), comma (,), question mark (?), slash (/), or greater than sign (>). The author does not know the reason for this behavior. These last two situations do not arise in practice very frequently.

Here is an example of each of the situations:

```
$ type 'prefix'_table.dat
-or-
$ type 'file_name''file_type;
-or-
$ @log_line "Done with file ''f$parse(file_spec)'"
-or-
$ @log_line "It took ''cpu_pc'% of the CPU."
-or-
$ @log_line "Number of ''what' = ''count."
```

Trailing apostrophes will be used in this book only when necessary, because the author believes this convention improves the readability of command procedures, even though it means that the programmer must remember the foregoing rules. If you find the rules confusing, by all means specify a trailing apostrophe for all substitutions.

5.1.1

Substituting Expression Values

With the apostrophe, it is easy to substitute the value of a symbol or lexical function. It is a little trickier to substitute the value of an arbitrary expression. This is accomplished with two lexical functions. The `F$INTEGER` function takes an arbitrary expression and returns its value, converted to an integer if necessary. The `F$STRING` function takes an arbitrary expression and returns its value, converted to a string if necessary. These functions allow the value of an arbitrary expression to be inserted in a command.

Assume the symbol `COUNT` is set to the number of people attending a future meeting. This command prints two copies of the agenda for each person:

```
$ print/copies='f$integer(count*2) agenda.txt
```

When `COUNT` is 3, the following command is executed by DCL:

```
$ print/copies=6 agenda.txt
```

If you did not use the lexical function and instead just wrote `'COUNT*2`, the multiplication would not occur and the following command would result:

```
$ print/copies=3*2 agenda.txt
```

Unfortunately, an expression is not a legal value for the `/COPIES` qualifier (or any other qualifier, for that matter).

5.1.2

Common Mistakes

Programmers are often confused about the context in which apostrophe substitution is required, and this results in overuse of the construct. Substitution is difficult to comprehend and debug, particularly when used in string literals, and therefore should be avoided when not necessary. The primary point to keep in mind is this: when DCL allows an **expression** in a given context, explicit substitution is normally unnecessary because DCL evaluates all symbols and lexical functions as part of evaluating the expression. DCL allows expressions in the following contexts:

- On the right-hand side of an assignment statement (using `=` or `==`, but not `:=` or `:=:`).
- In the square brackets on the left side of an assignment statement that performs substring or bit-field replacement.
- In the `DEPOSIT`, `EXAMINE`, `EXIT`, `IF`, `RETURN`, and `WRITE` statements.

- As an argument to a lexical function, regardless of where the function itself is used.

The following pairs of lines show commands as they might be entered with unnecessary substitution and the same commands without the substitution:

```
$ string1 = ''string2''
```

```
$ string1 = string2
```

–or–

```
$ buffer['index,1] := " "
```

```
$ buffer[index,1] := " "
```

–or–

```
$ if 'count + 1 .gt. 10 then goto 19
```

```
$ if count + 1 .gt. 10 then goto 19
```

–or–

```
$ delete 'f$parse("''spec'', ".tmp;*)
```

```
$ delete 'f$parse(spec, ".tmp;*)
```

5.2

Implicit Substitution

After DCL performs all the explicit substitutions specified by apostrophes, it performs a single implicit substitution. This substitution is often called **automatic substitution** or **personal command substitution**. DCL analyzes the beginning of the command line and determines whether the first item is a symbol. If so, it replaces the symbol with its value. Implicit substitution is done at the beginning of the command analysis phase and thus occurs after apostrophe substitution.

All the following conditions must be met for DCL to carry out an implicit substitution:

- The command must not be an assignment statement. Implicit substitution is never performed in assignment commands, because it would render the assignment command useless.
- The first item in the command must be a symbol.
- The symbol must not be a label.
- The symbol must have a value.
- The value cannot be hidden with the SET SYMBOL/SCOPE command.

▷ Ch. 7

Once an implicit substitution is performed, DCL does *not* rescan the new portion of the command. This prevents recursive substitutions, as we shall see.

5.2.1

Personal Commands

One of the most common uses for implicit substitution is in the definition of **personal commands**. A personal command is a synonym for a command verb or a contraction for all or part of a complete command. Using personal commands, you can build up a collection of customized DCL verbs tailored to your working environment. When a personal command is defined in the login command procedure, it is available for use immediately.

The following personal commands illustrate some of the possibilities:

```
$ backup == "backup/log"
$ deletedir == "@sys$login:deletedir"
$ notices == "type sys$system:notice.txt"
$ send == "mail"
$ sp == "set process/privilege="
```

Assume these commands were defined in the login procedure. The following pairs of lines show a command as it might be entered and the resulting command executed by DCL.

```
$ backup *.*; msa0:save.bck/saveset/rewind
$ BACKUP/LOG *.*; MSAO:SAVE.BCK/SAVESET/REWIND

-or-

$ deletedir work
$ @SYS$LOGIN:DELETEDIR WORK

-or-

$ notices
$ TYPE SYS$SYSTEM:NOTICE.TXT

-or-

$ send agenda.txt sally_smith /subject="Tomorrow's agenda."
$ MAIL AGENDA.TXT SALLY_SMITH /SUBJECT="Tomorrow's agenda."

-or-

$ sp (oper,sysprv)
$ SET PROCESS/PRIVILEGE= (OPER,SYSPRV)
```

The first example illustrates why DCL does not rescan the new portion of a command after implicit substitution. If it did rescan, the substitution process would never terminate, because the personal command BACKUP is again present in the command line after each substitution.

5.2.2

Personal Commands in Procedures

Personal commands and DCL procedures interact in two important ways. Command combinations that are frequently used in the procedure can be assigned synonyms during procedure initialization. This assists in making the procedure more readable. On the other hand, personal commands that were defined outside of the procedure can wreak havoc on the procedure's execution. Predefined personal commands must be disabled while the procedure is running.

There are a few commands that are so common that synonyms are useful in virtually every command procedure of any complexity. The examples in the rest of this book rely on the following personal command definitions, which are assumed to be defined in every main procedure. Additional personal commands can be defined according to the requirements of the application. Do not abbreviate commands used in procedures; this will just confuse the future reader.

```
$ define = "define/nolog"
$ inquire = "inquire/nopunctuation"
$ display = "write sys$output"
$ undefine = "deassign"
```

When you invoke a procedure from the DCL prompt, all the personal commands defined outside the procedure are available to the procedure. Rather than being a help, this feature is extremely dangerous. Imagine what would happen if personal commands like the following were defined and then mistakenly employed by an "unsuspecting" procedure.

```
$ backup = "backup/rewind" ! Rewind tape before using.
$ delete = "delete/confirm" ! Ask user before deleting.
$ mount = "mount/assist" ! Do operator-assisted mounts.
$ rename = "rename/log" ! Log renames to terminal.
```

The first definition, for example, would cause a backup saveset to be placed at the beginning of the tape volume, overwriting any data previously recorded. This is not the default behavior of the BACKUP utility, and it is extremely dangerous. One of the first things you should do in a main procedure is to prevent DCL from considering personal commands defined outside of the procedure. Unfortunately, there is no straightforward way to accomplish this. You must define a personal

▷ Ch. 7

command for each DCL command used in the procedure, thus “hiding” any personal commands defined outside the procedure.

5.3

Ampersand Substitution

A second form of explicit substitution is called **ampersand substitution**. Using an ampersand (&) you can request that DCL perform a limited kind of substitution, one that replaces a symbol with its value. Ampersand substitution takes place after apostrophe and implicit substitution but before the command is executed. This second form of explicit substitution was intended to be used in concert with apostrophe substitution in certain rare instances where double replacement is necessary.

▷ Ch. 22

Double replacement is most often useful when simulating arrays with symbols. A simple example is presented here. Imagine a procedure that accepts multiple parameters, each one a file specification. The procedure displays the contents of the files at the terminal:

```
$ i = 0
$10: i = i + 1
$ if i .gt. 8 then goto 19
$ if p'i .eqs. "" then goto 19
$ type &p'i ! Uses both forms of substitution.
$ goto 10
$19:
```

The interesting line is the one invoking the TYPE utility. Assume the value of I is 3 and the third parameter is NAMES.TXT. After apostrophe substitution, the command becomes TYPE &P3. After ampersand substitution, the final command is TYPE NAMES.TXT. The loop types all the files, terminating after the eighth one or as soon as a null parameter is encountered.

A command containing an ampersand substitution can always be replaced by two commands with an apostrophe substitution in each one. For example, the critical line in the preceding example can be replaced by

```
$ file = p'i
$ type 'file
```

Ampersand substitution should be avoided, for two reasons. First, its use results in rather inscrutable code, particularly for the majority of people who are unfamiliar with it. Second, ampersand substitution is subject to the following restrictions and idiosyncracies:

- An ampersand may only be used with a symbol, not with a lexical function.
- Because there is no trailing delimiter, the character following the symbol must not be a symbol character (i.e., alphanumeric, dollar sign, underscore).
- The ampersand is ignored in the at-sign (@) and CALL commands.
- The ampersand is ignored in string literals, even if doubled.
- Ampersand substitution is performed after the command is converted to uppercase letters. This can have startling results. For example, if the value of the symbol being substituted is another symbol name that happens to be in lowercase letters, DCL will not recognize the second symbol.
- If verification has been enabled with SET VERIFY, the command line is traced before ampersand substitution is performed.

Chapter 6



Displaying Output

This chapter discusses methods for displaying output during the execution of a procedure. The word *display* is used because today most interactive users have access to video display terminals rather than hardcopy terminals. DCL provides facilities for displaying text and simple graphics on video terminals. Such facilities are used in procedures to present results, status information, and error messages.

The user's terminal has both an input and an output capability. DCL treats these two functions of a terminal as separate devices: an input device, which receives keystrokes from the keyboard, and an output device, which displays characters on the video screen. When a user logs in, VMS creates a process to execute programs on behalf of the user. Associated with this process is a **process-permanent file**, which can be used to write to the video screen. This output file is assigned the logical name `SYS$OUTPUT` and is always available. Text is displayed on the terminal by writing to `SYS$OUTPUT`.

▷ Ch. 20

6.1 *Displaying Text*

Text is displayed with the `WRITE` command. This command takes two arguments. The first is a logical name referring to the file that is to receive the output (note that it is a logical name, not a symbol). The second is an expression or list of expressions whose values are to be written. When you want to display text on the terminal, use the logical name `SYS$OUTPUT`:

```
$ write sys$output "Greetings from the land of procedures."
```

-or-

```
$ write sys$output "The value of X is: ", x
```

-or-

```
$ write sys$output "Your name is ", name, -  
                    "and your age is ", age, "."
```

-or-

```
$ write sys$output "The average is: ", (v1+v2)/2
```

The last example shows that the values can be complex expressions, not just string literals or symbols. Because the WRITE command accepts expressions, apostrophe substitution is not usually required to produce the values to write.

Procedures often write to the terminal, so it is handy to set up a short personal command that can be used in place of WRITE SYS\$OUTPUT. This is done during procedure initialization:

```
$ display = "write sys$output"
```

This leaves more space on the line to include lengthy expressions:

```
$ display "Your name is ", name, "and your age is ", age, "."
```

6.2 **Terminal Control Sequences**

All modern video terminals accept **control sequences**, which specify video operations above and beyond the display of simple text. Control sequences can be used to clear the screen, change the video rendition to boldface or reverse video, draw lines, and so forth. Using the WRITE command, you can send control sequences to the terminal.

A control sequence is a series of ASCII characters beginning with a control sequence initiator. On eight-bit terminals, such as the VT200 and VT300 series, the initiator is the CSI character (hexadecimal 9B). On seven-bit terminals, such as the VT100 series, the initiator is two characters, an ESC character (hexadecimal 1B) followed by the open square bracket ([). Because older terminals use seven-bit characters and thus initiate a control sequence with an escape character, control sequences are often called **escape sequences**.

In this book, seven-bit control sequences are illustrated because they work on both seven- and eight-bit terminals. Furthermore, control sequences acceptable

to the VT100, VT200, and VT300 family of terminals are used. To set up the control sequence initiator for such a terminal, it is best to use a bit-field assignment command to create a character string containing the ESC character. Do not place an actual ESC character in the procedure file, as this can cause unpredictable behavior if the procedure is typed to the terminal or printed on a line printer. Here is how to set up the control sequence initiator:

```
$   esc[0,8] = %x1B   ! Prepare a string containing an ESC
$   csi = esc + "["   ! followed by an open bracket.
```

Once this is done, control sequences are sent to the terminal by writing the initiator followed by the appropriate additional characters. The complete repertoire of control sequences is described in the user's guide for your particular terminal.

```
$   display csi,"2J"           ! Clear the entire screen.
$   display csi,"1;1H"        ! Position cursor to 1,1.
$   display csi,"0;1m",-      ! Select boldface rendition,
        "A Boldface Heading",- ! display a heading,
        csi,"0m"              ! and reset normal rendition.
```

Some control sequences are used frequently, such as the one that clears the entire screen. Using some additional symbols, you can assign a symbolic name to a complete sequence. This helps avoid errors in the repeated specification of identical control sequences.

```
$   bold   = csi + "0;1m"
$   clear  = csi + "2J"
$   home   = csi + "1;1H"
$   normal = csi + "0m"
```

With these symbols, the DISPLAY commands in the preceding example are simplified and become more readable:

```
$   display clear
$   display home
$   display bold, "A Boldface Heading", normal
```

Table 6.1 F\$FAO Directives

Directive	Description	Argument
!AS	Insert a character string.	String expression
!SL	Insert an integer in decimal.	Integer expression
!/	Begin a new line.	None
!*x	Repeat a character (denoted by <i>x</i>) the number of times specified by the field width.	None
!%T	Insert the current time in VMS format.	The integer 0 (any other value causes an error)
!%D	Insert the current date and time in VMS format.	The integer 0

6.3

Formatting Output

It is often desirable to perform some fancy formatting of text before displaying it. For example, if you are displaying a table containing integers, the integers look best when shown in columns with the low-order digits lined up, that is, right-justified. This is easily accomplished if every integer occupies the same number of columns. DCL provides a lexical function, `F$FAO`, which is used to format text.

The `F$FAO` lexical function provides a moderately powerful facility for text formatting. The acronym `FAO` stands for “formatted ASCII output.” The function accepts multiple arguments, the first being the **control string** and the remainder being expressions whose values are to be formatted according to the control string. In other words, the control string argument acts as a template for the formatted text, while the remaining arguments are the values to be inserted in the template. The `F$FAO` function returns a character string containing the formatted text (it does not actually display the text).

The control string contains fixed text plus directives. The fixed text is placed in the resulting string exactly as it appears. A directive is not placed in the resulting string, but instead is replaced with the corresponding argument value, formatted in accordance with the directive. A directive consists of an exclamation point (!) followed by a one- or two-character code indicating the formatting to be performed. The code, by convention, is specified in uppercase letters. An integer may be included between the exclamation point and the code to specify the width of the formatted field. Table 6.1 describes some basic directives; the complete set is described in the *VMS DCL Dictionary*.

An example is now in order. Assume that a procedure has calculated statistics for a set of numbers. The symbols COUNT, MINIMUM, MAXIMUM, and MEAN contain the statistical results. The goal is to clear the screen and format the statistics as pictured here:

```
Statistics on 18-NOV-1987
-----
```

```
Count:      73
Minimum:    3
Maximum:   485
Mean:      119
```

This is accomplished with the following code:

```
$ display clear, home
$ display f$fao("Statistics on !11%D", 0)
$ display f$fao("!25*-!/")
$ display f$fao("Count: !7SL", count)
$ display f$fao("Minimum:!7SL", minimum)
$ display f$fao("Maximum:!7SL", maximum)
$ display f$fao("Mean: !7SL", mean)
```

Notice that the !SL directive right-justifies the integer in the specified field width.

6.4 Redirecting Program Output

Many VMS utilities and other programs display output under the assumption that they are being used interactively, that is, directly by the user. This assumption fails when the programs are used by procedures as tools to accomplish some part of the application at hand. In this case, the output may be misleading to the user who does not know how the procedure operates internally. Another possibility is that the output must be directed to a file and then processed by a later step of the procedure. In these cases, it is necessary to redirect the displayed output so that it does not appear on the terminal.

Many VMS utilities support the /OUTPUT qualifier on their command line. With this qualifier you can specify the destination for output that would, by default,

be displayed on the terminal. In the following example, the output from the DIRECTORY command is redirected to a scratch file:

```
$ directory/output=sys$scratch:dirout.tmp sys$manager:*.com
```

▷ Ch. 15

Rather than appearing on the terminal, the list of command procedures in the system manager's directory is placed in the file DIROUT.TMP in the system scratch directory. The procedure can then process the file as required.

Some commands do not accept the /OUTPUT qualifier, and many programs are not invoked with a DCL command at all. In these cases, the program's output must be redirected with the DEFINE command:

```
$ define/user_mode sys$output sys$scratch:pgmout.tmp
$ run sys$system:pgm
```

▷ Ch. 14

The DEFINE command redefines the SYS\$OUTPUT logical name to refer to the desired scratch file. The /USER_MODE qualifier makes this redefinition temporary, lasting only for the *next* image execution. Therefore, all output from the PGM program is redirected to the PGMOUT.TMP file, and then SYS\$OUTPUT reverts to the terminal for subsequent programs.

Redefining SYS\$OUTPUT redirects output but does not redirect error messages resulting from errors signaled by the program. This is because error messages are written to both SYS\$OUTPUT and SYS\$ERROR. The process-permanent file SYS\$ERROR is directed to the terminal, so error messages appear there even if normal output does not. If you know that a program is going to produce error messages that the user should not see, you must suppress them. Use a second DEFINE command to do this:

```
$ define/user_mode sys$output sys$scratch:pgmout.tmp
$ define/user_mode sys$error nl:
$ run sys$system:pgm
```

The second DEFINE command redirects the SYS\$ERROR logical name to the null device (NL:). All output to the null device is discarded.

The null device is useful when you want to discard all output and error messages produced by a program. In this case, both `SYS$OUTPUT` and `SYS$ERROR` are redirected to the null device:

```
$  define/user_mode sys$output nl:
$  define/user_mode sys$error nl:
$  run sys$system:pgm
```

All output produced by the PGM program is discarded.

6.5 ***Displaying a File***

This chapter does not describe how to display the contents of a data file, because files have not yet been discussed in detail. Chapter 17 addresses the issues surrounding the display of data files.



The DCL Environment

The review of basic DCL concepts and facilities was completed in the previous chapter. The remainder of the book describes how to use these facilities to create real programs: programs that are useful, correct, and robust. The proposed techniques are directed toward the development of complex procedures, applications composed of multiple procedures, or applications to be used by many people. If you are just writing a 20-line procedure for personal use, don't worry about following every guideline to the letter. Pick and choose the ones that are pertinent to your task.

The term **DCL environment** refers to a collection of information that guides the actions and responses of the DCL command interpreter. The environment information affects the operation of DCL commands, the content of messages displayed by DCL, and the behavior of command procedures. Commands are available for setting and displaying the various items of information making up the environment. This chapter describes the most important aspects of the environment:

- Verifying command procedures.
- Hiding predefined personal commands.
- Setting message components.
- Setting process privileges.

- Setting the user identification code (UIC).
- Changing the default directory.

7.1 Aspects of the Environment

The following sections describe some important aspects of the DCL environment. These environment features will be important in the remainder of this book.

7.1.1 Procedure Verification

As an aid to debugging command procedures, the execution of a procedure can be traced. A procedure trace shows each line as it is executed by DCL, after apostrophe and personal command substitution is performed. The act of tracing a procedure is called **verifying** the procedure. The SET VERIFY command is used to enable and disable verification as needed. Some texts suggest the use of this command to disable verification upon entry to a large procedure. The rationale for this suggestion is that no user could possibly want to see the trace of such a procedure. The author believes it is best not to change the state of verification in a procedure; do not enable or disable it. The user may be pursuing a suspected bug in the procedure, in which case the trace is invaluable.

The SET VERIFY command is discussed in detail in Chapter 12.

7.1.2 Predefined Personal Commands

Chapter 5 discussed the danger inherent in allowing predefined personal commands to affect the execution of a command procedure. For example, if the symbol BACKUP is defined as "BACKUP/REWIND", savesets will be placed at the beginning of a tape volume (overwriting any existing data), even though this is not the default behavior of the utility. There is no simple way to prevent a procedure from using personal commands defined outside it. In the procedure, you must define a new personal command for each command used by the procedure. This new personal command, because it is a procedure-level symbol, will hide any prompt- or global-level symbols of the same name:

```
$ backup = "backup"  
$ copy = "copy"  
$ delete = "delete"
```

Each personal command is defined as the command verb, by itself, with no qualifiers. In other words, each command is defined as itself. When DCL makes a personal command substitution for BACKUP, the result will simply be BACKUP, with no extra qualifiers or parameters to affect its operations. Any definitions of BACKUP outside the procedure are effectively hidden.

It is tedious to define a personal command for every DCL command used in the procedure. What you will probably end up doing is defining a personal command for potentially “dangerous” commands, such as BACKUP, COPY, DELETE, MOUNT. Once a new procedure is near completion, give some thought to which commands need to be safe from interference from personal commands.

DCL does make an attempt to solve the personal command problem with the SET SYMBOL command. This command allows a procedure to circumvent the normal rules of symbol lookup and hide *all* of the prompt- and global-level symbols. Unfortunately, the command also prevents the procedure from creating any new global symbols, something which many large procedures need to do. Any attempt to create a new global symbol results in an error. If you are developing a procedure that does not create any global symbols, you can use the following command near the top of the procedure to hide all existing personal commands:

```
$ set symbol/scope=(nolocal,noglobal)
```

The NOLOCAL keyword specifies that all existing prompt-level symbols and outer procedure-level symbols are ignored by DCL. It is as if they are hidden from the current procedure level. The NOGLOBAL keyword specifies that all existing global symbols are ignored by DCL and that *no new ones can be created*.

The examples in this book assume that all the procedures need to create global symbols; therefore they do *not* use the SET SYMBOL command.

7.1.3

Messages

Messages are an important aspect of the VMS style of user communication. A VMS message consists of four components:

Facility. The name of the utility or application that generated the message.

Severity. The severity level of the message: success, informational, warning, error, or fatal error.

Identification. An identifier for the message, unique among all the identifiers for this facility. The message identifier is used to index the message in documentation.

Text. The actual text of the message, which conveys the information or describes the error.

When VMS displays a message, it normally includes all four components in order:

```
%DCL-I-IVVERB, unrecognized command verb-check validity and spelling
```

The facility is DCL, the severity I (informational), the identifier IVVERB, and the text everything following the comma.

VMS provides a command that allows you to customize the format of the message, omitting components of no interest. A well-written procedure does not produce arbitrary error messages during normal operation, so any abnormal circumstances that do occur can result in unexpected error messages that are out of context (after all, the user did not type the command that produced the error). It is best, then, to display all message components so that the user has the greatest chance of understanding the problem in context and correcting it.

Because you cannot be sure which message components are enabled when the procedure is invoked, it is necessary to enable all of them as part of procedure initialization. Sound modular design also demands that the user's original message settings be restored when the procedure exits. The following two commands will record the current message settings and then enable all components:

```
$ saved_msg = f$environment("MESSAGE")
$ set message/facility/severity/identification/text
```

The first command uses the F\$ENVIRONMENT lexical function to obtain the current message settings and store them in the `SAVED_MSG` symbol. They are automatically stored in the form required by the `SET MESSAGE` command, which

will ultimately reset them. For example, if all components are disabled except the text, the following string will be stored in the symbol: "/NOFACILITY /NOSEVERITY /NOIDENTIFICATION /TEXT". The second command enables all message components.

Upon exit from the main procedure, the original message settings in effect when the procedure was invoked must be restored. This is accomplished with the following command, which reestablishes the saved message settings using the `SAVED_MSG` symbol:

```
$ set message 'saved_msg
```

7.1.4

Privileges

Some procedures, particularly those that perform system management functions, may require that specific privileges be enabled for proper execution. Because you cannot be sure which privileges are enabled when the procedure is invoked, it is necessary to enable the required ones as part of procedure initialization. Again, accepted modular design practice requires that the user's original privilege settings be restored when the procedure exits.

The `F$SETPRV` lexical function is used to enable and disable privileges in a command procedure. This function takes one argument, a list of privilege settings. The user's process privileges are enabled or disabled according to the list (assuming the user is authorized for such privileges), and the *previous* settings of the specified privileges are returned in a similar list. For example:

```
$ prev_privs = f$setprv("OPER")           ! Enable OPER.
-or-
$ prev_privs = f$setprv("SYSPRV,NOBYPASS") ! Enable SYSPRV,
$                                           ! disable BYPASS.
-or-
$ prev_privs = f$setprv("ALL")           ! Enable every
$                                           ! privilege.
-or-
$ prev_privs = f$setprv("NOWORLD,WORLD") ! Enable WORLD (uses
$                                           ! last setting).
```

The following two commands will record all privilege settings in the symbol `SAVED_PRIVS` and then enable some required ones:

```
$ saved_privs = f$setprv("ALL")
$ junk = f$setprv(saved_privs+",OPER,PRMMBX,WORLD")
```

The first line enables *all* privileges and then sets the symbol `SAVED_PRIVS` to their previous state. The user's original privilege settings are therefore recorded in the symbol. The second line creates a new list of privileges, which is identical to the original list but with the addition of `OPER`, `PRMMBX`, and `WORLD`. Privileges are enabled and disabled according to the new list, thus reestablishing the original privilege state except for the required privileges, all three of which are enabled. The value returned by the second `F$SETPRV` is superfluous, so it is stored in the symbol `JUNK`.

There is a good reason to record the state of all privileges and not just the ones being changed. Doing so ensures that all privileges will be reset to their original state, even if some are changed during the execution of the main procedure and its subprocedures. Upon exit from the main procedure, all original privilege settings in effect when the procedure was invoked must be restored. This is accomplished with the following command, which resets the privileges to their saved state:

```
$ junk = f$setprv(saved_privs)
```

7.1.5 User Identification Code

Every VMS process has a user identification code (UIC) associated with it. This UIC determines which objects the process can access and which operations it can perform on those objects. The UIC also determines the owner of a file created by the process: the file will be owned by the UIC of the process unless the owner is otherwise specified. The ways in which the UIC affects file operations are described in detail in Chapter 13.

You should avoid changing the process UIC in a command procedure. Changing the UIC requires the `CMKRNL` (Change Mode to Kernel) privilege, which most users do not have. Once the UIC is changed, the process may be able to access files it normally could not and may not be able to access others. Any files created by the process will be owned by the new UIC, not the UIC normally assigned to the user of the procedure. Always try to design procedures so that they do not need to change the process UIC.

7.1.6

Default Directory

The **default directory** is the directory in which VMS locates and creates files when no other directory is explicitly included in a file spec. Because the word *default* is overused in VMS, this book uses the term **working directory** or **current directory**. The working directory exists as a convenience to the interactive user, so that an explicit directory need not be included in every file spec. The SET DEFAULT command is used to establish a new working directory.

Do *not* change the working directory in a command procedure. Include a device and directory in every file specification unless it is one entered by the user. Leaving the working directory as is and being explicit about directories has the following benefits:

- The user can reasonably expect that file specs entered without a directory refer to the working directory that was current when the procedure was invoked. Many procedures prompt the user for a file; it is disconcerting to the user if the procedure does not operate on the file in the working directory that was current when the procedure was invoked.
- You are forced to consider every file reference and determine where the file will be or where it should be created. For example, scratch files should be created in SYS\$SCRATCH, not in the working directory.

There are, of course, exceptions to every rule. If the primary purpose of a procedure is to operate on files in a particular directory, and if that procedure always prompts the user for files in that directory, then it may be appropriate for the procedure to change the working directory. An example is a procedure that builds a software system in a specific “build” directory. Any procedure that changes the working directory should reset it to the original directory before it exits.

7.2

Putting It All Together

Now that the important environment items have been described, it is time to illustrate the beginning and end of a main procedure. The items that must be set during the procedure (e.g., privileges) are saved and established by initialization code and reestablished by termination code. The items to be left alone (e.g., working directory) are simply not mentioned.

```

$   backup = "backup"      ! Hide dangerous personal commands.
$   copy   = "copy"
$   delete = "delete"
$
$   saved_msg = f$environment("MESSAGE")
$   set message/facility/severity/identification/text
$   saved_privs = f$setprv("ALL")
$   junk = f$setprv(saved_privs+",OPER,PRMMBX,WORLD")
$       .
$       .
$exit:
$   junk = f$setprv(saved_privs)
$   set message 'saved_msg
$   exit

```

The order of initialization is important. Predefined personal commands are hidden first, so that the entire procedure is unaffected by them. The message components are enabled next in order to ensure that all messages, even ones occurring during the remainder of procedure initialization, are displayed in full. Privileges are set last. Termination activities are carried out in reverse order, unfolding the “protective cover” wrapped around the procedure by its initialization.

These code sequences do not take into account the difficulties that arise when the user presses <CTRL/y> during initialization. This topic will be addressed in Chapter 10, after we have discussed interrupt handling.

Chapter 8



Error Handling

Error handling is a critical aspect of every software application. It is sheer folly to assume that everything will go as planned, that no unusual or unforeseen events will occur. Error handling is no less important for DCL applications, particularly since they are often interactive and operate in the highly variable VMS environment. It is impossible to predict every variation in the environment when you write the application.

Error handling is a weak point in many DCL applications. Error handling logic is either missing entirely or suffers from a lack of overall organization. In other words, there is no error-handling scheme to deal with all errors in a consistent and predictable fashion. This chapter presents an overview of the VMS error mechanism and describes how this mechanism is employed by DCL. It then goes on to present an error-handling scheme for DCL applications.

8.1

Status Codes

The **status code** is the means by which a VMS program returns an indication of the success or failure of a requested operation. All programs, be they system services, run-time library routines, utilities, or entire applications, return a status code upon completion. The user or program that requested the operation can determine from the status whether the request succeeded or failed. The status may also indicate the reason for the success or failure.

Table 8.1 Status Code Fields

Bits	Field	Description
0–2	Severity	Severity of the status code (see Table 8.2).
3–14	Message number	Identifies a specific status code for the facility. A facility is limited to 4,096 status codes.
15	Facility specific	This bit is on for a status code that is specific to a facility. It is off for a status code that is shared by multiple facilities.
16–26	Facility number	Identifies the software facility that defines the status code.
27	Customer facility	This bit is on for software facilities produced by third-party vendors or by customers. It is off for facilities produced by DIGITAL.
28	Inhibit display	This bit is set on to inhibit the EXIT command or \$EXIT system service from displaying the message corresponding to the status.
29–31	(reserved)	Reserved for future use by DIGITAL.

8.1.1

Format

A status code is a 32-bit (longword) value that is divided into several bit fields. Table 8.1 describes the fields. Bits 16–27 identify a particular software facility, either supplied by DIGITAL or another source. Bits 3–15 identify a particular status code produced by the facility. Therefore, bits 3–27 uniquely identify every status code defined by every facility. Because a status code contains so many bit fields, it is usually specified in hexadecimal.

▷ Apx. A

The severity of a status code designates whether it represents successful or unsuccessful completion of the facility. There are five severity levels, each described in Table 8.2. The severity levels indicating a successful completion have odd values, while those indicating an unsuccessful completion have even values. This allows a procedure to treat a status code as a boolean value: true if odd (successful), false if even (unsuccessful).

Table 8.2 Status Code Severity Levels

Level	Name	Description
0	Warning	A noncritical error or unusual condition occurred.
1	Success	The program completed normally.
2	Error	The program completed unsuccessfully because of some error.
3	Informational	The program completed normally but with an alternative condition.
4	Severe error	A fatal error terminated the program prematurely.
5-7	(reserved)	Reserved for future use by DIGITAL.

8.1.2

Command Status

Like all programs, VMS commands and utilities return a status code. DCL sets the reserved symbol `$STATUS` to the status code and sets the reserved symbol `$SEVERITY` to its severity. A well-written procedure needs an efficient way to check the status code after every command and must take corrective action if the code indicates an error. There are three methods you can employ to check the status code:

- Rely on a DCL feature that checks the status code automatically and supplies a default error handler.
- Rely on DCL's automatic status checking, but provide your own error handler.
- Disable DCL's automatic status checking, and check the status code explicitly.

These three methods will be described in the following sections.

There are a few DCL commands, such as `IF` and `GOTO`, that do not return a status code when they complete *successfully* and therefore do not always cause the `$STATUS` and `$SEVERITY` symbols to be set. The list of such commands is difficult to remember, so the author recommends that you not rely on this "interesting" but inconsistent behavior. Write the error-handling code in procedures as if every command always returned a status code and set the reserved symbols.

DCL provides a feature whereby a command procedure can return a status code. The EXIT command accepts an expression parameter whose value is the status code to be returned. When an EXIT command is encountered, DCL ceases execution of the current command procedure, evaluates the expression, and returns control to the calling procedure with the status code specified by the expression. When the *main* procedure exits, control is returned to the DCL command level and the dollar sign (\$) prompt appears.

You should use the EXIT command to return a status code from every procedure, both main ones and subprocedures. This allows the methods of status checking described in the rest of this chapter to be used when a procedure is invoked, treating it exactly like a command in that regard. Consistency among commands, utilities, and procedures makes an overall error-handling scheme significantly simpler.

A DCL program requires some application-specific status codes to use with the EXIT command. These codes are returned from procedures to indicate various success or error conditions peculiar to the application. In particular, every application needs a success status code to return from procedures when they complete normally. Status codes are usually defined with the MESSAGE utility, but it was designed to be used with conventional programming languages that are compiled and linked. For DCL procedures, symbols whose values are the status codes must be established during procedure initialization.

Before you can establish status codes, you must choose a customer facility number for your application. For personal applications, you may as well choose the number zero. For other applications, it is best to have a “facility registrar,” who keeps track of all the facilities in your organization and assigns facility numbers. The XDA example application is assigned the facility number 66. During procedure initialization, a basic status code is established from which all others are derived:

```
$ xda__status = %x18428000
```

This status code has a severity and message number of zero. The facility-specific bit is set to indicate that the status code is specific to this application. The facility number is 66 (42 in hexadecimal). The customer facility bit is set to indicate that this is a customer application. The inhibit-display bit is set to prevent the EXIT command from displaying the status.

By convention, the name of a customer facility status code begins with the facility code and two underscores.

With `XDA__STATUS` established, additional status codes are defined by adding the appropriate message number and severity to it. For example, a success status code is always needed. It is assigned message number zero with severity 1:

```
$ xda__success = xda__status + %x0001
```

The hexadecimal number 0001 includes the severity in bits 0–2 and the message number in bits 3–14, as required by the status code format. The value of `XDA__SUCCESS` is `%x18428001`. Now it is possible for a subprocedure to exit with a success status code using the following command:

```
$ exit xda__success
```

8.2 **The Default Error Handler**

After the completion of each command, utility, or procedure, DCL normally checks the returned status code. When the severity indicates an error or a severe error, DCL executes its default **error handler** if no other error handler has been established. The default error handler terminates the current procedure and returns the error status code. Before exiting, the code is modified by setting the inhibit-display bit so that the message associated with the status code is not displayed as part of the procedure exit. This is done under the assumption that the message was already displayed when the command or utility terminated. The default error handler is not executed if the status code contains a success, informational, or warning severity.

The default error handler was designed with typical utilities and applications in mind. Errors and severe errors indicate that some disastrous problem has occurred and that the program has terminated prematurely. Warnings, on the other hand, supposedly indicate a minor problem that did not affect the outcome in any harmful way.

A significant problem with DCL's default error handler is that syntax errors in DCL flow-of-control commands (e.g., `IF`, `GOTO`) result in *warning* messages. When such a warning occurs, the error handler is not invoked and the procedure continues to execute with indeterminable results. There is an exception to this rule in the case of the `GOTO` command. If the specified label does not exist, a warning is issued *but* the procedure exits as if the severity had been error or severe error. For some reason, the original designers of DCL felt that this

particular syntax error, unlike others, was drastic enough to warrant terminating the procedure.

A procedure should detect syntax errors during its execution; such errors indicate significant bugs in the procedure. The default error handler does not do so, and therefore you should replace it with one that does. The following section describes how this is done.

8.3 Changing Error-Handling Behavior

In order to modify DCL's behavior so that it invokes the error handler for warnings in addition to errors and severe errors, you must establish your own error handler. An error handler is established with the `ON` command, which specifies how status codes are to be checked and what action is to be taken when an unsuccessful status is detected. The `ON` command is placed near the top of a command procedure:

```
$ on warning then exit $status .or. %x10000000
```

The use of the word `WARNING` specifies that warnings, errors, and severe errors are to be detected. The command following the keyword `THEN` is the error handler itself; this command is executed when the specified severities are detected. In this case, it is an `EXIT` command, which causes the procedure to exit and return the status code specified by the expression `$STATUS .OR. %X10000000`. This expression takes the status that caused the handler to be invoked (stored in `$STATUS`) and turns on the inhibit-display bit so that the status message will not be displayed as the procedure exits. The `ON` command sets up an error-handling environment identical to the default one, except that warnings are also detected.

The examples in the remainder of this book assume that every procedure contains this `ON` command unless otherwise stated. The error-handling environment established with this command is a simple and robust one: every procedure detects unsuccessful status codes and immediately exits, returning the status to its caller. However, there are variations of the `ON` command that are also useful. The error-handling statement, rather than simply exiting, can perform a `GOTO` to a more complex error handler requiring multiple lines of code. An example of this kind of error handler is presented in Section 8.5.

Checking a Status Code Explicitly

There are times when a procedure must check a status code explicitly and act on it in special ways, rather than relying on the automatic error-handling mechanism. For example, the DIFFERENCE command returns a success status code if two files are identical or an informational code if the files are different. Differentiating the two cases is quite simple, since neither status code will trigger the error handler. You can simply test the \$SEVERITY symbol after the command completes:

```
$ difference old_file.txt new_file.txt
$ if $severity .eq. 3 then action when different
```

Things become more complicated when a command returns a warning or error status code. Assume you want to copy a file from a remote node and need to handle the failure of the copy in a special fashion. In other words, the error handler should not gain control when COPY returns an error status code, but rather the procedure should continue executing in order to check the code explicitly. The following procedure will not work:

```
$ copy remote-file.dat local-file.dat
$ if .not. $status then action upon failure
```

The error status code from COPY will trigger the error handler, which will cause the procedure to exit immediately. It is necessary to disable DCL's automatic checking of status codes, thus allowing the procedure to continue and execute the IF command. This is accomplished with the SET NOON ("set no on") command and its complement, the SET ON command:

```
$ set noon      ! Not SET 12:00 o'clock noon, but SET NO ON.
$ copy remote-file.dat local-file.dat
$ status = $status
$ set on
$ if .not. status then action upon failure
```

The SET commands are wrapped around the sequence of commands that must execute without automatic status checking. It is necessary to save the status code from the copy in another symbol *before* executing the SET ON, otherwise the value of \$STATUS will be replaced by the status code from the SET ON itself. After status checking is reenabled, the status code from the copy can be checked explicitly.

Automatic status checking should be disabled in the shortest possible sequences of commands. Leaving it disabled across many commands can result in undetected errors. Note carefully that `SET NOON` does not prevent error messages from being displayed; it only disables automatic status checking.

8.5 **Handling Errors When Cleanup Is Required**

It is often the case in complex procedures that some **cleanup** must be performed before the procedure exits. For example, a procedure that opens a file should close it before exiting. The cleanup presents no problem when the procedure exits normally; it is simply performed just before the `EXIT` statement. However, if an error occurs during procedure execution, the error handler must also arrange to perform the cleanup. This is not possible if the error handler simply exits, as illustrated in Sections 8.2 and 8.3. The error handler must instead perform a `GOTO` to some code, which eventually executes the cleanup logic.

▷ Ch. 15

The following procedure illustrates the technique for establishing an error handler that includes cleanup. The procedure opens a file, which must be closed before exiting:

```
$ xda__status = %x18428000
$ xda__success = xda__status + %x0001
$ status = xda__success
$ on warning then goto error
  .
  .
$ open/read file 'p1
  .
  .
$ goto exit
$
$error:
$ status = $status
$ goto exit
$
$exit:
$ set noon
$ close file
$ exit status .or. %x10000000
```

Examine this procedure starting at the bottom. Cleanup and exit is performed at the label `EXIT`. A `SET NOON` command is executed to disable automatic checking of status codes. This allows the remainder of the procedure to execute completely regardless of any error that may occur during cleanup. The `CLOSE` command

ensures that the file is closed regardless of whether the procedure exits normally or because of an error. The procedure finally exits with the status code contained in the symbol `STATUS`. Every possible path to the cleanup code must set `STATUS` to a meaningful exit status.

To initialize the procedure, the symbol `STATUS` is set to a facility-specific success status code. This is the code that will be returned by default if no other code is appropriate. The `ON` statement establishes an error handler that detects all errors and jumps to the label `ERROR`.

The procedure opens a file, processes it, and ultimately exits.

If a warning or other error is detected while the procedure is executing, the error handler jumps to the `ERROR` label. This causes the error status code to be placed in the `STATUS` symbol, so that it will be returned by the procedure. The error logic then jumps to `EXIT`, which performs the cleanup and exits with the error status code.

This organization, although a bit complicated, accomplishes the following important goals:

- The cleanup code is executed regardless of the cause of procedure exit. The file is always closed.
- The status code returned by the procedure is either a success code (default) or an error code detected and handled by the error handler. The status code returned by the procedure gives a true indication of what happened.

8.6 Procedure Call Unwinding

A DCL application is often large enough to require more than one command procedure. One of the procedures is the main procedure; it invokes other procedures as subroutines, which in turn may invoke still more procedures. With such nested flow of control, an overall error-handling scheme must be devised to deal with errors at all levels in the procedure hierarchy. The simplest such scheme is called **procedure call unwinding**. The idea is simple: when an error occurs in a subprocedure, exit the subprocedure and all intervening subprocedures until the main procedure is reached. The main procedure's error handler will then deal with the error.

This scheme is straightforward to implement. If you follow the rules set forth in the preceding sections, procedure call unwinding will occur automatically. The rules are as follows:

- Every procedure must return a success or informational status code when it completes normally.
- Every procedure must have an error handler that exits the procedure. It might have to perform some cleanup first.
- The error handler must return the error status code that caused it to be invoked, thus propagating the error status up one procedure level to the caller.

Procedure call unwinding is illustrated by three cooperating procedures, as follows:

```
$! Procedure MAIN.  
$  
$ xda__status = %x18428000  
$ xda__success = xda__status + %x0001  
$ status = xda__success  
$ on warning then goto error  
.  
.  
.  
$ @sub1 'p1  
.  
.  
.  
$error:  
    error handling for MAIN procedure  
  
$! Subprocedure SUB1.  
$  
$ status = xda__success  
$ on warning then goto error ! Cleanup required.  
.  
.  
$ open/read file 'p1  
$ @sub2  
.  
.  
$ goto exit
```

```

$error:
$   status = $status
$   goto exit
$
$exit:
$   set noon
$   close file
$   exit status .or. %x10000000

$!  Subprocedure SUB2.
$
$   on warning then exit $status .or. %x10000000
    .an error occurs here!
    .

```

When an error is detected in procedure SUB2, the following sequence of events occurs:

1. Its error handler exits with the error status code. The inhibit-display bit prevents DCL from displaying the status message.
2. The flow of control returns to procedure SUB1. DCL checks the status code from the invocation of SUB2, and because it is an error code, the error handler in SUB1 is invoked. It jumps to the ERROR label.
3. The error logic saves the error status code, closes the file, and exits with the error code.
4. The flow of control returns to procedure MAIN. Again DCL checks the status code and invokes the error handler in MAIN. It jumps to the ERROR label.
5. The code at the ERROR label performs the error handling for the main procedure.

The error in procedure SUB2 causes the procedure calls to be unwound, and each procedure's error handler gets a chance to run. This technique works regardless of the depth of procedure call.

8.7 **Ignoring an Error**

Sometimes it is necessary for a procedure to ignore an error caused by a subprocedure. Imagine a main procedure that allows the user to enter the name of another procedure to execute. The main procedure might want to ignore any errors in the user's subprocedure and continue execution rather than invoking its error handler. Assume that the user entered a procedure name and it was stored in the symbol XDA_PROC:

```
$ set noon
$ @'xda_proc
$ set on
.
.
```

The SET NOON command causes DCL to ignore any error status that may be returned when the subprocedure exits. Thus, the main procedure's error handler is not invoked. The SET ON command reestablishes automatic error checking.

8.8 **Obtaining the Message for a Status Code**

A VMS status code is an integer. VMS provides a service to translate a status code into message text so it can be displayed in a meaningful format. This service makes use of **message files**, which are tables that map each status code into its facility name, identification, and message text. The translation service is available to the DCL programmer through the F\$MESSAGE lexical function.

The F\$MESSAGE function requires one argument, a status code. It looks up the code in certain message files and returns a character string composed of the code's facility, severity, identification, and message text in the standard display format. If the status code cannot be located in the message files, the following string is returned:

```
%NONAME-s-NOMSG, Message number xxxxxxxx
```

In this string, the *s* represents the severity, and *xxxxxxx* represents the hexadecimal value of the status code.

VMS includes several message files in the SYS\$MESSAGE directory, but not all of them are searched by the F\$MESSAGE function. The SYSMSG.EXE file is always searched, because it contains common messages pertaining to VMS system services, the RMS record management system, and file operations performed by many utilities. No other message file is searched unless you explicitly request one with the SET MESSAGE command.

As an example, assume you are implementing a DCL application that uses the BACKUP command and needs to report problems that arise during backup operations. Many of the status codes returned by BACKUP are specific to the utility and thus are not stored in the common SYSMSG.EXE message file. Because F\$MESSAGE searches only the common message file, the following procedure will *not* work:

Table 8.3 VMS Message Files

File	Commands
CLIUTLMSG.EXE	ANALYZE/MEDIA, MAIL, PHONE, PRINT, SUBMIT, RUN, SET, SHOW, SEARCH
FILMNTMSG.EXE	ANALYZE/OBJECT, ANALYZE/IMAGE, EDIT/FDL, ANALYZE/DISK
LMF_MESSAGE.EXE	License Management Facility
PRGDEVMSG.EXE	SET COMMAND, DIFFERENCES, LIBRARIAN, LINK, MESSAGE
SYSMGTMSG.EXE	ACCOUNTING, BACKUP, INSTALL, MONITOR, AUTHORIZE, SYSMAN
SYSMMSG.EXE	Common system messages
TPUMSG.EXE	TPU

```
$ set noon
$ backup ... ! Perform the backup operation.
$ backup_status = $status
$ set on
$ if .not. backup_status
$ then
$ message = f$message(backup_status)
. perhaps write the message to a log file
.
$ endif
```

The F\$MESSAGE function is not able to return the message string for those BACKUP status codes that are specific to the BACKUP utility. If presented with a facility-specific BACKUP status, it returns the NOMSG status. In order to ensure that the function can return the message string for BACKUP statuses, you must request that VMS search an additional message file, namely the system management message file SYSMGTMSG.EXE. This is accomplished with the SET MESSAGE command:

```
$ set message sys$message:sysmgtmsg.exe
$ set noon
$ backup ... ! Perform the backup operation.
$ backup_status = $status
$ set on
$ if .not. backup_status
$ then
$ message = f$message(backup_status)
. perhaps write the message to a log file
.
$ endif
```

Now the F\$MESSAGE function will search the system management message file, followed by the common system message file. The status codes specific to BACKUP are found in the system management file, while other status codes are found in the common system file.

Unfortunately, the SET MESSAGE command only allows one additional message file to be searched at any given time. If an application gets status codes from two or more facilities whose messages are in different files, the SET MESSAGE command must be used repeatedly to establish the appropriate message file for each status as it is looked up.

Table 8.3 lists the message files you are most likely to need, along with some of the VMS commands whose messages reside in those files. A complete list is included in the *Guide to Setting Up a VMS System*.

Chapter 9



Interrupt Handling

An **interrupt** is caused by an event requiring immediate attention when it occurs. If a program is executing when an interrupt takes place, program execution must be suspended so that the event can be handled as soon as possible. Once interrupt handling is complete, program execution is resumed at the point of suspension. DCL procedures need only deal with one kind of interrupt, that which occurs when the <CTRL/y> key is pressed. The <CTRL/y> key is the means by which the VMS user signals that program execution is to be temporarily suspended or permanently canceled.

9.1 Interrupt Situations

There are two situations in which a CTRL/Y interrupt can occur during procedure execution. The first is while DCL is interpreting the command lines in the procedure. In this situation, both the <CTRL/y> and the <CTRL/c> keys generate an interrupt: the two keys are equivalent. The second situation is during the execution of a program image invoked by a command in the procedure (e.g., the COPY command invokes the COPY . EXE image). In this situation, the <CTRL/y> key generates an interrupt. Whether the <CTRL/c> key is equivalent to <CTRL/y> depends on how the program image handles <CTRL/c>. If the image explicitly requests special handling of <CTRL/c>, then that key will not generate an interrupt.

Regardless of the circumstance under which a CTRL/Y interrupt occurs, it is handled in a consistent fashion by DCL. The remainder of this chapter presents the various methods of handling interrupts.

9.2

The Default Interrupt Handler

DCL has a default interrupt handler, which it uses if you do not specify any other handler. Upon the occurrence of a CTRL/Y interrupt, DCL suspends execution of the current procedure and invokes its default interrupt handler. The action taken by the default handler depends upon the current procedure level.

When a main procedure is executing, DCL creates a **temporary command level** and issues the dollar sign (\$) prompt. At this point you can enter DCL commands to display information about the executing procedure. For example, you can use the `SHOW SYMBOL` command to display the value of symbols used by the procedure. However, care must be taken to enter only commands interpreted directly by DCL. If you should enter a command that invokes an image, the temporary command level is canceled, the DCL procedure is canceled, and the image is executed back at the DCL prompt level. This feature reduces the usefulness of temporary command levels. To resume execution of the main procedure once the temporary command level is no longer needed, enter the `CONTINUE` command. To cancel the procedure altogether, enter the `STOP` command.

▷ Ch. 20

One command that can safely be entered at the temporary command level is `SPAWN`. With the `SPAWN` command you can create a new VMS process with its own environment and DCL interpreter. In this new process you can issue any VMS command whatsoever. However, since the process has its own environment, independent from the original process, you cannot alter the state of the interrupted command procedure. (To terminate the subprocess and continue in the temporary command level, enter the `LOGOUT` command.)

Another command that can be entered at the temporary command level is `SHOW SYMBOL`. This command can be used to inspect the value of symbols created by the procedure.

When a subprocedure is executing, default interrupt handling is somewhat different. If neither the subprocedure's caller nor any of its caller's callers have established their own interrupt handlers, then the interrupt handler behaves as described above. In other words, if all procedures use the default interrupt handler, then the default interrupt handler establishes a temporary command level. However, as soon as a procedure establishes its own interrupt handler, then any

subprocedures it calls will acquire a different default handler. Rather than establishing a temporary command level, this default handler simply causes the subprocedure to exit with a status of %X1000001. The status has a success severity with the inhibit-display bit set so that no message is displayed as the subprocedure exits. Once the subprocedure exits, its caller continues normal execution, ignoring the interrupt because the status has a success severity. The caller's interrupt handler is *not* invoked. The interrupt is forgotten!

To summarize the default interrupt handler:

- The default interrupt handler for a procedure creates a temporary command level, unless
- A caller of the procedure had established its own interrupt handler, in which case the default handler causes the procedure to exit.

The default interrupt behavior is relatively complex. The value of temporary command levels is minimal, and the appearance of one in a production application would surely confuse the naive user. The default behavior is not modular, since the action taken by a handler depends upon the handlers of its calling procedures. Finally, the default interrupt handler does not always guarantee that cleanup code in procedures is executed. All in all, the default interrupt behavior is unsatisfactory; sophisticated procedures require their own interrupt handlers.

9.3

Changing Interrupt Handler Behavior

The easiest way to manage sophisticated interrupt handling for a DCL application is to integrate its interrupt handling with its error handling. The underlying premise is that a CTRL/Y interrupt is an error, a fatal error that should result in controlled termination of the procedure. (There are some situations in which an interrupt should not terminate the entire procedure. See Section 9.4.) The integration of interrupt handling with error handling requires that you establish your own interrupt handler in each procedure, which is accomplished with the ON CONTROL_Y command. The place to begin is with a main procedure. The following example takes the error-handling code from Chapter 8 and adds CTRL/Y interrupt handling:

```

$ xda__status = %x18428000
$ xda__success = xda__status + %x0001
$ xda__ctrly = xda__status + %x000C ! A CTRL/Y status.
$ status = xda__success
$ on control_y then goto control_y ! Establish the handler.
$ on warning then goto error
$
$
$ open/read file 'p1
$
$
$ goto exit
$
$control_y: ! Handle the interrupt.
$ status = xda__ctrly
$ goto exit
$
$error:
$ status = $status
$ goto exit
$
$exit:
$ set noon
$ close file
$ exit status .or. %x10000000

```

The additional commands required for interrupt handling are marked with comments. A status is created for CTRL/Y interrupts so that the procedure, when canceled because of an interrupt, can exit with a distinct status. The status has a fatal error severity and a message number of 1. The ON CONTROL_Y command establishes an interrupt action for the procedure, which jumps to the interrupt handler at the CONTROL_Y label. This interrupt handler sets the STATUS symbol to the special CTRL/Y status and then joins the procedure cleanup code at the EXIT label. Note the similarity between the interrupt handling and the error handling. Both set the STATUS symbol to reflect the reason for procedure termination, and both ensure that the cleanup code is executed.

The interrupt handler should be established with the ON command as close to the beginning of the procedure as possible. However, you must be careful not to establish the handler until all initialization required by the handler has been performed. Imagine that the ON command appeared first in the preceding example. If the user pressed <CTRL/y> after the ON command was executed but before the XDA__CTRLY symbol was established, then the interrupt handler would fail when trying to set the STATUS symbol to XDA__CTRLY.

When the interrupt handler cannot be established immediately, what happens if the user *does* press <CTRL/y> before it is established? The default interrupt handler creates a temporary command level and presents it to the user. The naive user, not realizing this, simply continues typing DCL commands. These are executed at the temporary level until one of them invokes an image. The temporary level and the procedure are then canceled and the user's environment is returned to normal. Other than a slight delay as the procedure is canceled, the user does not see any confusing behavior!

To complete the interrupt handling for a complex application, you must deal with subprocedures. If a subprocedure has cleanup code, then the interrupt handling is performed precisely as it is in the main procedure. Once the cleanup code is completed, the subprocedure exits with the XDA__CTRLY status. Because this is a fatal status, the error handler in the subprocedure's caller will be invoked, the entire procedure stack will be unwound, and the application will terminate. This illustrates the advantage of fashioning an interrupt to behave like an error: no matter how deep the procedure calls are, an interrupt causes the application to terminate.

If a subprocedure has no cleanup code, then its interrupt handler can be simplified. Instead of branching to a CONTROL_Y label, it can simply exit with the appropriate status:

```
$    on control_y then exit xda__ctrly
```

If you study the preceding main procedure example, imagining that it has no cleanup code, you will see that it simply exits with the XDA__CTRLY status after an interrupt. So the ON CONTROL_Y command can specify the interrupt handler directly when cleanup code is not needed.

One problem remains with subprocedure interrupt handling. If some initialization is required before the ON CONTROL_Y command is executed, the user has a chance to press <CTRL/y> before the interrupt handler is established. In this case DCL's default interrupt handler will cause the subprocedure to exit, because the main procedure has already established its own handler. Unfortunately, the exit status has a success severity, so the subprocedure's caller will blithely continue to execute as if the subprocedure had completed successfully. There are two choices in this situation. The first is to ignore the problem. After all, the probability of its occurrence is low. The second is to use the SET NOCONTROL command to block CTRL/Y interrupts during initialization:

```

$   set nocontrol=y
    .
    . initialization
    .
$   set control=y
$   on control_y then ...

```

The SET NOCONTROL command disables CTRL/Y interrupts. If the user presses <CTRL/y> during the initialization sequence, the interrupt is ignored. The SET CONTROL command reenables interrupts. This may appear to do the trick, but the problem is still not solved and a new problem is introduced:

- A CTRL/Y interrupt just before the SET NOCONTROL command (i.e., as the subprocedure is being invoked but before the first line is executed) still causes the subprocedure to exit with a success status.
- If the initialization code goes awry and begins to loop or takes a bad branch, CTRL/Y interrupts will remain disabled. The only way to cancel the procedure may be to stop the VMS process from another terminal.

The author recommends ignoring the interrupt “window” at the beginning of a subprocedure. There is no way to completely eliminate the window.

9.4 **Using Interrupts for Other Purposes**

There may be times when you want to use <CTRL/y> for something other than canceling the entire application. For example, a procedure may display a long list of items for the user to peruse. If the user is not required to inspect the entire list, <CTRL/y> can be used to cancel the list and go on to the next step of the procedure. The procedure must temporarily override normal interrupt handling while the list is being displayed. The following example illustrates a subprocedure in which a list is displayed:

```

$   on control_y then goto control_y
$   on warning then goto error

$   on control_y then goto cancel_display
    .
    . display the list
    .
$cancel_display:
$   on control_y then goto control_y

```

The subprocedure establishes the standard interrupt handler when it starts up. At the point where the list is to be displayed, it establishes a new interrupt handler, which jumps to the `CANCEL_DISPLAY` label, thereby canceling the display. This interrupt handler is in effect during the entire process of displaying the list. After the display completes or is canceled, the standard interrupt handler is reestablished for the remainder of the procedure. The standard handler must be explicitly reestablished, because it was superseded by the special handler for the list display. Special interrupt handlers can be established whenever needed, but do not forget to reestablish the standard one.

9.5 ***Tying It All Together***

Chapters 7, 8, and 9 have presented much complex material. The DCL environment and error and interrupt handling not only are deeply intertwined but are also the most frequently neglected aspects of DCL programming. The next chapter ties all this material together by discussing the overall structure of DCL applications.

Chapter 10



Application Structure

The previous chapters have addressed the DCL programming environment, error handling, and CTRL/Y interrupt handling. These three aspects of DCL application design are by far the most complicated. You have gotten through the most difficult part of this book. With the knowledge gained so far, you can structure an entire DCL application, large or small, and feel confident that the application will behave in a consistent and reliable fashion.

This chapter brings together all the information presented so far in order to demonstrate one method of structuring an entire application. It also describes some conventions that should be followed when developing an application so that the application will be compatible with other VMS applications and can coexist with them.

10.1 Naming Conventions

A wide variety of software products can be simultaneously installed and running on a VMS system. Such products may be provided by DIGITAL, by third-party software vendors, or by your own development staff. In order to prevent these products from interfering with one another, a series of guidelines must be followed by each and every product. These guidelines are presented in the VMS document entitled *Guide to Creating VMS Modular Procedures*. The majority of guidelines do not pertain to DCL, but a few important ones do. These are the naming conventions for various system objects, such as logical names and

files. If you adhere to the naming conventions carefully, your DCL application will not interfere with other applications written in DCL or any other language. The following paragraphs describe the naming conventions relevant to DCL applications:

Directories. The name of the top-level directory containing the application files must be the same as the facility code or begin with the facility code and an underscore (e.g., XDA or XDA_ROOT).

Files. The names of all the component files of the application must consist solely of the facility code or begin with the facility code and an underscore (e.g., XDA.COM or XDA_CONFIG.DAT).

Global symbols. The names of all global symbols created by the application must begin with the facility code and an underscore (e.g., XDA_MODE). This is particularly important because DCL applications usually leave global symbols around after they exit. These global symbols remain in existence until the process terminates, so they must not conflict with global symbols created by other applications.

Logical names. All logical names defined by the application must begin with the facility code and an underscore (e.g., XDA_SYSTEM).

Status codes. The status code symbols created by the application should begin with the facility code and two underscores (e.g., XDA__SUCCESS). Because status symbols are not global, this convention is only a suggestion.

The examples in this book follow these naming conventions.

10.2 *Invoking the Application*

A complex DCL application can consist of many procedures and data files. When the application is installed on a VMS system, the files must be placed in appropriate directories, so that they can be accessed when the application is run. The simplest directory organization involves creating a single directory to contain all of the application's files. A system logical name is defined to refer to the directory, so that users can locate the application files.

The following discussion assumes that the example application, XDA, has been installed in a single directory. The logical name XDA_SYSTEM is defined to refer to that directory. To run the application, a user must invoke its main procedure, which establishes the application environment and directs its execution. The main procedure for the sample application is named XDA.COM. The following personal

command is established in each user's login procedure or in the system login procedure:

```
$ xda == "@xda_system:xda"
```

Once the command XDA is established, a user invokes the application in a fashion similar to other VMS applications:

```
$ xda
```

10.3 **The Main Procedure**

Every DCL application has a **main procedure**, the procedure that executes first when the application is run. The main procedure is responsible for establishing the overall environment in which the application will execute. A small application may consist only of the main procedure, but most applications will include other procedures invoked by the main procedure, which execute in its established environment.

The techniques used to establish the application environment have been introduced in preceding chapters. These techniques are brought together in the following main procedure, which is the main procedure of the hypothetical application, XDA:

```
$! The main procedure for the XDA application.
$
$! 1. Block the use of dangerous "external" personal commands.
$
$ backup = "backup"
$ copy = "copy"
$ :
$ .
$
$! 2. Save the current message and privilege settings.
$
$ saved_message = f$environment("MESSAGE")
$ saved_privs = f$setprv("all")
$
$! 3. Define some simple status codes for the application.
$
$ xda__status = %x10428000
$ xda__success = xda__status + %x0001
$ xda__ctrly = xda__status + %x000C
$
```

```

$! 4. Establish the interrupt and error handlers. Make sure the
$! summary status indicates success if we exit normally.
$
$   status = xda__success
$   on control_y then goto control_y
$   on warning then goto error
$
$! 5. Establish the message and privilege environment.
$
$   set message/facility/severity/identifier/text
$   junk = f$setprv(saved_privs+"PRMMBX,SYSPRV")
$
$! 6. Set up some useful symbols for the application.
$
$   ask      = "read sys$command /prompt="
$   define   = "define/nolog"
$   libcall  = "@xda_system:subroutine-library"
$   false    = 0
$   display  = "write sys$output"
$   true     = 1
$   undefine = "deassign"
$
$
$   .
$   . begin the application
$   .
$   goto exit
$
$! The interrupt handler sets the summary status to indicate
$! that <CTRL/y> was pressed and then joins the cleanup code.
$
$control_y:
$   status = xda__ctrly
$   goto exit
$
$! The error handler sets the summary status to the "offending"
$! status and also joins the cleanup code.
$
$error:
$   status = $status
$   goto exit
$

```

```

$!   The exit routine first performs cleanup.  Then it restores the
$!   original DCL environment.  Finally it exits with the summary
$!   status.
$
$exit:
$   set noon
$   .
$   .cleanup code
$
$   set message 'saved_message
$   junk = f$setprv(saved_privs)
$   exit status .or. %x10000000

```

The application environment is initialized by performing the following steps in order (numbers refer to comments in the code):

1. Personal commands are defined for “dangerous” VMS commands, so that any personal commands defined outside the procedure are effectively hidden and will not interfere with the application.
2. The current settings of DCL environment items, such as the message format and privileges, are saved.
3. Basic status values needed by the error and interrupt handlers are defined.
4. The error and interrupt handlers are established. Up to this point, errors and interrupts will cancel the procedure harmlessly. From this point on, they will cause the appropriate handler to be invoked.
5. The settings of DCL environment items are altered as required by the application.
6. Various symbolic constants, personal commands, and other useful symbols are defined. These can be used by the rest of the application.

Once these steps are completed, the actual application can commence. The remainder of the main procedure, along with all subprocedures of the application, run in the environment established by the initialization code.

The interrupt and error handlers are identical to those illustrated in previous chapters. They set the STATUS symbol to the appropriate status and join the cleanup code at the EXIT label. The cleanup code performs application-specific cleanup operations, such as closing open files or deleting temporary ones. It then reestablishes the DCL environment as it was prior to the application and exits with the summary status. Another procedure that invokes the XDA application can use this status to determine the success or failure of its request.

A Subprocedure

When a DCL application is complex, it is difficult to organize all the necessary code in one procedure. Just as large programs written in conventional languages are broken down into modules and subroutines, so are large DCL programs. One of the application procedures is the main procedure and the rest are **subprocedures**. The subprocedures are designed to run in the environment established by the main procedure. All the procedures must cooperate to allow error and interrupt handling to proceed smoothly.

The application procedures invoke one another using the logical name referring to the directory containing the procedures. For example, the main procedure in our application invokes subprocedure XDA_SEARCH as follows:

```
$ @xda_system:xda_search parameter...
```

The skeletal structure of a subprocedure depends upon whether it requires any cleanup code. Cleanup code consists of commands that *must* be executed regardless of the manner in which the subprocedure exits. It might close open files, delete temporary files, or deassign logical names. A subprocedure without cleanup code is organized as follows:

```
$! A subprocedure for the XDA application.
$
$ on control_y then exit xda__ctrly
$ on warning then exit $status .or. %x10000000
$
$ .:initialization
$ .
$
$ .:body of subprocedure
$ .
$ exit xda__success
```

The subprocedure establishes a CTRL/Y interrupt handler, which simply exits with the XDA__CTRLY status. This status has a fatal error severity, so the caller's error handler is invoked and the procedure stack is unwound. The subprocedure also establishes an error handler, which exits with the offending status (the inhibit-display bit is set to prevent display of the corresponding message). The error handler traps warnings, errors, and fatal errors, returning them to the caller, so that again its error handler is invoked. Both interrupts and errors cancel the

subprocedure and allow its caller to determine subsequent actions. The initialization section and subprocedure body perform the actions attributed to the subprocedure. Once complete, the subprocedure exits with a success status and its caller continues execution.

The organization of a subprocedure with cleanup code is a bit more elaborate. The interrupt and error handlers must arrange to execute the cleanup code before the procedure exits. The normal paths through the subprocedure must do the same:

```
$!   A subprocedure for the XDA application.
$
$   status = xda__success
$   on control_y then goto control_y
$   on warning then goto error
$
$   .
$   : initialization
$   .
$
$   .
$   : body of subprocedure
$   .
$
$   goto exit
$
$control_y:
$   status = xda__ctrlly
$   goto exit
$
$error:
$   status = $status
$   goto exit
$
$exit:
$   set noon
$   .
$   : cleanup code
$   .
$   exit status .or. %x10000000
```

A subprocedure with cleanup code is organized almost identically to the main procedure. The symbol `STATUS` must be set to the desired exit status before joining the cleanup code at the `EXIT` label. In the normal case, `STATUS` is set to the success status, the subprocedure executes, and eventually it branches to the `EXIT` label. If an interrupt or error occurs, the handler resets the `STATUS` symbol to the appropriate status and then branches to the `EXIT` label. In all

cases, the cleanup code is executed and the status is returned to the caller with the inhibit-display bit set.

Variations on the preceding themes are certainly possible. In Chapter 9, for example, interrupt handling was modified to allow a display to be canceled in the middle without terminating the application. You may find that other variations are required by your application; do not hesitate to devise new procedure organizations to accept these variations. Just keep in mind the following requirements:

- Every procedure must exit with a success or error status that accurately reflects what happened in the procedure.
- Unintentional errors and interrupts must be detected and cause the procedure to exit with the appropriate error status.
- Cleanup code must be executed regardless of the reason for procedure exit.

If you are diligent in the application of these rules, your application will behave in a predictable manner through all execution paths and regardless of any errors or interrupts that might occur.

10.5 *The Subroutine Library*

A large DCL application often includes a handful of subprocedures that perform general operations needed by all parts of the application. A good example is the `SIGNAL` subroutine, which is called to display an informational or error message in the standard format used by all VMS software. This operation is performed in quite a few places in the application, and many lines of code are necessary to implement it, so it is a perfect candidate for a subroutine. Section 10.6 describes the `SIGNAL` subroutine in detail. Rather than creating a separate procedure file for each of these general subroutines, it is easier to group them together into a single large procedure, which serves as the subroutine “library” for the application. The subroutine library is not a VMS library in the technical sense, but it serves a similar purpose: to collect a set of useful subroutines in a single place for easy access and maintenance.

The library resides in a single procedure file named `SUBROUTINE-LIBRARY`. A subroutine is called by invoking the procedure with certain parameters. The first parameter is always the name of the desired subroutine. The remaining parameters act as “arguments” to the subroutine; they specify information the subroutine needs to perform its assigned function. Here is a skeletal procedure library that might be included in the XDA sample application:

```

$   on control_y then exit xda_ctrly
$   on warning then exit $status .or. %x1000000
$
$   goto 'p1
$
$ASK: ! P1:ASK P2:symbol P3:prompt P4:data-type P5:default
$     !           P6:options
      .perform ask function
$
$SIGNAL: ! P1:SIGNAL P2:severity P3:identifier P4:text ...
      .perform signal function
$
$UNIQUE_NAME: ! P1:UNIQUE_NAME P2:symbol P3:pattern
      .perform unique name function

```

After establishing the usual interrupt and error handlers, the procedure performs a GOTO to the label specified by P1, the first parameter. The apostrophe substitution causes the value of P1, the subroutine name, to be substituted in the GOTO command. The target of the branch thus becomes the name of the subroutine.

Each subroutine begins with the label that names the subroutine (e.g., SIGNAL) and acts as the target of the initial GOTO. A comment is provided to describe the subroutine's parameters. The subroutine is responsible for performing the requested function and finally exiting with a summary status: a success status if the operation succeeded or some error status if it failed. The success status will cause the calling procedure to continue executing, but an error status will trigger its error handler. (The library illustrated here includes subroutines named ASK, SIGNAL, and UNIQUE_NAME.)

The easiest way to call subroutines in the library is to set up a personal command. Assuming that the logical name XDA_SYSTEM has been established prior to invoking the application, the following personal command can be established in the main procedure's initialization code:

```

$   libcall = "@xda_system:subroutine-library"

```

Once this personal command is established, individual subroutines can be called as follows:

```
$ libcall ask xda_ans s "What is your name:"
```

—or—

```
$ libcall signal xda e baddev "Invalid device specified."
```

—or—

```
$ libcall unique_name xda_name xda_?.tmp
```

A general-purpose DCL subroutine library is presented in Appendix C. This subroutine library is not specific to any one application, but contains subroutines useful to all applications. The library procedure is self-initializing and thus does not rely on any symbols or logical names defined by the application. The author hopes that you will find the subroutine library useful for your own DCL programming.

10.6 *The SIGNAL Subroutine*

▷ Ch. 8

The SIGNAL subroutine is the first subroutine introduced in this book, and it is perhaps the most important one. SIGNAL provides the means for a procedure to display a success or error message and then invoke its error handler if appropriate. The term **signal** is borrowed from the VMS run-time library routine LIB\$SIGNAL, which is the routine called by programs written in conventional languages to perform a similar function. The SIGNAL subroutine performs two major steps:

1. It displays one or more messages based upon its parameters. These messages are formatted according to the VMS convention for informational and error messages.
2. It exits with a status whose severity depends upon one of its parameters. If the severity is an error or fatal error, the caller's error handler will be invoked.

The content of the messages and the severity of the exit status are determined by the subroutine's parameters. At least five parameters must be specified, more are allowed:

- P1. As with all subroutines in the library, the first parameter is the name of the subroutine: SIGNAL.
- P2. The second parameter is the facility code of the facility signaling the error.

- P3. The third parameter is the severity of the signaled message. A single letter is used: S for success, I for informational, W for warning, E for error, F for fatal error.
- P4. The fourth parameter is the identification code for the message. It is composed of letters, digits, and underscores (_).
- P5. The fifth parameter is the text of the message. It must be enclosed in quotation marks.
- P6–P8. If multiple messages are desired, additional text parameters can be specified. Each message is displayed on a separate line.

The following examples illustrate calls to the SIGNAL subroutine and the resulting messages.

```
$ libcall signal xda i copydone "All files have been copied."
%XDA-I-COPYDONE, All files have been copied.
```

–or–

```
$ libcall signal xda i copydone "Copying is complete." -
    "'file_count files were processed.'"
%XDA-I-COPYDONE, Copying is complete.
-XDA-I-COPYDONE, 37 files were processed.
```

–or–

```
$ libcall signal xda e nodisk -
    "The specified disk does not exist."
%XDA-E-NODISK, The specified disk does not exist.
```

–or–

```
$ libcall signal xda f nosuchfil -
    "The file "'file_spec does not exist.'"
%XDA-F-NOSUCHFIL, The file SYS$MANAGER:SPECIAL.DAT does not exist.
```

The SIGNAL subroutine treats warning messages in a special fashion. Instead of returning a warning status to correspond to the warning message, it returns an informational status. This prevents the caller's error handler from being invoked and allows warning messages to be issued as a matter of course.

The SIGNAL subroutine is fully documented in Appendix C, Subroutine Library.

Captive Accounts

VMS supports a special type of user account called a **captive account**. The idea behind a captive account is that a user logs in to the account and is then held “captive” by a login procedure that begins execution immediately after the user name and password are verified. This procedure completely controls which applications the user can run. In order to maintain this tight control, the procedure must not allow the user to escape to DCL and enter arbitrary commands.

DCL provides the features necessary to write a captive procedure from which the user cannot escape. Here is a skeleton captive procedure:

```
$  on control_y then continue
$  on warning then logout/brief
    .procedure actions
    .
$  logout
```

This procedure does not establish an interrupt handler as described in the previous sections. Instead it specifies the CONTINUE command as the handler. The CONTINUE command effectively ignores interrupts and continues execution of the procedure and any image that may be running at the time. Thus CTRL/Y interrupts are disabled.

The procedure also does not handle errors in the normal fashion. Instead, the error handler executes a LOGOUT command, which immediately logs the user out of the system. More sophisticated error handling is certainly possible in a captive procedure, but you run the risk of opening a hole through which the user can escape to DCL.

Any application programs invoked from a captive procedure must also take care to hold the user captive. An application that allows the user to execute arbitrary DCL commands, as some editors do, is certainly not safe for captive accounts.

Chapter 11



Obtaining User Input

VMS is an interactive operating system, so it is not surprising that many of the programs written for it are also interactive. The dialog between a computer program and a user flows in two directions, from the program to the user, and from the user to the program. Chapter 6 explained how to format and display output for the benefit of the user. This chapter discusses how to obtain input from the user.

▷ Ch. 20

A video terminal consists of an output device, the screen, and an input device, the keyboard. In addition to the process-permanent files `SYS$OUTPUT` and `SYS$ERROR`, which are used to write information to the screen, VMS maintains two process-permanent files from which input can be obtained. These files are accessed via the logical names `SYS$COMMAND` and `SYS$INPUT`. Exactly which files are accessed depends upon the current environment, as follows:

At the DCL prompt. Both `SYS$COMMAND` and `SYS$INPUT` refer to the terminal keyboard.

During execution of an interactive procedure. `SYS$COMMAND` refers to the terminal keyboard while `SYS$INPUT` refers to the procedure file.

During execution of a batch procedure. Both `SYS$COMMAND` and `SYS$INPUT` refer to the procedure file.

At the DCL prompt or during interactive procedure execution, keyboard input can be obtained by reading from SYS\$COMMAND. During batch procedure execution, there is no keyboard from which to obtain input.

11.1 **The INQUIRE Command**

The INQUIRE command is the simplest method of obtaining input from the user. Unfortunately, the command has many quirks, which diminish its usefulness. The command requires two parameters: a symbol and a prompt string. The prompt string cannot be an expression but must be a simple literal string enclosed in quotation marks. The prompt string is displayed, a line is read from SYS\$COMMAND, and the symbol is set to a character string containing the text of the line. The character string does not include the carriage return at the end of the line. The following behavior should be noted:

- The prompt string is actually optional. The INQUIRE command uses the symbol name as the prompt if the string is omitted.
- A colon and a space are appended to the prompt string in an effort to make it pretty (this can be suppressed with the /NOPUNCTUATION qualifier, as described below).
- An apostrophe in the input causes substitution to take place before the symbol is set to the input line. This is potentially dangerous, because the user can “sneak in” the value of any symbol whose name is known.
- Leading and trailing whitespace is removed from the input, and each occurrence of embedded whitespace is collapsed to a single space. The term **whitespace** refers to a sequence of one or more spaces or tabs.
- The input is converted to uppercase letters.
- The previous two actions do not take place for portions of the input enclosed in quotation marks, but the quotation marks themselves are removed.

This complex behavior makes the content of the input difficult to predict and control.

The format of the prompt can be controlled using the /NOPUNCTUATION qualifier. When this qualifier is present, the colon and space are not appended to the prompt; the prompt will be displayed exactly as it appears in the command. There are other qualifiers, which are described in the *VMS DCL Dictionary*.

Because of the idiosyncratic nature of the INQUIRE command, it should only be used in personal command procedures that perform simple input operations.

The READ Command

The READ command is the preferred means of obtaining input from the user. The READ command can do everything that the INQUIRE command can, yet it has none of the latter's idiosyncracies. The READ command requires two parameters: a logical name identifying the file from which input is to be read, and a symbol that is set to the input. A line is read from the file and the symbol is set to a character string containing the text of the line. Terminal input is obtained by reading from the file SYS\$COMMAND. In addition, the /PROMPT qualifier is necessary in order to specify a prompt string. Here are a few examples of the READ command, along with the resulting prompts:

```
$ read sys$command/prompt="Name: " line
Name:
```

- or -

```
$ read sys$command/prompt="Do you want to delete the file? " -
  answer
Do you want to delete the file?
```

- or -

```
$ prompt = f$fa0("(!5%T) Command: ", 0)
$ read sys$command/prompt="''prompt'" command
( 9:28) Command:
```

The third example demonstrates how the F\$FAO lexical function can be used to create a prompt string containing the current time, which is then substituted in the READ command line. Unlike the INQUIRE command, the READ command does not alter the prompt string or input text in any way. Unfortunately, there is no way to use the READ command to obtain single keystrokes, cursor control keys, or editing keys like those on the application keypad.

A personal command can be used to make READ commands more concise:

```
$ ask = "read sys$command/prompt="
  :
  :
$ ask "Name: " line
$ ask "Are you sure you want to delete the file? " answer
$ ask "''f$fa0("(!5%T) Command: ", 0)'" command
```

It is a VMS convention to use the CTRL/Z character to signify **end-of-file** when reading from a terminal. Of course, there is no physical end-of-file on a terminal; the user can always type more characters. However, it makes sense to rely on a logical end-of-file to terminate a series of related input lines. For example, most interactive VMS programs will exit when the user enters <CTRL/z>, because the <CTRL/z> marks the end of the input related to the program. If you are creating a procedure to look like any other VMS program, this behavior should be emulated.

The READ command accepts an /END_OF_FILE qualifier that specifies a label. If (and only if) <CTRL/z> is pressed while the read is in progress, DCL transfers control to the command containing the label. This is an “arbitrary goto” and should be used with care so as not to adversely affect the readability of the program.

```
$ ask "Command: " command /end_of_file=15
      .process command
$ goto 19
$15:
      .process end-of-file
$19:
```

If some input is typed on the line before <CTRL/z> is pressed, the symbol is set to the input typed so far, and then control is transferred to the end-of-file label. The code at that label may need to process the characters in the symbol before performing end-of-file cleanup.

11.3 **Default Answers**

It is common practice to associate a **default answer** with a question. The default answer is the answer the program will assume if the user presses <RETURN> without entering any information. The default answer should be the most likely answer to the question, unless that answer is potentially dangerous. For example, the default answer to the question “Do you want to delete all your files?” should be no. By convention, the prompt string for a question contains the default answer in square brackets at the end of the question, before the trailing punctuation character.

It is quite simple to implement default answers in DCL. If the user presses <RETURN> without entering any other characters, the symbol specified in the

READ command will be set to the null string. This condition can be determined and the symbol set to the default answer.

```
$ ask "Are you sure you want to delete the file [NO]? " delete
$ if delete .eqs. "" then delete = false
```

-or-

```
$ ask "Which directory is to be listed [working]: " dir
$ if dir .eqs. "" then dir = f$environment("DEFAULT")
```

-or-

```
$ ask "Number of copies to print [1]: " copies
$ if copies .eqs. "" then copies = 1
```

In each case, a check is made to see if the input symbol is set to the null string. If so, it is reset to the default answer.

11.4 ***Editing Input***

In some circumstances, it is necessary to edit the user's input before processing it. For example, if the input is a product name that is to be compared to a set of valid names, it is necessary to convert the name to uppercase letters so that it will compare correctly with the valid choices. The F\$EDIT lexical function provides a set of editing capabilities that make it quite easy to perform most common input editing. The F\$EDIT function takes two arguments: the string to be edited and a string containing a keyword or list of keywords, which determine the exact editing performed. The function returns the edited string.

Assume that some input has been read into the symbol LINE. The following are examples of some of the editing you can perform:

```
$! Convert the line to uppercase.
$ line = f$edit(line,"UPCASE")
```

-or-

```
$! Remove leading and trailing whitespace from the line.
$ line = f$edit(line,"TRIM")
```

-or-

```
$! Remove leading and trailing whitespace, and compress
$! occurrences of whitespace to a single space.
$ line = f$edit(line,"TRIM,COMPRESS")
```

—or—

```
$! Remove leading and trailing whitespace, compress
$! whitespace, and convert to uppercase.
$ line = f$edit(line,"TRIM,COMPRESS,UPCASE")
```

As the examples illustrate, the various editing operations can be performed singly or in any combination.

11.5 Checking the Validity of Input

Once a procedure accepts input from the user, it is often necessary to check the input for errors. For example, if the input is supposed to be an integer, the procedure must ensure that the characters entered form a valid integer and that the integer is in the desired range. It is a mark of a well-written procedure that all user input is checked as carefully as possible, as soon as possible. The longer a procedure waits to check its input, the more out-of-context the resulting error messages will be. The following sections discuss input checking for each type of data.

11.5.1 Integers

Whenever a procedure expects an integer value in response to a query, it must check the user's input to ensure that it represents a valid integer. Because the READ command always reads its input as a character string, the user's input will be represented as a string even though it is intended to be an integer. The F\$TYPE lexical function is used to check the type of the input: if the input string contains the valid external representation of an integer, the function will return "INTEGER", otherwise it will return "STRING".

After it is determined that the input is a valid integer, the input is converted to an integer data type. Once this is done, any additional validation such as range checking, can be performed. If any of the tests fail, an appropriate message is issued and the question is asked again. The following loop illustrates a method for checking an integer that is intended to be a year number:

```

$   input_ok = false
$10:  ask "Which year do you want to report on: " year_string
$     if f$type(year_string) .eqs. "INTEGER"
$     then
$       year = f$integer(year_string) ! Convert year to an integer.
$       input_ok = year .ge. 1900 .and. year .le. 2100
$       if .not. input_ok then libcall signal xda w badyear -
$         "The year must be in the range 1900--2100."
$     else
$       libcall signal xda w badint "Please enter an integer."
$     endif
$     if .not. input_ok then goto 10

```

The loop is controlled by the boolean variable `INPUT_OK`, which is initialized to `FALSE` before the loop is entered and tested at the bottom of the loop to determine whether another iteration is necessary. On each iteration the user is prompted for a year. If the input is a valid integer *and* the integer is in the required range, then `INPUT_OK` is set true. Otherwise an appropriate warning message is displayed and `INPUT_OK` remains false. The loop is continuously executed until valid input is received.

After the loop terminates, the symbol `YEAR` will be set to the desired year, stored as an integer. Note the use of the `F$INTEGER` lexical function to convert the year from a string to an integer.

11.5.2 Character Strings

Strings are generally the easiest kind of input to check. Sometimes no checking is required at all, because any string is valid. Often the length of the string must be checked to ensure that it is within a required range. At other times, the content of the string must be inspected because only certain characters are allowed. The following loop asks the user to enter a comment for a history file:

```

$10:  ask "History comment: " comment
$     input_ok = f$length(comment) .ge. 1 .and. -
$           f$length(comment) .le. 80
$     if .not. input_ok then libcall signal xda w commlen -
$       "The comment must contain 1--80 characters."
$     if .not. input_ok then goto 10

```

This loop uses the `INPUT_OK` variable in the same fashion as the previous example. No data type checking is required, because the input is guaranteed to be a string. The length of the input is checked to ensure that it is in the required

range 1–80. The COMMENT symbol contains the comment string after the loop terminates.

The following example illustrates the situation where the input must be chosen from among a fixed set of valid strings:

```
$10:  ask "Print DETAIL or SUMMARY report [SUMMARY]: " report_type
$     if report_type .eqs. "" then report_type = "SUMMARY"
$     report_type = f$edit(report_type,"UPCASE")
$     input_ok = report_type .eqs. "DETAIL" .or. -
$               report_type .eqs. "SUMMARY"
$     if .not. input_ok then libcall signal xda w badtype -
$               "Please enter DETAIL or SUMMARY."
$     if .not. input_ok then goto 10
```

The checking methodology is similar to the one in the previous example, except that the input must be one of two particular strings. The input is converted to uppercase letters before it is checked against the two strings; this allows the user to enter input in uppercase or lowercase. The query also provides a default answer ("SUMMARY").

11.5.3

Booleans

▷ Tab. 3.1

Strictly speaking, validation of a boolean response is unnecessary because any string is a valid boolean value. In spite of this, you should do the user a favor and accept only the words YES or NO and their abbreviations. Otherwise the user can accidentally enter TO when NO was meant, and DCL will interpret this as a true value because it begins with T.

The following loop asks a yes/no question and validates the answer:

```
$10:  ask "Do you want to clear the directory [NO]? " clear_dir
$     if clear_dir .eqs. "" then clear_dir = "NO"
$     clear_dir = f$edit(clear_dir,"UPCASE")
$     input_ok = f$locate(clear_dir,"YES") .eq. 0 .or. -
$               f$locate(clear_dir,"NO") .eq. 0
$     if .not. input_ok then libcall signal xda w badyesno -
$               "Please enter YES or NO."
$     if .not. input_ok then goto 10
```

A default answer of NO is provided, and the input is converted to uppercase letters for comparison purposes. The F\$LOCATE function is used to ensure that the input is YES, NO, or an abbreviation thereof. To test that the input is an abbreviation of YES, the input must match the string "YES" beginning at position zero. The only

inputs that pass the test are "Y", "YE", and "YES". A similar test is used for NO, allowing only the inputs "N" and "NO". All other inputs are rejected.

11.6 **Main Procedure Arguments**

A procedure often requires input values before it can determine the exact operations to perform. These input values must be obtained before the procedure can begin its intended task. For example, a procedure that copies a file from the local network node to another node requires two input values: the file spec and the remote node name. One method of passing these values to the procedure is via procedure parameters. Unfortunately, this requires that the user remember both the order and the syntax of the parameters. Another method of obtaining the values is to prompt the user and read the values from the keyboard at the beginning of the procedure. This solves the first problem but fails to work when the procedure is submitted to batch because there is no keyboard. A combination of both techniques offers the maximum flexibility.

Assume that the procedure expects two parameters: the file to be copied and the remote node name. The following code will check for the parameters, and if either one is missing, prompt for it and read it from the keyboard:

```
$! This procedure accepts two parameters, as follows:
$! P1: the file to be copied
$! P2: the remote node to receive the copy
.
.
$ if p1 .eqs. "" then ask "File: " p1
$ if p2 .eqs. "" then ask "Remote node: " p2
```

For each parameter in turn, the procedure determines whether the parameter was specified when the procedure was invoked by testing it for the null string. If the parameter is missing, an ASK command is used to prompt and obtain a value for it (remember that the personal command ASK is short for READ SYS\$COMMAND /PROMPT=). The user can pass a required value as a parameter *or* choose to be questioned for the value. When submitting the procedure to batch, the values must be supplied as parameters. If they are not, the ASK command fails and the procedure terminates with an error.

A Sophisticated Input Subroutine

As demonstrated in the previous sections, the operation of asking a question is both a common and a complex one. For this reason, it is useful to embody the techniques for asking questions in a subroutine that can be invoked whenever a question needs to be asked. This subroutine is included in the subroutine library listed in Appendix C, where it is fully documented. Only some of its features are described here.

The subroutine is called ASK and requires at least three parameters: a global symbol that will be set to the answer, a data type specifier, and a prompt string. There are optional parameters, the first of which is the default answer (if omitted, there is no default and the user must input a value). The procedure performs the following steps, shown here in simplified form:

1. Display the prompt, including the default answer if specified.
2. Read the answer from SYS\$COMMAND.
3. If <RETURN> was pressed without entering any data:
 - Go back to step 1 if no default was specified.
 - Assume the default answer if one was specified.
4. Check the answer to ensure that it conforms to the data type specifier. If not, display a message and go back to step 1.
5. Set the global symbol to the answer.
6. Exit.

Once the subroutine exits, the answer is available in the global symbol provided as the first parameter.

The data type specifier is a single letter that restricts the type of data the user can enter: I for integer, S for string, B for boolean (yes/no).

Here are a few examples:

```
$! Ask for the user name of the person to whom the mail
$! is sent. The answer must be a string.
$
$ libcall ask xda_user s "User to receive mail:"
```

–or–

```

$! Ask for the number of copies to be printed. The answer
$! must be an integer, the default being 1.
$
$ libcall ask xda_copies i "Number of copies to print:" 1

```

–or–

```

$! Determine if the user really wants to delete the file.
$! The answer must be a boolean, the default being NO.
$
$ libcall ask xda_delete b -
      "Do you really want to delete the file?" no

```

–or–

```

$! Wait for the user to press RETURN.
$
$ libcall ask xda_ s "Press RETURN when ready:" continue

```

▷ Ch. 10

Notice how the global symbols used in the ASK calls conform to the VMS convention for naming global symbols. When the user’s answer is just discarded, as in the last example, the “throw-away” symbol XDA_ is specified.

11.8 **Redirecting Program Input**

An interactive VMS program normally obtains its input from SYS\$INPUT. This works fine when the program is run from the DCL prompt but has an unexpected consequence when the program is run from a procedure. Because SYS\$INPUT is directed to the procedure file while a procedure is executing, the program attempts to read its input from the procedure file. This is why the procedure examples in this chapter read from SYS\$COMMAND: it is directed to the terminal keyboard during procedure execution. There is a way to force a program image to read from SYS\$COMMAND.

Before tackling that problem, however, it is interesting to point out some uses for the standard behavior. If a procedure runs an interactive program but wants to provide “canned” answers to all the questions asked by the program, these answers can be placed directly in the procedure file. For example, assume you want to extract message number 1 from the mail folder named BOILERPLATES and place it in a temporary file:

```

$ mail
read boilerplates 1
extract/noheader sys$scratch:xda_boilerplate.tmp
exit
$

```

:process the boilerplate file

Because SYS\$INPUT is directed to the procedure file, the MAIL utility will read its commands from the procedure. The three commands (READ, EXTRACT, EXIT) accomplish the task of extracting the message into a temporary file, just as if an interactive user had typed them in. The EXIT command is not strictly necessary, because the first line containing a dollar sign signals end-of-file to the program. Program input data included directly in a procedure is called **image data**. The technique of placing input in the procedure works only when the input is entirely static. Apostrophe substitution is not performed on the input lines, so they cannot be customized the way normal DCL commands can be.

When an interactive program needs to read its input from the terminal, it must be forced to read from SYS\$COMMAND. This is accomplished with the DEFINE command. Suppose, for example, you want to run the PHONE utility as part of a procedure and have it read from the terminal:

```

$ define/user_mode sys$input sys$command
$ phone

```

The DEFINE command redefines the SYS\$INPUT logical name so that it refers to the SYS\$COMMAND logical, which in turn is directed at the terminal keyboard. The /USER_MODE qualifier makes this redefinition temporary, lasting only for the next image execution. Therefore the PHONE utility will obtain its input from the terminal, even though it is invoked by a procedure.

▷ Ch. 14

11.9 **Displaying Large Amounts of Text**

During the execution of a command procedure, SYS\$INPUT refers to the procedure file. If a procedure displays large amounts of text whose content does not vary from one time to the next, it is easier to use a single TYPE command than multiple DISPLAY commands. In the following example, a lengthy help message is displayed:

```
$ type sys$input:
```

This is a lengthy help message. It doesn't change from one use to the next, so symbol values aren't used and substitution is unnecessary. Because `SYS$INPUT` refers to the procedure file, the `TYPE` command displays lines directly from the file. The command will display lines until it encounters the next procedure command (i.e., a line beginning with a dollar sign).

We can even have blank lines for paragraph separation. This is much easier than a series of `DISPLAY` commands with strings in double quotes.

```
$
```

```
$ next command
```

Chapter 12



Debugging

The task of program debugging is an integral part of developing DCL applications, just as it is with every programming project. Unfortunately, the comprehensive collection of VMS debugging facilities is of no help to the DCL programmer. There is no symbolic debugger, no backtrace upon program failure, and no program analysis tools. You are forced to use “old-fashioned” debugging techniques, such as printing the values of variables at critical points in the program.

DCL does have one advantage over most other programming languages. Because it is strictly interpreted, the edit/compile/link/test cycle is simplified to an edit/test cycle. Making simple modifications to a procedure so it will display debugging information is not particularly painful, because you do not have to wait for a compile and link. The following sections describe the various techniques for debugging DCL procedures.

12.1 Verification

Procedure verification is the most powerful debugging tool available to the DCL programmer. There are two types of verification, each of which can be independently enabled or disabled:

Procedure verification. When this is enabled, DCL displays each command line in the procedure as it is executed. The command line is displayed *after* apostrophe and personal command substitution is performed.

Image verification. When this is enabled, DCL displays each image data line as it is read by an executing image. **Image data** is program input data that appears directly in a command procedure.

When verification is enabled, command and data lines are displayed at the terminal (specifically, to SYS\$OUTPUT), intermixed with the normal output from the procedure.

12.1.1 **Enabling Verification before Invoking the Procedure**

The easiest way to verify a procedure is to enable verification before the procedure is invoked. This is accomplished with the SET VERIFY command:

```
$ set verify
```

This simple form of the command will enable both procedure and image verification. To enable only one type of verification, leaving the other type in its current state, use the following commands:

```
$ set verify=procedure
```

–or–

```
$ set verify=image
```

Once the SET VERIFY command is entered, all subsequent procedures and sub-procedures will be verified. To disable verification when the debugging session is over, use the SET NOVERIFY command:

```
$ set noverify
```

12.1.2 **Enabling Verification in a Procedure**

When an entire procedure is verified, the volume of output may be so large that it becomes overwhelming. In this case, you can enable verification for a portion of the procedure, but this requires that you modify the procedure by inserting some temporary command lines. One line is inserted before the commands to be verified, and one after:

```
$ saved_verify = f$verify(true) !!!
      .
      .commands to be verified
      .
$ junk = f$verify(saved_verify) !!!
```

The `F$VERIFY` lexical function enables or disables both procedure and image verification according to its boolean argument, enabling them if the argument is true, disabling if false. The function returns a boolean value that reflects the state of procedure verification *before* the function was called. The `SAVED_VERIFY` symbol is therefore set to the prior state of procedure verification. Note that the function enables or disables both procedure and image verification but returns the prior state of procedure verification only, an interesting inconsistency, which is addressed in the next section. Once verification is enabled, the existing commands are displayed at the terminal as they are executed. Finally, procedure verification is reset to its former state by using the `SAVED_VERIFY` symbol as an argument to the `F$VERIFY` function. In this case, the return value from the function is uninteresting, so it is discarded.

The former state of verification is saved and restored so that the verification of one sequence of command lines is independent of the verification of any other sequence. If, in a subprocedure, verification were enabled and then indiscriminately disabled, any verification in progress in a caller of the subprocedure would be thwarted. The minimal extra work to make each verification independent is well worth the effort.

The two temporary debugging commands are flagged with a triple exclamation point so they can be found and removed when debugging is complete. Triple exclamation points stand out when you read the procedure and are easily located when searching with an editor or the `SEARCH` utility. The author recommends that all debugging lines be flagged with a unique marker of your own choosing.

Some experienced DCL programmers advocate using `SET NOVERIFY` or the `F$VERIFY` function at the beginning of main procedures to disable verification in case the “innocent” user accidentally has it enabled when the procedure is invoked. The author believes this is usually a mistake, because there are legitimate reasons to see the procedure verification. One of the few reasons to disallow verification arises in situations where a user might learn something about system security by seeing the contents of a procedure.

12.1.3

Advanced Verification Features

As mentioned in the previous section, the F\$VERIFY lexical function exhibits some inconsistent behavior. It sets both procedure and image verification according to its argument but only returns the prior state of procedure verification. In fact, F\$VERIFY will accept two arguments, the first specifying the new setting for procedure verification and the second the new setting for image verification:

```
$ saved_verify = f$verify(true, false)
```

In this example, procedure verification is enabled, but image verification is disabled. If F\$VERIFY is given one argument, the argument is used to set the state of both types of verification. If it is given two arguments, then the first is used for procedure verification and the second for image.

This additional feature still does not remove the inconsistency. Regardless of the number and value of its arguments, F\$VERIFY returns only the prior state of procedure verification. If this single value is saved and used to reset verification after a sequence of commands, both procedure and image verification will be reset to the original state of *procedure* verification. The original state of image verification is lost. You can work around this inconsistency by employing the F\$ENVIRONMENT lexical function in addition to F\$VERIFY:

```
$ saved_image = f$environment("VERIFY_IMAGE") !!!
$ saved_proc = f$verify(true) !!!
.
. commands to be verified
.
$ junk = f$verify(saved_proc, saved_image) !!!
```

The additional first line uses the F\$ENVIRONMENT function to obtain the current state of image verification and saves the state in SAVED_IMAGE. The second line enables both types of verification and saves the previous state of procedure verification. The final line correctly restores the state of both types of verification.

Nine times out of ten it is unnecessary to distinguish the two types of verification. The simpler method illustrated in the previous section will almost always suffice.

12.2

Temporary Debugging Output

There are circumstances in which procedure verification is overkill or does not display the necessary information. In particular, verification displays only command lines, not the values of symbols used in those command lines. In the case of assignment or IF commands, the values of symbols in the evaluated expression may be important. Such values are easy to display using temporary debugging lines:

```
$ show symbol count    !!! Count seems to be way off.  
$ blocks = 50 + count*20
```

— or —

```
$ show symbol blocks    !!! Something is wrong.  
$ show symbol threshold !!!  
$ if blocks .lt. threshold then blocks = threshold
```

Any DCL commands that produce pertinent output can certainly be employed to display temporary debugging information. Just remember to remove such commands before installing the DCL application for public use. A marker like the triple exclamation point is invaluable for finding and removing temporary debugging commands.

12.3

Permanent Debugging Output

Sometimes it is advantageous to leave debugging output commands in a procedure even when it is available for public use. This might be true, for example, during the “field test” of a new application. Under normal operation, the procedure must not display the debugging output. But there should be a way to enable debugging output when a problem is suspected. This can be accomplished using a global debug flag.

A global symbol is designated as the debug flag. The debug flag for the example application is XDA_DEBUG. If the symbol is undefined or set to false, debugging output is suppressed. If the symbol is set to true, debugging output is produced. The following initialization lines are needed in the main procedure to establish the debugging environment:

```
$ if f$type(xda_debug) .eqs. "" then xda_debug == false
$ if_debug = "if xda_debug"
```

The first line uses the F\$TYPE lexical function to determine whether the global symbol XDA_DEBUG is defined. If not, it is defined but set false to prevent debugging output. The second line establishes the personal command IF_DEBUG, which is used to prefix debugging output lines.

When a command is prefixed with IF_DEBUG, it becomes conditional, executed only if debugging is enabled:

```
$ if_debug then display "This is some debugging output." !!!
```

After personal command substitution, the line becomes:

```
$ if xda_debug then display "This is some debugging output."
```

Output is produced only if the XDA_DEBUG symbol is true. The symbol is true if it was explicitly set true by a knowledgeable person interested in tracking down a bug. If the symbol is undefined, it will be defined and set false by the first initialization line shown above. The personal command can also be used with a block IF command when multiple debugging lines are needed:

```
$ if_debug !!!
$ then
$ display "This is some debugging output."
$ display "This is some more debugging output."
$ endif
```

Using the scheme presented here, debugging commands can be left in a procedure permanently. They are rendered harmless unless explicitly enabled by an informed user. Even when disabled, however, debugging commands use up some execution time. Procedure execution slows down in proportion to the number of command lines contained in the procedure.

12.4

Capturing Procedure Output

It is difficult to debug a procedure that displays a lot of information quickly. The information tends to flash by, scroll off the terminal screen, and disappear. The problem is exacerbated when a procedure is displaying significant amounts of debugging information intermixed with normal output.

There are two ways to capture the output of a procedure in a text file. The first is to use the /OUTPUT qualifier on the at-sign command that invokes the procedure.

This qualifier specifies a file into which all the output of the procedure and its subprocedures is placed:

```
$ @xda_display/output=debug.lis
```

Any output written to SYS\$OUTPUT is placed instead into the file DEBUG.LIS in the working directory. Note that it does not appear on the screen.

Because the output does not appear on the screen, this method of capturing procedure output may be of little use in debugging an interactive DCL application. The second method of capturing output produces a log of the entire terminal session, while still allowing output to appear on the screen. This method requires the use of DECnet. Use the SET HOST command to log in to your local node, creating a second interactive session:

```
$ set host 0 /log=debug.log
```

The SET HOST command usually creates a connection from the local DECnet node to a remote node, allowing you to log in at the remote node. In this case, however, the remote node is specified as 0. Node 0 is always the local node, so this command creates a second terminal session on the same node. The /LOG qualifier specifies that a log file of the entire session is to be kept in the file DEBUG.LOG in the working directory.

Once the second terminal session is initiated, you can log in and test the application in a normal fashion. All output appears at the terminal; everything behaves as usual. When the second session is terminated, a complete log of keyboard input and screen output can be found in the log file.

Chapter 13



Files and Directories

This chapter begins the discussion of VMS files. A **file** is a collection of information stored on some physical medium, such as magnetic disk or magnetic tape. All permanent information available to a VMS system and its applications ultimately resides in files, so every VMS programmer must understand files. This understanding is even more critical for DCL programmers because DCL applications are often centered on the manipulation of files and their contents.

13.1 File Specifications

A **file specification** is a character string that identifies an individual file or set of files on a VMS system. A similar kind of specification can also be used to name a device or a disk directory, without reference to any particular files. In this book, the term **file spec** refers to all manner of device, directory, and file specifications, even though some do not identify individual files.

The format of a file spec was reviewed in Chapter 1, but it will be presented here in greater detail. A complete VMS file spec has the following format:

node::device:[directory...]name.type;version

13.1.1 Node

The node component serves to specify a file on a particular DECnet node. The name of the node is included in the spec, followed by a double colon to distin-

guish it from a device name. Multiple nodes can be included, each one followed by a double colon. In this case, the final node names the location of the file, with preceding nodes specifying the path to be taken from the local node to the final node. For example, `BOSTON::STLOUS::LADUE::` results in a path from the local node through node `BOSTON`, through node `STLOUS`, and finally to the target node `LADUE`. Whether intermediate nodes must be specified or can be inferred by DECnet depends upon the network organization. Talk to your system manager to find out.

DECnet nodes are actually identified by integers; the node names are a convenience to the user. Do not be surprised if you see file specs with numeric node components, such as `17::` or `12639::`. This usually occurs when information is sent from a remote node and the local node does not know its name. In order to determine the equivalence between names and numbers, you must again consult your system manager.

It is perfectly valid for a file spec to include the local node as its target DECnet node. In this case, DECnet simply makes a connection “to itself” to access the file. The node component is not ignored: a real connection is established, albeit to the local node. Therefore, specifying the local node on a file spec is not without network overhead. Instead of specifying the local node name you can always refer to the local node via the number zero, as in `0::`. This still makes a DECnet connection; it is just easier to specify.

13.1.2 Device

When a file is being accessed, the device component specifies the disk or tape on which the file resides. If the file spec refers to the device by itself and does not contain a directory or file name, then the device can also be a terminal, card reader, or any other device that is not file-structured (i.e., does not contain individual files). Device names have various formats depending upon the type of device, how many of them there are, whether the device is attached to the local VAXcluster node or another one, and so on. It is sheer folly to assume you know the format of a device name.

▷ Ch. 19

It is a common and well-advised practice to refer to devices with logical names. When a procedure refers to a device with a logical name, it is quite easy to substitute one device for another by simply redefining the logical name. If the procedure refers directly to the device, the procedure will have to be modified when a different device is substituted.

The logical name `SYS$DISK` always refers to the disk containing the working directory, as established with the `SET DEFAULT` command.

A handy device is the **null device**, referred to as `NL:.` Any output to this device is discarded. Any attempt to read from the device results in an end-of-file condition, as if it represented an empty file.

13.1.3 **Directory**

The directory component specifies the directory containing the file. A directory is itself a file, but its purpose is to act as a catalog for a collection of files. Every file on a disk is cataloged in some directory (possibly more than one) so that it can be easily located.

Because a directory is a file, it can appear in another directory. This allows a “tree structure” of directories, with each directory containing some files and possibly some other directories, which are called **subdirectories** of the original directory. The original directory is called the parent directory, and the subdirectories are its child directories. The entire directory tree is rooted in a special directory called the **master file directory**. The master file directory has the name `000000` (six zeros). Here is a possible directory tree on a disk referred to by the logical name `DISK1`:

```
disk1:[000000]           ! The master file directory.
disk1:[jones]            ! The directory for user JONES.
disk1:[jones.project]   ! Jones' project directory,
disk1:[jones.project.data] ! with a subdirectory for data.
disk1:[jones.personal]  ! Jones' personal directory.
disk1:[smith]           ! The directory for user SMITH.
.
.
```

The directory portion of a file spec is enclosed in square brackets. Each subdirectory is separated from its parent with a dot. It is not necessary to specify the master file directory except when referring precisely to that directory.

The empty directory specification `[]` stands for the working directory, as established with the `SET DEFAULT` command. When used with the `SYS$DISK` logical name, as in `SYS$DISK: []`, a complete specification of the working disk and directory results. This specification allows you to be explicit in commands that manipulate files in the working directory (see Section 13.6).

The directory specification [.DATA] stands for the DATA subdirectory under the working directory. In general, any subdirectory of the working directory can be accessed by beginning the directory portion of the file spec with a dot.

The directory specification [-] stands for the parent directory of the working directory. The dash can be followed by a subdirectory, as in [-.OTHER], which refers to the directory named OTHER under the parent directory of the working directory (i.e., OTHER is a directory parallel to the working directory).

A little-known fact about directory specifications is that they can also be enclosed in angle brackets, such as in DISK1:<JONES.PROJECT>. This is because angle brackets are international characters and appear on more terminals than do square brackets. Angle brackets must be taken into account when performing various manipulations of file specs, as illustrated in Section 13.2.1.

13.1.4 Name

The name component, taken together with the file type, identifies a particular file in a directory. The purpose of the file name is to assign a meaningful identifier to the file. The name can be composed of letters, digits, and the dollar sign (\$), underscore (_), and hyphen (-) characters. Its length is restricted to a maximum of 39 characters, but that allows plenty of room for meaningful names. Because a directory is just a file, directory names follow the same rules as file names.

13.1.5 Type

The type component is used with the file name to uniquely identify a file in a directory. This two-level name/type scheme allows the file name to identify a family of related files, while the file type identifies particular kinds of files within the family. For example, a small application written in Fortran may include a set of files all having the name BUDGET, but with the following file types:

- FOR. The Fortran source file.
- OBJ. The object file produced by the Fortran compiler.
- LIS. The listing file produced by the compiler.
- EXE. The executable image produced by the linker.
- MAP. The image map produced by the linker.

The standard file types have been chosen over the years to be meaningful abbreviations for the contents of the file. The type OBJ stands for “object” and EXE for “executable.” The restrictions on character set and length for file types are the same as those for file names. Most of the standardized file types are three letters, because they were once limited to three characters in length.

13.1.6

Version

On many computer systems, the device/directory/name/type is sufficient to identify any individual file. Such systems do not allow multiple copies of the same file and thus run into trouble when a modified version of a file is created. The modified version replaces the original version, which cannot be recovered if an error or program bug corrupts the modified version. VMS supports the concept of multiple file generations or versions. More than one version of the “same file” can exist in a given directory. The term *same file* is in quotes because each version is an autonomous file, entirely distinct from the other versions, but all the versions represent successive generations of the same information.

Every file is assigned a version number. When a file with a new name and type is created in a directory, it receives version number 1. When another file with the same name and type is created in that directory, it receives version number 2. The version 1 file is not deleted. Whenever a new generation of an existing file is created, it is assigned the first available version number that is higher than all of its ancestors in the directory. Version numbers range from 1 to 32767.

An existing version of a file is accessed by including its unique version number in the file spec. For example:

```
$disk1:[smith]login.com;3
```

This file spec accesses version 3 of the file LOGIN.COM. If version 3 does not exist, an error is signaled. It does not matter which other versions exist.

Because certain versions of a file, most notably the latest one, are accessed frequently, VMS provides some special version number syntax. If an existing file is being accessed and the version number is missing or specified simply as a semicolon (e.g., LOGIN.COM or LOGIN.COM;) or as version number 0 (e.g., LOGIN.COM;0), the latest version of the file is accessed. This is the file with the highest version number. If the version number is specified as -1 (e.g., LOGIN.COM;-1), then the second-latest version of the file is accessed. This is the file with the next highest version number (not necessarily 1 less than the highest). Versions numbers -2, -3, and so on are also allowed, and access successively

older versions of the file. The version number -32768 can be used to access the file with the lowest version number (e.g., LOGIN.COM;-32768). The lowest version number is not necessarily 1, because some of the original versions of the file may have been deleted.

13.2 **Parsing**

The process of preparing a file spec to access a directory or file is called **parsing** the file spec. The goal of parsing is to take a file spec that may have some missing components and flesh it out into a complete spec with all components included. VMS parses a file spec whenever the spec is used to access a file for any kind of input, output, or control operation. The complete file spec returned by the parsing operation is called the **resultant file spec**. The algorithm used to parse a spec is made available to the DCL programmer through the F\$PARSE lexical function.

The general form of the lexical function is shown here:

```
f$parse(base-spec,default-spec,related-spec,component,options)
```

The only required argument is the first one; the rest are optional. The *base-spec* is the file spec to be parsed and expanded into a complete file spec. It may be missing certain components, such as the directory or version number. Rarely does a user enter a complete file spec with all components specified explicitly. The function parses the base file spec, expands it into a resultant file spec with all components, and returns part or all of the resultant file spec. The parsing algorithm can make use of two additional file specs, the *default-spec* and the *related-spec*. These files specs may be complete, or, as with the base spec, they may include only some components. The parsing algorithm is as follows:

1. Examine the base spec. If any components are missing, use the components from the default spec (if it was specified).
2. If components are still missing, use the components from the related spec (if it was specified).
3. If any of the node, device, or directory components are still missing, use the components from the user's current working directory.
4. If any components are missing from this complete spec, do not fill them in.
5. If the *component* argument is not specified, return the complete resultant file spec. If it is specified, extract the specified component and return it by itself. The component argument is one of the following keywords: "NODE", "DEVICE", "DIRECTORY", "NAME", "TYPE", "VERSION".

To summarize, the base spec is completed using three levels of defaults: the default spec, the related spec, and the working directory. The lexical function then returns either the entire resultant spec or an individual component of it. (The complete algorithm used by the parse system service is actually more complicated because of logical names.)

In the process of parsing the file spec, VMS checks for the existence of the device and directory. If either does not exist, the parse fails and the lexical function returns the null string. You may wish to disable this checking, particularly when assembling a file spec that will be used to create a new directory. To disable the check, specify the keyword string "SYNTAX_ONLY" as the *options* argument to the lexical function. This option also results in a faster parse operation.

Here are examples of file spec parsing. The examples assume that a file spec has been obtained from the user and saved in the symbol USER_SPEC.

```

$! Parse the file spec, filling in the device and directory
$! from the user's working one.
$
$   parsed_spec = f$parse(user_spec)

-or-

$! Parse the file spec, filling in unspecified components
$! from the default spec SYS$SCRATCH:XDA_DATA.TMP
$
$   parsed_spec = f$parse(user_spec,"sys$scratch:xda_data.tmp")

-or-

$! Create a spec for the new directory [XDA_WORK] on the
$! working disk. Make sure VMS doesn't check for prior
$! existence of the directory.
$
$   dir = f$parse("[xda_work]",,,,"SYNTAX_ONLY")
$   dir = f$parse(dir,,,"DEVICE") + f$parse(dir,,,"DIRECTORY")

-or-

$! Obtain the file name and type from the spec.
$
$   name = f$parse(user_spec,,,"NAME")
$   type = f$parse(user_spec,,,"TYPE")

```

When you obtain an individual component from the resultant spec, as in the last example above, the resulting string contains the punctuation that accompanies the component. A node contains the double colon, a device the single colon, a directory the brackets, a file type the dot, and a version number the semicolon. The file name does not include any punctuation.

13.2.1 **Parent and Child Directories**

There are two file spec manipulations that cannot be performed by the F\$PARSE lexical function. The first one takes a directory spec and a subdirectory name and produces the merged spec naming the subdirectory. Given an arbitrary file spec containing the directory [JONES.PROJECT] and the subdirectory name DATA, it must produce [JONES.PROJECT.DATA]. Assume that the symbol SPEC contains a directory spec and the symbol CHILD contains the name of the desired subdirectory. The merge is easily accomplished with the following commands:

```
$ parent = f$parse(spec,,,"DIRECTORY")
$ subdir = "[" + (parent - "["-"]"- "<">") + "." + child + "]"
```

The first line extracts the directory portion of the file spec. The second line strips the brackets off the directory (be they square brackets or angle brackets) and builds a new directory spec identifying the desired subdirectory. The resulting directory spec uses square brackets regardless of the brackets present in the original spec.

Another file spec manipulation that you might be tempted to perform is that of determining the parent of a given arbitrary directory. This manipulation is virtually impossible and should be avoided at all costs. Almost insurmountable problems arise when the file spec includes a rooted directory.

▷ Ch. 14

13.2.2 **Changing the File Type**

The individual files in a related family of files often have the same name but different types. For example, a Fortran source file has the type FOR, while the object and listing files generated by the compiler have the types OBJ and LIS. The Fortran compiler must construct the object and listing file specs by replacing the file type of the source file with the new types OBJ and LIS. This “retyping” operation is common in sophisticated procedures.

The F\$PARSE function does the work quite easily:

```
$ source_file = f$parse(source_file)
$ obj_file = f$parse(".OBJ",source_file)
$ lis_file = f$parse(".LIS",source_file)
```

The source file spec is parsed first in order to fill in missing components from the working directory. The resultant file spec is stored back in the same symbol. The related object file spec is created by parsing a base spec containing only the desired file type, using the source spec to fill in all other components. The listing file spec is created similarly. Note that the base specs .OBJ and .LIS include the dot punctuation for file types. If they did not, the strings OBJ and LIS would be mistaken for file *names*.

This same technique can be used to replace any other component of a file spec.

13.3 **Searching Directories**

A VMS user frequently needs to locate an individual file or group of files in a directory. Sometimes the user is not sure how a file name is spelled or which directory the file is in. The **wildcard** facility allows the user to search a directory or directory tree for a set of files. The wildcard facility provides special characters, which can be inserted in a normal file spec at various points. A file spec containing wildcard characters is presented to VMS and a **search operation** is requested. Instead of locating a single file, as it does when given a normal file spec, VMS locates one or more files whose specs match the wildcard spec.

A file spec can include the following wildcard characters:

- * An asterisk can be used in the directory name, file name, type, or version. It matches any number of characters, including none.
- % A percent sign can be used in the directory name, file name, or type. It matches exactly one character.
- ... An ellipsis can be appended to a directory name in a directory spec. It matches the entire directory subtree below the named directory.

The simplest way to perform a search operation is with the DIRECTORY command. This command accepts a wildcard file spec and lists all of the files matching the spec:

```
$ directory *.dat
```

–or–

```
$ directory $disk1:[smith...]xda*.com
```

–or–

```
$ directory [project*...]*.%
```

The file spec in the first example matches all the files with type DAT in the working directory, regardless of name. The asterisk in the name position matches any characters and therefore any name.

The file spec in the second example searches the subdirectory tree under the [SMITH] directory of disk \$DISK1. The ellipsis specifies that the subdirectory tree is to be searched. The spec matches any files in the subdirectory tree whose name begins with XDA and whose type is COM.

The file spec in the third example causes a search of the subdirectory trees under any directory beginning with PROJECT. The directory spec PROJECT* matches any directory beginning with PROJECT and the ellipsis causes the subdirectories under those directories to be searched. The spec matches any file regardless of name, as long as the type is exactly one character in length.

As you can see, wildcards provide a powerful tool for searching directories and selecting files within them.

DCL provides a lexical function that a procedure can use to perform directory searches. The F\$SEARCH lexical function requires one argument, which must be a file spec. The spec can name a single file or it can contain wildcards. When F\$SEARCH is called with a file spec that does not contain wildcards, it determines whether or not the specified file exists. First it parses the file spec using the working device and directory as the default. It then checks to see whether the file exists, and if so, returns the resultant, fully expanded file spec. If the file spec is invalid or the file does not exist, F\$SEARCH returns the null string.

Here is an example using F\$SEARCH to check whether a specified file exists:

```
$10:  libcall ask xda_file s "Enter a file spec:"
$     if f$search(xda_file) .nes. "" then goto 19
$     libcall signal xda w filnotfnd -
           "File ''xda_file does not exist."
$     goto 10
$19:
```

This procedure asks the user for a file spec. It then uses the F\$SEARCH function to check for the existence of the file. If the function returns an expanded file spec, the file exists. If the function returns the null string, a message is displayed and the user is asked for another file spec. Note that this procedure does not distinguish the case of a nonexistent file from the case of an invalid file spec. If you want to distinguish these two cases, use the F\$PARSE function in conjunction with F\$SEARCH:

```

$10:  libcall ask xda_file s "Enter a file spec:"
$      if f$parse(xda_file,,,"SYNTAX_ONLY") .nes. ""
$      then
$          if f$search(xda_file) .nes. "" then goto 19
$          libcall signal xda w filnotfnd -
$              "File ''xda_file does not exist."
$      else
$          libcall signal xda w invspec -
$              "File spec ''xda_file is invalid."
$      endif
$      goto 10
$19:

```

This procedure first checks the validity of the file spec with the F\$PARSE function. If the spec is valid, it goes on to search for the file. If the spec is invalid, it displays a message and requests another spec.

When the F\$SEARCH function is called with a wildcard file spec, it searches for the first file matching the spec and returns its resultant spec. If there are no files matching the spec, the null string is returned. An additional matching file spec is obtained by calling F\$SEARCH again *with the same wildcard spec*. Each call returns the next matching file spec. When there are no more matching files, the null string is returned. The following code displays the name of every data file in the working directory, each on its own line:

```

$10:  file = f$search("*.dat;")
$      if file .eqs. "" then goto 19
$      display f$parse(file,,"NAME")
$      goto 10
$19:

```

The F\$SEARCH function is called with the wildcard spec `"*.DAT;"`. The asterisk matches every file name. A semicolon is specified for the version number, so that only the latest version is located. If no version number is given, F\$SEARCH locates all versions.

The following procedure asks the user for a file spec and counts the number of files matching the spec:

```
$ libcall ask xda_file s "Files to count:"
$ count = 0
$10: file = f$search(xda_file)
$ if file .eqs. "" then goto 19
$ count = count + 1
$ goto 10
$19:
$ display "Count of files found: ", count
```

There is a special precaution you must take when programming a search loop if you are not sure whether the file spec will contain wildcards. The preceding example loops forever if the user enters a file spec without wildcards, because the F\$SEARCH function never returns the null string. It continually locates the single file and returns its resultant spec. To prevent an infinite loop, extra code is needed:

```
$ libcall ask xda_file s "Files to count:"
$ count = 0
$10: file = f$search(xda_file)
$ if file .eqs. "" then goto 19
$ count = count + 1
$ if f$locate("*,xda_file) .ne. f$length(xda_file) .or. -
    f$locate("%",xda_file) .ne. f$length(xda_file) .or. -
    f$locate("...",xda_file) .ne. f$length(xda_file) then -
    goto 10
$19:
$ display "Count of files found: ", count
```

The procedure no longer performs an unconditional loop back to label 10. Instead, it checks to see if the file spec contains any wildcard characters and only loops back if it does. The loop is executed exactly once for file specs without wildcards, multiple times for file specs with wildcards.

13.3.1

Simultaneous Searches

The F\$SEARCH function begins a new search whenever it is presented with a new wildcard file spec. In the previous examples, the same file spec was used on each call to F\$SEARCH, so the function carried on the search from the previous call. Once there are no more matching files, the function returns the null string and resets its search context so that the next call will begin a new search regardless of the file spec. This is reasonable behavior, which you can rely on whenever you are performing one search at a time. However, when multiple searches must be performed simultaneously, another feature of F\$SEARCH is required.

The F\$SEARCH function accepts a second argument, called the **stream ID**. The stream ID is an arbitrary integer that identifies a particular search context and distinguishes it from other search contexts that may be in progress simultaneously. Whenever the F\$SEARCH function is used to perform more than one search at a time, a unique stream ID is required for each different search. There are two common cases where multiple searches are performed simultaneously:

- A single procedure needs to match two sets of files at the same time. For example, a procedure searches the master file directory of a disk for every top-level directory beginning with PROJECT. Then, for each such directory, the procedure locates all the data files in that directory tree. Two searches are progressing in parallel: the search for top-level project directories and the search within each such directory.
- A procedure locates each file that matches a wildcard spec and then calls another procedure that operates on the file. If the second procedure uses F\$SEARCH as part of its file processing, then two searches are in progress simultaneously.

Whenever multiple searches are in progress at the same time, each one must use a different stream ID in its calls to F\$SEARCH.

The following example illustrates the first scenario:

```
$10:  dir = f$search("$disk1:[000000]project*.dir;",10)
$     if dir .eqs. "" then goto 19
$     file_spec = "$disk1:[" + f$parse(dir,,,"NAME") + "...]*.dat;"
$20:  file = f$search(file_spec,11)
$     if file .eqs. "" then goto 29
$     display file
$     goto 20
$29:  goto 10
$19:
```

The outer loop searches for directories whose names begin with PROJECT. The stream ID for the directory search is 10. The inner loop searches for data files in the directory tree found by the outer loop. The stream ID for the file search is 11. The file spec for the inner search is constructed using the name of a directory found by the outer search. For example, if directory PROJECT_DRIVER is found, the file spec \$DISK1:[PROJECT_DRIVER...]*.DAT; is constructed. This spec will locate all the data files in the directory tree under the PROJECT_DRIVER directory.

Each stream ID is an arbitrary positive integer. The only requirement is that each simultaneous search use a different ID.

13.4 **File Protection**

VMS provides a comprehensive scheme for protecting files from unauthorized access. This scheme is called **file protection** and is complex enough to require its own chapter (Chapter 18).

Table 13.1 F\$FILE_ATTRIBUTES Items

Keyword	Type	Description of Result
"BDT"	String	Backup date/time.
"CDT"	String	Creation date/time.
"EDT"	String	Expiration date/time.
"EOF"	Integer	Number of blocks occupied by data.
"MRS"	Integer	Maximum record size.
"ORG"	String	Organization: "SEQ" for sequential, "REL" for relative, "IDX" for indexed.
"PRO"	String	Protection mask.
"RDT"	String	Revision date/time.
"RVN"	Integer	Number of times file has been modified.
"UIC"	String	Owner UIC string.

13.5

File Attributes

Each file has a set of descriptive information associated with it. These items of information are called **file attributes**. The file attributes include some well-known items like the file's creation time and owner UIC. There are also many items that are rarely used by DCL procedures.

The F\$FILE_ATTRIBUTES lexical function can be used to obtain the attributes of a file. It requires two arguments: a file spec and an item keyword. The file spec must identify a single existing file. The item keyword is chosen from among a fixed set of keywords that specify the file attributes. Table 13.1 describes some of the common file attributes.

The following example requests a file spec from the user and displays important dates associated with the file.

```

$10:  libcall ask xda_file s "File spec:"
$      if f$search(xda_file) .nes. "" then goto 19
$      libcall signal xda w filnotfnd -
           "File 'xda_file does not exist."
$      goto 10
$19:
$      display "Creation time:  ", f$file_attributes(xda_file,"CDT")
$      display "Revision time:  ", f$file_attributes(xda_file,"RDT")
$      display "Revision count: ", f$file_attributes(xda_file,"RVN")
$      display "Backup time:    ", f$file_attributes(xda_file,"BDT")
$      display "Expiration time: ", f$file_attributes(xda_file,"EDT")

```

After asking the user for a file spec, the procedure uses the F\$SEARCH function to check that the file exists. If not, the user is asked again. The check is important because the F\$FILE_ATTRIBUTES function signals an error if given a file that does not exist.

A complete list of the attributes of a file can be displayed with the DCL command DIRECTORY/FULL.

13.6 **File Operations**

VMS provides a comprehensive set of utilities for manipulating files and directories. These utilities are described in detail in the *VMS DCL Dictionary*, and you are advised to consult that book to learn about the utilities and their various features. The *Dictionary* will sometimes refer you to another VMS document when the utility is complicated and requires many pages of description. Table 13.2 summarizes the operations performed by the file utilities. The descriptions are given in terms of a single file, but most utilities can operate on a set of files using wildcard file specs.

Table 13.2 VMS File Utilities

Command	Description
ANALYZE/RMS_FILE	Allows you to interactively peruse the structure and contents of a file. Has a mode that checks the structural integrity of a file. Has another mode that generates a File Definition Language description of the file.
APPEND	Appends a file to the end of an existing file.
BACKUP	Performs various kinds of disk and file backup operations.
CONVERT	Copies one file to another, changing the organization and structure of the file in the process.
COPY	Makes a copy of a file, placing the new file on the same disk or another one. Can also concatenate a set of files into one output file.
CREATE	Creates a sequential file.
CREATE/DIRECTORY	Creates a new directory on a disk.
DELETE	Deletes a file.
DIFFERENCES	Determines the differences between the contents of two files.
DIRECTORY	Displays a list of files and various file attributes.
MERGE	Combines two or more sorted files, producing an output file with all records sorted.
PURGE	Deletes older (lower-numbered) versions of a file, leaving only the newest version or versions.
RENAME	Changes a file's name or type. Can also move a file from one directory to another on the same disk.
SEARCH	Searches the contents of a file for records matching certain patterns and displays the matching records.
SET DEFAULT	Changes the working device and directory.
SET DIRECTORY	Changes the attributes of a directory.
SET FILE	Changes the attributes of a file.
SHOW DEFAULT	Displays the working device and directory.
SORT	Sorts the records in a file according to various criteria, producing a new file of sorted records.
TYPE	Displays the contents of a file.

Table 13.3 File Selection Qualifiers

Qualifier	Description
<code>/BACKUP</code>	Selects files based on their backup time. Used in conjunction with the <code>/BEFORE</code> and <code>/SINCE</code> qualifiers.
<code>/BEFORE=time</code>	Selects files that are dated before a given time. Used in conjunction with the <code>/BACKUP</code> , <code>/CREATED</code> , <code>/EXPIRED</code> , or <code>/MODIFIED</code> qualifiers.
<code>/BY_OWNER=uic</code>	Selects files owned by a particular UIC.
<code>/CREATED</code>	Selects files based on their creation time. Used in conjunction with the <code>/BEFORE</code> and <code>/SINCE</code> qualifiers.
<code>/EXCLUDE=file-spec</code>	Does not select files that match the excluded file spec. A list of file specs may be specified, and they may contain wildcards.
<code>/EXPIRED</code>	Selects files based on their expiration time. Used in conjunction with the <code>/BEFORE</code> and <code>/SINCE</code> qualifiers.
<code>/MODIFIED</code>	Selects files based on their revision time. Used in conjunction with the <code>/BEFORE</code> and <code>/SINCE</code> qualifiers.
<code>/SINCE=time</code>	Selects files that are dated after a given time. Used in conjunction with the <code>/BACKUP</code> , <code>/CREATED</code> , <code>/EXPIRED</code> , or <code>/MODIFIED</code> qualifiers.

In addition to wildcard file specs, file utilities accept a standard group of command qualifiers, which further select and restrict the target files. These qualifiers provide a powerful facility for selecting files according to single or multiple criteria. Table 13.3 describes these file selection qualifiers.

- ▷ Ch. 15 VMS also has a set of commands for creating, reading, and writing files from a DCL procedure.
- ▷ Ch. 17



Logical Names

A **logical name** is a named entity that you can create and assign a value. The name then stands for the value when the logical name is used in certain contexts, such as file specifications. In these contexts, VMS *automatically* replaces the logical name with its value. A logical name may appear to be the same as a symbol, but there are important differences:

- Logical names are created, maintained, and deleted using a different set of VMS commands from those used for symbols.
- Logical names reside in **logical name tables**. A given logical name table can be made available to a single process, a family of processes, or every process on the system. In this way, unlike symbols, logical names can be shared by multiple processes.
- When a logical name appears in an appropriate context, such as a file spec, VMS uses its value automatically. This differs from apostrophe substitution, which must be requested explicitly, although it is similar to personal command substitution.

The most common use for a logical name is to stand for part or all of a file spec. For example, the logical name SYS\$HELP stands for the system device and directory containing the help libraries and release notes for VMS and its layered products. If you want to type the release notes for DEC/CMS Version 2.2, you can use the following command:

```
$ type sys$help:cms022.release_notes
```

The logical name SYS\$HELP is used in place of an explicit device and directory to refer to the file. VMS automatically uses the value of the logical name to complete the file spec and locate the release notes.

One purpose of logical names in file specs is to save the user from having to remember the device and directory. Another purpose is to allow the contents of a directory to be moved to a new directory. Once the logical name is redefined to refer to the new directory, users can work as usual, unaware that a change has been made. This technique will only succeed, however, as long as users always refer to files in the directory via the logical name.

14.1 **Defining and Using Logical Names**

A logical name is created with the DEFINE command. The basic form of the DEFINE command is as follows:

```
$ define logical-name value
```

The first parameter specifies a sequence of characters that name the logical name. The name can consist of letters, digits, dollar sign (\$), and underscore (_). The second parameter specifies the value of the logical name. The value can be composed of any characters and must be enclosed in double quotes if it contains characters other than those that may appear in a file spec. The value is sometimes called the **equivalence string**.

When a logical name is defined for use in file specs, its value can comprise any or all of the components of the file spec. Here are some related examples:

```
$! Define a logical name for the disk used by the marketing
$! department. Such definitions are rarely necessary, since a
$! logical name is automatically defined when a disk is mounted.
$
$ define disk_market dub1:
$
$! Define a logical name for the directory on the marketing disk
$! which contains monthly reports. Notice how one logical name
$! can be defined in terms of another. Define a synonym for the
$! logical name.
$
$ define market_reports disk_market:[reports]
$ define market_documents market_reports
```

```

$!   Define a logical name for the latest monthly report file.
$
$   define market_latest_report market_reports:monthly.txt;

```

In each of these examples, the value of the logical name consists of some or all of the components of a file spec: the device; the device and directory; the device, directory, file name, type, and version. The components may be specified explicitly or derived in turn from other logical names. Once a logical name is defined, a file can be accessed using a file spec beginning with the logical name. In this case, VMS looks up, or **translates**, the logical name and uses its value to obtain components missing from the file spec. If the value of the logical name contains a second logical name, VMS uses the value of the second name to obtain components for the first, and then uses all the components in the final file spec. This is called **iterative translation**. All of this logical name translation is performed by the VMS file spec parsing service, available to the DCL programmer through the F\$PARSE lexical function. Section 14.2 describes the complete file spec parsing algorithm.

▷ Ch. 13

Here are some sample file specs, shown with the final spec resulting from a parsing operation:

```

disk_market:[archive]history.dat
DUB1:[ARCHIVE]HISTORY.DAT;

```

-or-

```

market_reports:history.dat
DUB1:[REPORTS]HISTORY.DAT;

```

-or-

```

market_documents:history.dat
DUB1:[REPORTS]HISTORY.DAT;

```

-or-

```

market_latest_report
DUB1:[REPORTS]MONTHLY.TXT;

```

The value of a logical name can include components that are not necessarily adjacent in the file spec. For example:

```

$!   Define a logical name for the monthly report directory and
$!   include the file type TXT to refer to text files without
$!   a specific name.
$
$   define market_report_texts market_reports:.txt

```

This logical name can be used to refer to all the text files in the reports directory, or to a specific one:

```
market_report_texts  
DUB1: [REPORTS].TXT;
```

- or -

```
market_report_texts;-1  
DUB1: [REPORTS].TXT;-1
```

- or -

```
market_report_texts:summary  
DUB1: [REPORTS]SUMMARY.TXT;
```

Every logical name resides in a logical name table. Section 14.10 describes logical name tables in detail. There are four logical name tables accessible to every process:

Process. Each process has a private logical name table whose logical names are used only by the process. The DEFINE/PROCESS command enters logical names in the process table. If no table qualifier is specified on the DEFINE command, logical names are entered in the process table by default.

Job. Each job has a logical name table whose logical names are available to the processes in the job. A job consists of the detached process created when you log in to VMS, plus any subprocess created by the main process. The DEFINE/JOB command enters logical names in the job logical name table.

▷ Ch. 20

Group. All processes belonging to a particular UIC group have a group logical name table whose logical names are shared by those processes. This allows processes in the same group to communicate with one another via logical names. The DEFINE/GROUP command enters logical names in the group logical name table.

System. There is a single system logical name table shared by all processes on the system. Logical names needed by every process reside in the system logical name table. This includes the standard VMS logical names and those defined by Digital and third-party software products. The DEFINE/SYSTEM command enters logical names in the system logical name table.

Users can define logical names in their process or job tables. The GRPNAM or SYSPRV privilege is required to define group logical names. The SYSNAM or SYSPRV privilege is required to define system logical names.

When you use the `DEFINE` command to define a logical name that already exists, VMS issues a message to remind you that the old value is being superseded. Such a message can confuse the user if it appears during the execution of a command procedure. The `/NOLOG` qualifier suppresses the message.

The `ASSIGN` command is a second command that can be used to define logical names. It expects its parameters in reverse order: first the equivalence string and then the logical name. The order of parameters to the `DEFINE` command is more natural, so people occasionally reverse the parameters to the `ASSIGN` command by mistake. The author suggests that you always use the `DEFINE` command.

14.2 **Complete File Spec Parsing Algorithm**

Chapter 13 included a simplified description of how the parse system service parses a file spec. A parsing operation is performed whenever a file spec must be prepared to access a file or group of files. A parse operation can be explicitly requested by the DCL programmer through the `F$PARSE` lexical function. In either case, parsing involves up to three file specs: the primary spec, which is to be fleshed out to a full spec; a default spec, which provides components missing from the primary spec; and a related spec, which provides additional missing components.

Once logical names are introduced, the parsing algorithm becomes more complicated, because logical names need to be translated and their values used in the file spec. Here is a slightly simplified description of the complete file spec parsing algorithm:

- Begin with the primary file spec:
 - If a node name is present and is a logical name, replace the logical name with its translation. If a node name is still present, then the spec must be parsed on the remote node. This procedure is described below.
 - Otherwise, if a device name is present and is a logical name, replace the logical name with its translation.
 - Otherwise, if the primary spec is composed of a logical name by itself, replace the logical name with its translation.
- Obtain missing components from the default spec:
 - Translate any logical names in the default spec, but do not worry about node names that may appear.

- If any components are missing from the primary spec, use the corresponding components from the default spec (if they are included).
- Obtain additional missing components from the related spec in the same manner as the default spec.
- Obtain any missing device and directory components as follows:
 - Fill in any missing device component from the value of the logical name SYS\$DISK.
 - Fill in any missing directory component from SYS\$DISK or, if it does not include a directory, from the working directory.

When the primary file spec includes a node component, then the file spec must be parsed on the remote node. Components missing from the primary spec are filled in from the default and related specs, but *without translating any logical names*. If the primary spec is being used to access a file, it is then transmitted to the remote node for final parsing and processing. If the primary spec is being parsed with the F\$PARSE lexical function, the partially parsed spec is simply returned without further processing.

14.3 **Displaying Logical Names**

The value of a logical name can be displayed with the SHOW LOGICAL command. This command displays the logical name and its equivalence string:

```
$ show logical market_reports
  "MARKET_REPORTS" = "DISK_MARKET:[REPORTS]" (LNM$PROCESS_TABLE)

-or-

$ show logical market_documents
  "MARKET_DOCUMENTS" = "MARKET_REPORTS" (LNM$PROCESS_TABLE)
1 "MARKET_REPORTS" = "DISK_MARKET:[REPORTS]" (LNM$PROCESS_TABLE)
```

If the value of a logical name is another logical name, iterative translation is performed and the value of the second logical name is displayed in turn. Therefore, you can see the true value of a logical name when synonyms have been defined.

By default, the SHOW LOGICAL command looks for the specified logical name in all four logical name tables in order: process, job, group, system. It displays the translation for each occurrence of the logical name. The logical name table qualifiers /PROCESS, /JOB, /GROUP, and /SYSTEM can be used to restrict the search to a particular logical name table. For example, the logical name SYS\$DISK resides

in both the process and system logical name tables. The following command restricts the display to the logical name in the process table:

```
$ show logical/process sys$disk
  "SYS$DISK" = "DUA2:" (LNM$PROCESS_TABLE)
```

An asterisk (*) can be used in the logical name as a wildcard to specify all the logical names beginning with a certain string:

```
$ show logical market*

(LNM$PROCESS_TABLE)

  "MARKET_DOCUMENTS" = "MARKET_REPORTS"
  "MARKET_LATEST_REPORT" = "MARKET_REPORTS:MONTHLY.TXT;"
  "MARKET_REPORTS" = "DISK_MARKET:[REPORTS]"
  "MARKET_REPORT_TEXTS" = "MARKET_REPORTS:.TXT"

(LNM$JOB_802EA710)

(LNM$GROUP_000260)

(LNM$SYSTEM_TABLE)
```

When an asterisk is used in the logical name, iterative translation is not performed. Only one level of translation is performed, so that each logical name is displayed with its own value rather than with the value obtained by translating through any synonyms. The display is therefore easier to comprehend, because it illustrates the relations among all the logical names shown.

14.4

Access Modes

A VMS process can run in any of four **access modes** supported by the VAX architecture. The four access modes range from least privileged to most privileged, as follows:

User. Most VMS utilities and user-written programs run in user mode.

Supervisor. Command language interpreters like DCL run in supervisor mode.

Executive. The RMS record management system runs in executive mode.

Kernel. The VMS operating system runs in kernel mode.

When a logical name is defined, it is associated with a particular access mode. A given logical name table can contain multiple definitions for the same logical name in different access modes. Table 14.1 describes the commands used by a DCL procedure to define a logical name at each of the access modes. When

Table 14.1 Defining Logical Names in Different Modes

Access Mode	Command	Privileges
User	DEFINE/USER_MODE	(none)
Supervisor	DEFINE/SUPERVISOR_MODE (default)	(none)
Executive	DEFINE/EXECUTIVE_MODE	SYSNAM
Kernel	Cannot be created from DCL.	—

a request is made to translate a logical name, the access mode plays a role in determining which logical name is actually translated:

- If the translation request does not include a specific access mode, then the logical name table is searched for a logical name in user mode, then supervisor, executive, and kernel modes.
- If the request includes a specific access mode, the search begins with that mode, then the next most privileged, then the next, and so on. Logical names at less privileged modes are ignored.

The four logical name access modes are employed by VMS in the following manner:

User. A logical name defined in user mode exists until the next program image completes. All user mode logical names are deleted during image exit. Thus, user mode logicals provide a means of overriding, for the duration of one program, a logical name defined at a more privileged mode.

Supervisor. When a logical name is defined with the `DEFINE` command, it is entered in supervisor mode by default. These logical names exist for the duration of the process.

Executive. Many of the logical names defined by the RMS file system are entered in executive mode. These include the process-permanent logical names, such as `SYS$INPUT`. They also include the family of logical names that refer to system directories (e.g., `SYS$SYSTEM`).

Kernel. Logical names referring to logical name tables are often defined in kernel mode.

Table 14.2 Standard Process and Job Logical Names

Logical Name	Table	Description
SYS\$COMMAND	Process	The original source of commands for the process.
SYS\$DISK	Process	The device containing the process's current working directory. This logical name is redefined whenever a SET DEFAULT command is issued. The working directory can be accessed using the file spec SYS\$DISK: []
SYS\$ERROR	Process	The device or file to which VMS displays messages.
SYS\$INPUT	Process	The default input source for the process.
SYS\$LOGIN	Job	The login device and directory for the process. A reference to the login directory can be made with this logical name regardless of the current working directory.
SYS\$OUTPUT	Process	The default output destination for the process.
SYS\$SCRATCH	Job	The default device and directory in which temporary files are created. A procedure should always create temporary files in this directory.

14.5 **Standard Logical Names**

When you log in to VMS, certain process and job logical names are automatically defined. These are described in Table 14.2.

The process-permanent logical names SYS\$COMMAND, SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR are extremely important to the operation of command procedures. They specify the standard input sources and output destinations for the procedures.

▷ Ch. 20

14.6 **Other Uses for Logical Names**

The following sections describe some additional uses for logical names. These uses clarify the distinction between logical names and symbols.

14.6.1 **Root Directories**

A **root directory** is a directory whose fundamental purpose is to act as the parent for a related set of subdirectories. The root directory may contain a few data files, but it exists primarily to “root” a collection of subdirectories that contain the files for a software product or application. The root directory allows the files to be accessed or manipulated through a single known point rather than through an unrelated collection of directories.

A logical name is used to refer to the root directory. It is defined as follows:

```
$ define xda_root disk$products:[xda.]
```

The XDA directory on disk DISK\$PRODUCTS is the root directory for the XDA application. The fact that it is a root directory is indicated by the trailing period in the directory portion of the spec. The trailing period is required when defining a logical name for a root directory. Assume that three subdirectories exist under the root directory: PROGRAMS, DATA, and USER. A reference to one of the directories can be made using the logical name XDA_ROOT:

```
$ directory xda_root:[user]
```

```
Directory DISK$PRODUCTS:[XDA.] [USER]
```

```
USER-DATA1.DAT      USER-DATA2.DAT      USER-DATA3.DAT
:
:
```

Notice how VMS displays the directory spec. There are two directory components, [XDA.] representing the root directory and [USER] representing the subdirectory. This double directory syntax appears whenever rooted directories are used, and is another reason why file specs must only be parsed with the F\$PARSE lexical function, which knows how to deal with the double directory.

▷ Ch. 13

It is possible to hide the equivalence string of the logical name by using the /TRANSLATION_ATTRIBUTES qualifier on the DEFINE command:

```
$ define xda_root -
      disk$products:[xda.]/translation_attributes=concealed
```

The CONCEALED attribute specifies that the equivalence string is not to be displayed. Instead, VMS displays the root logical name without translation:

```

$ directory xda_root:[user]

Directory XDA_ROOT:[USER]

USER-DATA1.DAT;1      USER-DATA2.DAT;1      USER-DATA3.DAT;1
.
.
.

```

The CONCEALED attribute is associated with the equivalence string, not with the logical name as a whole. This is the reason the qualifier appears after the equivalence string in the DEFINE command.

In order to refer to the root directory itself, you use the convention adopted for referring to the master file directory of a disk. The special directory name 000000 is reserved to signify that the directory reference is to the root directory (not to a subdirectory named 000000):

```

$ directory xda_root:[000000]

Directory XDA_ROOT:[000000]

DATA.DIR;1      PROGRAMS.DIR;1      USER.DIR;1

Total of 3 files.

```

The root directory contains the three subdirectories mentioned above.

14.6.2 **User Addresses**

A **user address** is a string that identifies the location and name of a VMS user. For example, NODE1::SMITH specifies that user SMITH resides on DECnet node NODE1. VMS utilities such as MAIL and PHONE require user addresses as the destination of mail messages and phone calls. Utilities that work with user addresses are designed to accept a logical name in place of an explicit user address. When a utility receives a logical name, it translates the name and uses the resulting equivalence string as the user address.

Because a logical name can be used in place of an address, you can define personal logical names for your friends and associates:

```
$ define Bill node1::node2::Smith
$ define Cynthia node13::Jones
$ define Joe node42::Taylor
```

Once these logical names are defined, you can use them with the PHONE and MAIL utilities:

```
$ phone cynthia
```

— or —

```
$ mail message.txt bill,joe /subject="Here is the info I promised."
```

14.6.3 Search Lists

A logical name with more than one equivalence string is called a **search list**. Here is a logical name used to search two of the XDA subdirectories:

```
$ define xda_files xda_root:[data],xda_root:[user]
```

The two equivalence strings are separated by a comma. Once this logical name is defined, it can be used in a file spec just like any other logical name. When the file spec refers to a single file, VMS searches the directories in order until the first matching file is located. Once it is located, the search stops:

```
$ type xda_files:user-data2.dat
:
: contents of file XDA_ROOT: [USER] USER-DATA2.DAT;1
:
```

On the other hand, if the file spec contains wildcards, thus naming multiple files, VMS will process all the files that match in any of the directories:

```
$ dir xda_files:*2.dat

Directory XDA_ROOT: [DATA]

XDA_DATA.DAT;1    XDA_SUMMARY.DAT;3

Total of 2 files.

Directory XDA_ROOT: [USER]

USER-DATA2.DAT;1

Total of 1 file.
```

Search lists are useful when the files you want to access are spread across multiple directories whose organization is not obvious. Search lists are also useful when you want to access files in a public directory but supersede some of them with files in a personal directory. In this case, a search list can be defined whose first equivalence string is the personal directory and whose second equivalence string is the public directory. VMS will locate a file in the public directory unless a file with the same name exists in the personal directory.

When a search list logical name appears in a file spec being used to create a file, the file is always created in the *first* directory. It does not matter whether a file with the same name and type appears in some other directory in the search list.

14.6.4 Product Parameters

Because system logical names are available to everyone on a VMS system, they are sometimes used as a repository for product parameters. A **product parameter** is an item of information required by a software product in order to control its operation. A good example is the time zone required by VAX LISP to support its time manipulation functions. Because VMS does not provide time zone information, VAX LISP relies on the system manager to establish a logical name containing the local time zone. This logical name is called LISP\$TIME_ZONE, and its equivalence string is an integer or floating-point number specifying the local time zone relative to Greenwich Mean Time (e.g., 5 for Massachusetts, U.S.A.).

You should give careful consideration to the idea of using logical names for product parameters before you embark on such a scheme. There are two problems inherent in the technique:

- Someone has to ensure that the logical names are defined whenever the system is booted. The product installation procedure or the system manager must create a procedure containing the logical name definitions. The system manager must remember to invoke the procedure from the site-specific startup procedure.
- If the product has many parameters, the system logical name table will become cluttered with logical names. This takes up system memory and can be annoying when someone is displaying system logical names. When a product has more than a few parameters, it might be better to place them in a data file that the product reads when it starts up.

Obtaining Logical Name Information

Information about a logical name can be obtained with the F\$TRNLNM lexical function. The acronym TRNLNM stands for “translate logical name.” The function has the following format:

```
f$trnlnm(logical-name, table, index, mode, case, item)
```

The only required argument is *logical-name*. It is a string that specifies the logical name for which information is desired. When only the first argument is specified, VMS searches the process, job, group, and system logical name tables for a definition of the logical name. The equivalence string of the first definition is returned; iterative translation is not performed. If the logical name is not defined in any table, the null string is returned.

Here is an example:

```
$! The logical name parameter XDA_MANAGER specifies the user who
$! manages the XDA system.
$
$ xda_manager = f$trnlnm("xda_manager")
$ if xda_manager .eqs. "" then xda_manager = "SYSTEM"
```

The F\$TRNLNM function is used to translate the logical name XDA_MANAGER, a product parameter that specifies the user who manages XDA. If the logical name is not defined, the SYSTEM user is assumed.

The second argument, *table*, specifies the logical name table to search for the logical name. By default, the process, job, group, and system tables are searched, in that order. To restrict the search to a particular table, specify a *table* argument of "LNM\$PROCESS", "LNM\$JOB", "LNM\$GROUP", or "LNM\$SYSTEM". Section 14.10 describes logical name tables in more detail.

The third argument, *index*, specifies the index of the equivalence string to be returned. The default value is 0, which is the index of the first or only equivalence string. When the logical name is a search list, the additional equivalence strings are numbered 1, 2, and so on. The following code displays all the equivalence strings for the logical name XDA_FILES:

```

$   i = -1
$10: i = i + 1
$   string = f$trnlm("xda_files",,i)
$   if string .eqs. "" then goto 19
$   display "#", i, ": ", string
$   goto 10
$19:

```

The symbol *I* is used as the index argument to the `F$TRNLNM` function. Each equivalence string is displayed. The loop terminates when `F$TRNLNM` returns the null string, signifying that there are no more equivalence strings.

The fourth argument, *mode*, is used to select the access modes that are searched for the logical name. The argument is a keyword string, one of the following: "USER", "SUPERVISOR", "EXECUTIVE", or "KERNEL". The lexical function first looks for a logical name defined at the specified mode, then at the next most privileged mode, and so on. Logical names defined at less privileged modes are ignored. The default is user mode, which means that all logical names are searched and the value of the least privileged one is returned. Access modes are described in detail in Section 14.4.

The fifth argument, *case*, determines whether the logical name search is sensitive to the case of the letters in the logical name. By default, the search is case-blind, which means that the case of the *logical-name* argument can be ignored. VMS first searches for a logical name that matches the case of the first argument. If this fails, it searches for a logical name ignoring the case of the argument. When the search is case-sensitive, VMS searches only for a logical name whose case exactly matches the first argument. The *case* argument is specified as a keyword string, either "CASE_BLIND" or "CASE_SENSITIVE".

The sixth argument, *item*, specifies which item of information about the logical name is to be returned by the function. This argument is a keyword string; some choices are described in Table 14.3. The default item is VALUE, which returns the equivalence string.

Prior to VMS Version 4, logical names were translated with the `F$LOGICAL` function. The `F$LOGICAL` function is superseded by `F$TRNLNM` and should no longer be used.

Table 14.3 F\$TRNLNM Items

Keyword	Type	Description of Result
"ACCESS_MODE"	String	One of the following keywords to specify the access mode of the logical name: "USER", "SUPERVISOR", "EXECUTIVE", "KERNEL".
"CONCEALED"	Boolean	"TRUE" or "FALSE" to indicate whether the CONCEALED attribute was specified on the equivalence string.
"LENGTH"	Integer	The length of the equivalence string.
"MAX_INDEX"	Integer	The largest equivalence string index associated with the logical name. Note that this is 1 less than the number of equivalence strings, because the index is zero-based.
"TABLE"	Boolean	"TRUE" or "FALSE" to indicate whether the logical name is the name of a logical name table (see Section 14.10).
"TABLE_NAME"	String	The name of the table where the logical name was located.
"VALUE"	String	The equivalence string.

14.8

Deleting Logical Names

A logical name is removed from a logical name table with the DEASSIGN command. The command requires a single parameter, the logical name to be deleted:

```
$ deassign market_report_texts
```

The DEASSIGN command accepts the same logical name table qualifiers as the DEFINE command: /PROCESS, /JOB, /GROUP, /SYSTEM. The process table is the default.

The DEASSIGN command corresponds to the ASSIGN command, which is not used in this book. In order to create a command that corresponds more closely to the DEFINE command, the following personal command can be defined:

```
$ undefine = "deassign"
```

Then a logical name can be deleted as follows:

```
$ undefine market_report_texts
```

Overview of VMS Directory Organization

The VMS system directories are organized under a root directory on the system disk. The system disk is referred to with the logical name `SYS$SYSDEVICE`. The root directory is actually a search list defined by the logical name `SYS$SYSROOT`. Because all VMS files are located under the root directory, `SYS$SYSDEVICE` is rarely used, `SYS$SYSROOT` being the preferred method of accessing the files.

A system disk can contain multiple VMS systems, so each is given a number from 0 through 15. The root directory for system 0 is `SYS0`; the following explanation will concern itself with system 0. The logical name `SYS$SYSROOT` is a search list whose equivalence strings are the logical names `SYS$SPECIFIC` and `SYS$COMMON`. These logical names, in turn, refer to two root directories, one for the system-specific system directories and one for the common system directories shared by all systems. This double root scheme is quite powerful.

The majority of VMS files reside in subdirectories under the common root directory. The logical name `SYS$COMMON` refers to this directory, whose complete spec is `SYS$SYSDEVICE: [SYS0.SYSCOMMON.]`. Under the common root directory are subdirectories, such as `SYSEXE` and `SYSHLP` which contain the VMS files. When your VAX is part of a VAXcluster, it shares the same common root as the rest of the nodes in the cluster (there are exceptions to this rule). In this way, all the nodes in the cluster run the same version of VMS and share clusterwide data files, such as the user authorization file in the `SYSEXE` directory.

The logical name `SYS$SPECIFIC` refers to a second root directory whose complete spec is `SYS$SYSDEVICE: [SYS0.]`. Under the specific root directory is another complete set of VMS directories, also including `SYSEXE`, `SYSHLP`, and so forth. This is a separate collection of directories from the ones under the common root. In these directories reside files that are specific to the incarnation of VMS running on the local node. Because the search list `SYS$SYSROOT` specifies `SYS$SPECIFIC` before `SYS$COMMON`, a file in the specific directories will supersede one with the same name and type in the common directories. This allows the local node to have a private copy of a file normally found in the common directories and shared by all nodes. In a VAXcluster, each node has its own set of specific system directories. The specific directories contain only a few files, those which are specific to the node and which must supersede the common files.

As an example, assume a VAXcluster with two nodes, `HUMPTY` and `DUMPTY`. `HUMPTY` is assigned root 0 on the system disk, `DUMPTY` is assigned root 1. On both nodes, `SYS$SYSROOT` is a search list whose equivalence strings are the logical names `SYS$SPECIFIC` and `SYS$COMMON`. On `HUMPTY`, the `SYS$SPECIFIC`

Table 14.4 System Logical Names

Logical Name	Description
SYS\$COMMON	The root directory of the system directories common to all nodes in a VAXcluster.
SYS\$HELP	The directory of system help files.
SYS\$LIBRARY	The directory of system macro and text libraries.
SYS\$MANAGER	The directory of the system manager files.
SYS\$MESSAGE	The directory of the system message files.
SYS\$NODE	The DECnet node name of the local system.
SYS\$SPECIFIC	The root directory of the system directories specific to the local node.
SYS\$SYSDEVICE	The disk containing the system directories.
SYS\$SYSROOT	The search list used to access all files in the system directories. Its equivalence strings are SYS\$SPECIFIC and SYS\$COMMON.
SYS\$SYSTEM	The directory containing system programs and procedures.
SYS\$UPDATE	The directory containing system installation and update files.

logical name translates to SYS\$SYSDEVICE: [SYS0.], while on DUMPTY it translates to SYS\$SYSDEVICE: [SYS1.]. Under these two specific roots are separate sets of the system directories SYSEXE, SYSHLP, and so on. Because the system directories are separate, each node can have private copies of system files in them, which will supersede the common files because the specific directory is mentioned first in the search list. On HUMPTY, SYS\$COMMON translates to SYS\$SYSDEVICE: [SYS0.SYSCOMMON.], while on DUMPTY it translates to SYS\$SYSDEVICE: [SYS1.SYSCOMMON.]. These two directories act as the root for a third, *shared* set of system directories, which contains the majority of VMS files. Because both roots share the same subdirectories, both nodes are accessing the same copy of system files.

Assume that HUMPTY attempts to access the user authorization file SYSUAF.DAT, which resides in the SYSEXE directory. If it has its own copy of the file in the specific directory SYS\$SYSDEVICE: [SYS0.SYSEXE], that copy is accessed. If it does not, the shared copy in SYS\$SYSDEVICE: [SYS0.SYSCOMMON.SYSEXE] is accessed. The shared copy is available to the entire cluster.

This specific/common directory scheme is used on every VMS Version 5 system regardless of whether it is a cluster. Each of the system directories SYSEXE,

SYSHLP, and so on has its own logical name defined so that files in these directories can be accessed without having to understand the whole scheme. Table 14.4 describes many of the system logical names.

14.10 **Overview of Logical Name Tables**

Every logical name resides in a logical name table. Each logical name table is assigned a name so that it can be specified in various commands and system requests. For example, a logical name can be looked up in a table by specifying the name of the table and the name of the logical name. There are four standard logical name tables available to every process:

Process. The process logical name table is called LNM\$PROCESS_TABLE.

Job. The job logical name table is called LNM\$JOB_*jjj*, where *jjj* is a unique number for the job.

Group. The group logical name table is called LNM\$GROUP_*ggg*, where *ggg* is the group number.

System. The system logical name table, that which is shared by all processes, is called LNM\$SYSTEM_TABLE.

In order to determine the value of the system logical name SYS\$HELP, you must ask the system to “look up the logical name SYS\$HELP in the logical name table LNM\$SYSTEM_TABLE.” This is referred to as translating the logical name. In a command procedure, logical name translation is performed with the F\$TRNLNM lexical function.

To eliminate the requirement that you know the obscure names of various logical name tables such as LNM\$JOB_8021BC60, four logical names are defined to refer to the four standard logical name tables. Note the recursive nature of this scheme: logical names refer to logical name tables, which contain logical names. The logical names for the standard logical name tables are as follows:

Process. The logical name LNM\$PROCESS refers to the process logical name table.

Job. The logical name LNM\$JOB refers to the job logical name table.

Group. The logical name LNM\$GROUP refers to the group logical name table.

System. The logical name LNM\$SYSTEM refers to the system logical name table.

This scheme, although it is already rather complex, has two shortcomings:

- There is no place for the logical names that refer to logical name tables to reside.
- In order to look up a logical name, the user has to know in which logical name table it resides.

Both of these shortcomings are eliminated with the introduction of **logical name table directories**. A logical name table directory, or simply directory, contains a list of logical name tables and logical names that refer to logical name tables. Instead of requesting VMS to “look up this logical name in this table,” you can now request it to “look up this logical name in certain tables listed in this logical name table directory.” This request is inherently easier to formulate, because there are far fewer directories than there are logical name tables.

In fact, there are two logical name table directories. The first is associated with a process and contains the name of the process logical name table. It also contains logical names that refer to the process, job, and group logical name tables. In addition, it contains the name of the process logical name table directory itself: `LNM$PROCESS_DIRECTORY`.

The second directory is the system logical name table directory. There is one such directory for the entire system, shared by all users. The system directory contains the name of all the job logical name tables, group logical name tables, and the system logical name table. It contains various logical names that refer to collections of logical name tables. And finally, it contains the name of itself: `LNM$SYSTEM_DIRECTORY`.

Three of the logical names in the system directory are of particular interest:

`LNM$FILE_DEV`. This logical name is defined as a search list of logical name tables that are to be used when parsing a file spec. The default value of this logical name is the list `LNM$PROCESS`, `LNM$JOB`, `LNM$GROUP`, `LNM$SYSTEM`. It is this default list that causes the file spec parsing service to search for logical names first in the process table, then in the job, group, and system tables.

`LNM$DCL_LOGICAL`. This logical name is used by the `SHOW LOGICAL` command and `F$TRNLNM` lexical function to determine which logical name tables to search for the logical name being translated. By default, `LNM$DCL_LOGICAL` refers in turn to the logical name `LNM$FILE_DEV`.

`LNM$DIRECTORIES`. This logical name is defined as a search list of the two logical name table directories: first `LNM$PROCESS_DIRECTORY` and then second `LNM$SYSTEM_DIRECTORY`.

The logical name table scheme presented here makes logical names relatively easy for the average user to deal with. But never let it be said that the scheme is simple. The following is a list of the contents of the two logical name table directories on a small VMS system:

(LNM\$PROCESS_DIRECTORY)

```
"LNM$GROUP" = "LNM$GROUP_000260"  
"LNM$JOB" = "LNM$JOB_802E9DD0"  
"LNM$PROCESS" = "LNM$PROCESS_TABLE"  
"LNM$PROCESS_DIRECTORY" [table] = ""  
"LNM$PROCESS_TABLE" [table] = ""
```

(LNM\$SYSTEM_DIRECTORY)

```
"LMF$LICENSE_TABLE" [table] = ""  
"LNM$DCL_LOGICAL" = "LNM$FILE_DEV"  
"LNM$DIRECTORIES" = "LNM$PROCESS_DIRECTORY"  
    = "LNM$SYSTEM_DIRECTORY"  
"LNM$DT_FORMAT_TABLE" [table] = ""  
"LNM$FILE_DEV" [super] = "LNM$PROCESS"  
    = "LNM$JOB"  
    = "LNM$GROUP"  
    = "LNM$SYSTEM"  
"LNM$FILE_DEV" [exec] = "LNM$SYSTEM"  
"LNM$GROUP_000001" [table] = ""  
"LNM$GROUP_000010" [table] = ""  
"LNM$GROUP_000201" [table] = ""  
"LNM$GROUP_000260" [table] = ""  
"LNM$JOB_802E8E00" [table] = ""  
"LNM$JOB_802E94E0" [table] = ""  
"LNM$JOB_802E9DD0" [table] = ""  
"LNM$JOB_802E9E80" [table] = ""  
"LNM$JOB_802E9F30" [table] = ""  
"LNM$JOB_802EACF0" [table] = ""  
"LNM$JOB_802EBAB0" [table] = ""  
"LNM$JOB_802FOC80" [table] = ""  
"LNM$JOB_802F3880" [table] = ""  
"LNM$JOB_802F3EB0" [table] = ""  
"LNM$JOB_802F4380" [table] = ""
```

```
"LNM$PERMANENT_MAILBOX" = "LNM$SYSTEM"  
"LNM$STARTUP_TABLE" [table] = ""  
"LNM$SYSTEM" = "LNM$SYSTEM_TABLE"  
"LNM$SYSTEM_DIRECTORY" [table] = ""  
"LNM$SYSTEM_TABLE" [table] = ""  
"LNM$TEMPORARY_MAILBOX" = "LNM$JOB"  
"LOG$GROUP" = "LNM$GROUP"  
"LOG$PROCESS" = "LNM$PROCESS"  
    = "LNM$JOB"  
"LOG$SYSTEM" = "LNM$SYSTEM"  
"TRNLOG$_GROUP_SYSTEM" = "LOG$GROUP"  
    = "LOG$SYSTEM"  
"TRNLOG$_PROCESS_GROUP" = "LOG$PROCESS"  
    = "LOG$GROUP"  
"TRNLOG$_PROCESS_GROUP_SYSTEM" = "LOG$PROCESS"  
    = "LOG$GROUP"  
    = "LOG$SYSTEM"  
"TRNLOG$_PROCESS_SYSTEM" = "LOG$PROCESS"  
    = "LOG$SYSTEM"
```

Chapter 15



Sequential File Operations

Chapters 13 and 14 described VMS files and presented methods for manipulating files as a whole. This chapter begins the presentation of DCL commands that operate on the contents of files: commands for reading, writing, and updating files. In particular, this chapter addresses sequential files, which are collections of data that can only be accessed in order from beginning to end. Chapter 17 addresses indexed files.

When a program performs file operations such as reading and writing, the operations are carried out by the VMS **Record Management System**, or RMS. This chapter and later chapters will refer to RMS when discussing file operations.

A **sequential file** is a file made up of a sequence of individual **records**. Each record contains some logically related information, while all the records taken together contain the sum total of information in the file. One of the distinguishing features of sequential files is that the records are arranged in a particular sequence and can only be accessed in that order. The file is written one record at a time, from beginning to end, and afterwards can only be read in the same order.

The records in a sequential file can be composed of free-format text, or they can be laid out in a predefined format called a **record structure**. This chapter assumes that sequential files contain free-format text; record structures are described in the next two chapters. In a sequential free-format text file, or simply **text file**, each record can contain a different amount of text; the length of all

records need not be the same. Such records are called **variable-length** records. A text file can also have **fixed-length** records, but those files are not as common.

A program source file is one example of a text file. In particular, a DCL procedure is contained in a text file. Text files can be created and modified using a text editor, such as TPU or Emacs.

15.1 **Reading an Existing File**

Before the records in an existing file can be read, the file must be **opened**. The process of opening a file prepares it for future input operations. In DCL, files are opened with the `OPEN` command, which ensures that the file exists and determines whether the process is allowed to read the file. The `OPEN` command requires two parameters, a logical name and the specification of the file to be opened:

```
$ open/read xda_data_file sys$manager:user-list.dat;
```

The `/READ` qualifier specifies that the file is to be opened for read access. Records can be read from the file, but the file cannot be modified. The first parameter is a logical name (not a symbol), which is assigned by the `OPEN` command to refer to the opened file. The logical name is used in subsequent commands that need to access the open file (e.g., the `READ` command to obtain records from the file). The second parameter is a file spec that identifies the particular file to be opened.

▷ Ch. 20

The file is opened as a process-permanent file, which means that it remains open until it is explicitly closed by a subsequent `CLOSE` command or until the user logs out. The logical name is not equated to the full file spec of the open file. Instead it is equated to a binary-coded string containing an internal reference to the file. If, after the file is opened, you use the logical name in other DCL commands, the command will operate on the open file. In particular, the `READ` command will read records from the file.

If the file does not exist or cannot be opened for some other reason, an error status is returned by the `OPEN` command and the procedure's error handler is invoked. This fatalistic behavior may not be acceptable in a sophisticated procedure, particularly if the file spec was entered by the user in response to a query. The `/ERROR` qualifier can be used on the `OPEN` command to specify a label at which execution should proceed if an error occurs. The following code continually prompts the user for a file spec until the file can be opened:

```

$10:  libcall ask xda_file s "Enter a file spec:"
$      open/read xda_data_file 'xda_file /error=15
$      goto 19
$15:  libcall signal xda w badfile -
      "File ''xda_file cannot be opened." '$status
$      goto 10
$19:

```

If any error occurs while opening the file, DCL continues execution at label 15 rather than with the command following the OPEN. A warning is signaled and the user is prompted for another file spec. The warning message includes a second line containing the RMS status message.

Once a file has been opened successfully, records can be retrieved from it with the READ command. Each READ command obtains the next sequential record from the file and stores its contents in a symbol. The command requires two parameters: the logical name defined by the open operation, and the symbol in which the retrieved record is stored:

```
$ read xda_data_file line
```

The next record is read from the file specified by logical name XDA_DATA_FILE and then the record is stored in the symbol LINE. In a fashion identical to the OPEN command, any errors are returned by the READ command and cause the procedure's error handler to be invoked. The /ERROR qualifier can be used to override this behavior; it specifies a label at which execution should continue if an error occurs:

```
$ read xda_data_file line /error=20
```

If the entire file has been read and no more records are available, the **end-of-file** condition occurs. DCL treats this condition as an error, so an error status is returned from the READ command or, if the /ERROR qualifier is present, execution continues at the error label. Because end-of-file is a common condition that is often not treated as an error, a separate qualifier is provided to name a handler for it. The /END_OF_FILE qualifier specifies a label to which DCL branches when end-of-file is detected:

```
$ read xda_data_file line /error=20/end_of_file=30
```

If end-of-file occurs on this read operation, execution continues at label 30. If any other error occurs, execution continues at label 20.

Once all the necessary records have been read from a file, it must be closed. The close operation relinquishes access to the file and deassigns the logical name defined by the open operation:

```
$ close xda_data_file
```

The following loop reads and displays the file specified by symbol FILE_SPEC:

```
$ open/read xda_data_file 'file_spec /error=5
$ goto 10
$5: libcall signal xda w openerr -
      "Unable to open file ''file_spec''" '$status
$ goto 20
$10: read xda_data_file line /end_of_file=19
$ display line
$ goto 10
$19: close xda_data_file
$20:
```

▷ Ch. 20

It is crucial that an application close every file it opens. If a file is not closed explicitly by the application, it will not be closed automatically when the main procedure exits, because files opened by DCL are process-permanent files. The file remains open until the process terminates; in particular, it is still open if the application is run again. You might think that the second OPEN command would “reopen” the file by first closing and then opening it, but in fact the second OPEN command does absolutely nothing: it does not close the file, it does not open it a second time, nor does it return any kind of error status. Because a file is opened as a process-permanent file, DCL assumes that any subsequent OPEN commands using the same logical name need to access the file that is already opened. Therefore, the file remains in the state it was when the application terminated the first time, perhaps partly read or at end-of-file.

To ensure that all files are closed, a procedure that opens a file should close it in two places. First, the procedure should close the file as soon as it has no further need for the file. This means that a file is closed as soon as possible so that unused open files don't take up system resources. Second, the procedure should include cleanup code to close the file. This guarantees that the file is closed regardless of whether the procedure terminates normally or due to an error or interrupt. The CLOSE command in the cleanup code should include the /NOLOG qualifier, which suppresses any error messages if the file was already closed by the procedure in its normal course of execution. Here is the preceding example with cleanup code added:

```

$   open/read xda_data_file 'file_spec /error=5
$   goto 10
$5: libcall signal xda w openerr -
      "Unable to open file ''file_spec''" '$status
$   goto 20
$10: read xda_data_file line /end_of_file=19
$   display line
$   goto 10
$19: close xda_data_file
$20:
      :
      :
$exit:
$   set noon
$   close/nolog xda_data_file ! Close data file if left open.
$   exit status .or. %x10000000

```

15.2 **Creating and Writing a New File**

A DCL application can create a new sequential file and write records to it. The create operation is distinct from the write operation. There are many ways to create a new file, but the simplest way is to let the OPEN command do it:

```
$   open/write xda_new_file xda_system:xda_user-data.dat;/error=20
```

When the /WRITE qualifier is specified, the OPEN command either creates the specified file or returns an error. The rules are simple:

- If the specified file does not exist, it is created.
- If the specified file does exist:
 - If an explicit version is not included in the file spec (as in the preceding example), then a new version of the file is created with a version number higher than all existing versions.
 - If an explicit version is included and that version of the file does not exist, it is created.
 - If an explicit version is included and that version already exists, an error status is returned. The OPEN/WRITE command cannot be used to overwrite an existing file or to add records to it.

Once the file is created, the OPEN/WRITE command behaves exactly like the OPEN/READ command. The logical name is defined to refer to the file, so that subsequent WRITE and CLOSE commands can use it.

The WRITE command has been covered in previous chapters as it pertains to displaying information at the terminal. The same command is used to write records to a sequential file:

```
$ write xda_new_file line
```

The first parameter is the logical name defined by the OPEN command. The second parameter is an expression or list of expressions whose values are to be written together as one record. In the preceding example, the symbol LINE contains the record to be written. A write operation always creates exactly one record in the file.

The following example creates a new data file containing a list of the files in the system manager's directory. The list includes the modification dates of the files.

```
$ open/write xda_new_file sys$manager:file-list.dat
$10: file = f$search("sys$manager:*.*.");
$ if file .eqs. "" then goto 19
$ write xda_new_file file, " ", f$file_attributes(file,"RDT")
$ goto 10
$19:
$ close xda_new_file
```

Notice that the argument to the F\$SEARCH function includes an explicit semicolon so that only the latest version of each file is located. This prevents the list from containing multiple versions of the same file.

The WRITE commands illustrated above are limited to writing records of 1,024 bytes or less. Furthermore, the value of each individual expression is limited to 255 bytes. These limitations can be relaxed by including the /SYMBOL qualifier on the command, which allows expressions and records of up to 2,048 bytes. However, use of the /SYMBOL qualifier places another restriction on the WRITE command: the expressions specified in the command must be symbols; no string literals, lexical functions, or operators are allowed. When you want to write more than 1,024 bytes, you must build the output string in a symbol and then specify the symbol on the WRITE command.

The WRITE command exhibits one quirk when it creates a new file. Programmers familiar with VMS would expect it to create a standard sequential file with the **carriage return carriage-control** attribute, a file normally referred to as a **text file**. Such a file can be displayed, printed, edited with all available editors, and otherwise processed by any VMS utility or application. In fact, however, the command creates a sequential file containing records in the **variable with fixed**

control (VFC) format. Each record has a 16-bit header whose contents are not available to the DCL programmer.

Some editors and utilities cannot deal with VFC files. If you encounter a problem with VFC files, use the **CREATE** command to create the file, rather than allowing the **WRITE** command to do so. The **CREATE** command creates a standard text file and then reads lines from **SYS\$INPUT** to populate the file. During execution of a procedure, **SYS\$INPUT** is directed to the procedure file, so data lines in the procedure will end up as records in the file. If there are no data lines, the file will be empty:

```
$ create sys$manager:file-list.dat
$! There are no data lines, so the file will be empty.
```

Once the empty file is created, you can add records by opening the file and appending new records.

15.3 **Appending to an Existing File**

Additional records can be added to an existing sequential file by opening the file **for append**. Once a file is opened for append, write operations cause records to be added to the end of the file:

```
$ open/append xda_data_file sys$manager:file-list.dat;
```

In all other respects, a file opened for append behaves identically to a file opened for write.

The **/APPEND** qualifier is particularly useful when writing to an empty file made with the **CREATE** command. The file is a standard text file, not a VFC file, and thus DCL will write standard text records into it. Here is another example of creating a file list using the **CREATE** command:

```
$ create sys$manager:file-list.dat
$ open/append xda_new_file sys$manager:file-list.dat
$10: file = f$search("sys$manager:*.*.");
$ if file .eqs. "" then goto 19
$ write xda_new_file file, " ", f$file_attributes(file,"RDT")
$ goto 10
$19:
$ close xda_new_file
```

A DCL application often needs a **temporary file** to contain data for a short period of time. A temporary file is created by the procedure, filled with records, processed, and then deleted. The file does not exist after the application terminates. There are two important differences between temporary and permanent files.

The first difference concerns the location and name chosen for the temporary file. A temporary file should not be placed in the working directory or the directories dedicated to the VMS system and other software products. Temporary files will clutter those directories and confuse anyone looking in them. Don't forget that many users may be running your application or other similar applications. Temporary files should be placed in a directory dedicated to containing short-lived data. VMS provides a standard logical name, SYS\$SCRATCH, which can be defined to refer to such a directory. When you log in, SYS\$SCRATCH refers to your login directory, but you can certainly redefine it to a personal or systemwide directory created expressly for the purpose of containing temporary files.

The file name of a temporary file must be carefully chosen so that it does not duplicate the name of another temporary file. If two DCL applications create the file TEMP.DAT in the same directory, the results will surely be unpredictable. A unique name must be generated with a zero or diminishingly small probability of duplication. The process of generating a unique name is performed often enough so that it deserves a subroutine in the subroutine library. The subroutine is called UNIQUE_NAME and requires two parameters: a global symbol to be set to the generated name, and a pattern specifying how the name is formatted. The pattern can contain arbitrary text, but must include a question mark (?) to specify the position in which the unique portion of the name is inserted. The unique portion of the name is a ten-digit number. The subroutine is used in the following example:

▷ Apx. C

```
$ libcall unique_name xda_temp sys$scratch:xda_?.dat
```

The global symbol XDA_TEMP will be set to the unique name, which consists of the prefix SYS\$SCRATCH:XDA_ plus the unique portion plus the suffix .DAT. The unique portion consists of ten decimal digits, which are generated from the current time. The preceding subroutine call might return the following file spec:

```
SYS$SCRATCH:XDA_1643296105.DAT
```

The second difference between temporary and permanent files is that temporary files must be deleted before the application terminates. It is best to delete them as soon as they are no longer needed. In order to ensure that a temporary file is deleted, the DELETE command should be placed in the cleanup code of the procedure that created it. The following code creates a temporary file containing the names of the data files in the system manager's directory and mails it to the system manager:

```

$ libcall unique_name xda_data_list sys$scratch:xda_?.lis
$ directory/output='xda_data_list sys$manager:*.dat;
$ mail 'xda_data_list system -
    /subject="List of data files in SYS$MANAGER"
    .
    :continue processing
    .
$exit:
$ set noon
$ if f$type(xda_data_list) .nes. "" then -
    if f$search(xda_data_list) .nes. "" then -
        delete 'xda_data_list;*
    .
    :more cleanup
    .

```

A temporary file spec is generated and used in the /OUTPUT qualifier of the DIRECTORY command. The list of data files is therefore placed in the temporary file. The MAIL command sends this file to the system manager with an appropriate subject. The procedure's cleanup code includes three lines that delete the temporary file. These lines perform the following functions:

1. The first line ensures that the global symbol XDA_DATA_LIST exists. It is possible that an interrupt or error occurred before the symbol was created by the UNIQUE_NAME subroutine and therefore there is no file to delete.
2. The second line ensures that the temporary file does indeed exist. It is possible that an interrupt occurred between the call to UNIQUE_NAME and the completion of the DIRECTORY command.
3. The third line deletes the temporary file. Note that it deletes all versions of the file, just in case more than one was created.

The following example illustrates the technique for handling temporary files created with the OPEN command:

```

$ libcall unique_name xda_data_file sys$scratch:xda_?.tmp
$ open/write xda_data_file 'xda_data_file
.
. process file
.
$ close xda_data_file
.
.
$exit:
$ set noon
$ close/nolog xda_data_file
$ if f$type(xda_data_file) .nes. "" then -
    if f$search(xda_data_file) .nes. "" then -
        delete 'xda_data_file;*
.
. more cleanup
.

```

A unique file spec is generated and used in the OPEN command to create a temporary file and prepare it for writing. There is no reason to believe that the OPEN command will fail, so no /ERROR qualifier is specified; an error will cause the usual error handler to be invoked. Once open, the file is processed and eventually closed. The cleanup code must perform two operations on the temporary file. It must close the file if still open, and it must delete the file.

The SYS\$SCRATCH directory exists as a standard place in which to put scratch files. VMS does *not* automatically delete files in this directory.

15.5 **Displaying a File**

There are two ways in which a procedure can display a sequential file at the terminal. If the records are to be displayed exactly as they appear in the file, and if the file is to be displayed in its entirety, then the TYPE command can be used. The TYPE command writes the file's records to SYS\$OUTPUT, which normally refers to the terminal screen. The records are displayed without pausing, unless the /PAGE qualifier is included, in which case the records are displayed one screenful at a time. The following example asks the user for a file spec and displays the file screen by screen:

```

$ libcall ask xda_file s "File to display:"
$ type/page 'xda_file

```

▷ Ch. 9 It is courteous to allow the user to cancel the file display with <CTRL/y>. Only two lines must be added to this example:

```

$ libcall ask xda_file s "File to display:"
$ on control_y then goto 19
$ type/page 'xda_file
$19: on control_y then goto control_y

```

Another way to display sequential files is to read them one record at a time and display each record individually. This method allows much finer control over which records are displayed, how many are displayed, and the format of the display. The disadvantage is that the method is significantly slower than using TYPE. The following code displays the first five nonblank lines in a file:

```

$10: libcall ask xda_file s "File to summarize:"
$ open/read xda_file 'xda_file /error=20
$ goto 19
$20: libcall signal xda w filnotfnd -
      "File ''xda_file does not exist."
$ goto 10
$19:
$ n = 0
$20: read xda_file line /end_of_file=29
$ if f$edit(line,"TRIM") .eqs. "" then goto 20
$ n = n + 1
$ display line
$ if n .lt. 5 then goto 20
$29:
$ close xda_file

```

The F\$EDIT function is used to trim the trailing spaces from the line. If this results in the null string, then the line is blank and another line is immediately read. The loop terminates after the fifth nonblank line is displayed.

15.6 **Searching Files**

Certain DCL applications, particularly those in the domain of code management, system building, or system testing, require that text files be searched for occurrences of specific character sequences. For example, a procedure may need to search a set of files generated by a testing procedure for occurrences of the sequence "ERROR". VMS provides the SEARCH utility to assist with searching files. You could read and scan the lines of a text file directly in DCL, but such a procedure would be unacceptably slow for all but the smallest files.

Table 15.1 How /MATCH Affects Searching

Qualifier	Default?	Method of Matching
/MATCH=OR	✓	A text line matches if it contains <i>any</i> of the pattern strings.
/MATCH=AND		A text line matches if it contains <i>all</i> of the pattern strings.
/MATCH=NOR		A text lines matches if it contains <i>none</i> of the pattern strings.
/MATCH=NAND		A text lines matches if it contains <i>some</i> but not <i>all</i> of the pattern strings.

The SEARCH utility requires two parameters: the specification of the files to be searched, and an indication of what to search for (known as the search **pattern**). The utility can search a single file or any number of files that match a wildcard spec. The output from the utility consists of a listing of those lines in the files that match the search pattern. The output is normally directed at the terminal but can be redirected, as usual, with the /OUTPUT qualifier.

The search pattern is specified as one or more character strings in conjunction with the /EXACT and /MATCH qualifiers. The /EXACT qualifier determines whether the pattern strings must match the text exactly (/EXACT) or may match without regard to uppercase and lowercase letters (/NOEXACT). The default is /NOEXACT. Table 15.1 describes how the /MATCH qualifier determines the method of searching when used in conjunction with the pattern strings.

The SEARCH command accepts various other qualifiers, which control the amount and format of its output. One important feature of the command is the status code it returns. If any matches are found, the status code includes a success severity. If no matches are found, the status code includes an informational severity. Neither status is an error, but the two cases can be distinguished by the severity of the status.

The following example repeatedly asks the user for a file spec and tells the user whether the file contains the sequence "ERROR" or "WARNING":

```

$10: libcall ask xda_file s "File to be searched:"
$     if f$search(xda_file) .nes. ""
$     then
$         search/output=nl: 'xda_file "error","warning"/match=or
$         match = $severity .eq. 1
$         if match then display "The file contains errors."
$         if .not. match then display "The file contains no errors."
$     else
$         libcall signal xda w filnotfnd -
$                                     "File ''xda_file does not exist."
$     endif
$     goto 10

```

The /OUTPUT qualifier is used to discard all the output from the SEARCH command by directing it to the null device. Therefore, the only interesting result of the command is the status code it returns. If a success status is returned, then the file contains the pertinent sequences. If an informational status is returned, the file does not contain the sequences.

Chapter 16



Data Manipulation

This chapter presents various techniques and tricks for manipulating data in DCL. The techniques include methods for performing simple arithmetic operations, character string manipulation, and calculations related to dates and times.

16.1

Arithmetic Techniques

DCL provides only rudimentary arithmetic operations on integer values. Applications often require more sophisticated operations, which must be built up out of the operations that DCL provides. This section illustrates a few of the advanced operations and how they can be achieved in DCL.

The “maximum” function examines two or more integers and returns the largest one. The “minimum” function is similar but returns the smallest integer. These functions are not provided by DCL but can be implemented as follows:

```
$! Determine the largest of the values A, B, and C.
$
$ max = a
$ if b .gt. max then max = b
$ if c .gt. max then max = c
```

-or-

```
$! The block count is equal to the total file blocks, but can be
$! no larger than 600, i.e., minimum(total-file-blocks, 600).
$
$ block_count = total_file_blocks
$ if block_count .gt. 600 then block_count = 600
```

Sometimes it is necessary to round a number to a multiple of a particular integer. For example, when calculating the number of bytes a file will occupy, it makes sense to round up to a multiple of 512, since disk space is always allocated in units of 512-byte blocks. The following code rounds up the value in BYTES to a multiple of 512:

```
$ bytes = (bytes + 511) / 512 * 512
```

Because division truncates the quotient towards zero, the result of the division is always an integer. The multiplication is therefore guaranteed to produce an integer multiple of 512. By first adding 511, the division operation will round up rather than down. In general, if the number b is to be rounded up to a multiple of m , the command is

```
$ b = (b + m-1) / m * m
```

A variation on this expression can be used to calculate the remainder upon dividing one number by another. The following command determines the remainder when dividing A by B:

```
$ remainder = a - a/b*b
```

Note that the remainder has the same sign as the dividend.

DCL does not provide floating-point numbers, and it is rather tedious to simulate them with integers. However, there is one floating-point calculation that is often useful: determining the ratio of two numbers. This can be accomplished by multiplying the numerator by 100 and then performing the division:

```
$ ratio = blocks_used*100 / total_blocks
```

This command calculates the ratio of disk blocks used to total disk blocks. The ratio is an integer in the range 0–100, so it can also be displayed as a percentage. Note that the percentage is rounded down to the nearest integer.

For the purposes of this section, a **list** is a sequence of data items separated by commas or some other delimiting character. A character string can contain a list of items like the following:

```
"CYNTHIA ,PAT ,DIRK ,DAVID ,RACHEL ,BILL ,MEREDITH"
```

– or –

```
"TXT ,DAT ,LIS ,EXE ,OBJ ,CLD"
```

The first example is a list of people's first names, while the second is a list of file types. The first list might represent the names of friends to be invited to a party. The second list might represent the types of files found in a particular directory.

The following code builds a list of all the file types found in the directory whose file spec is in the symbol DIRECTORY:

```
$ type_list = ""
$10: file = f$search(directory+"*.*;")
$ if file .eqs. "" then goto 19
$ type = f$parse(file,,,"TYPE") - "."
$ type_list = type_list + "," + type
$ goto 10
$19:
$ type_list = f$extract(1, 9999, type_list)
```

The list is constructed in the symbol named TYPE_LIST, which is initialized to the null string before the loop. File types are added to the list by concatenating each type to the character string in the TYPE_LIST symbol, along with a comma to separate it from the previous type. When the loop terminates, TYPE_LIST contains a list of all the file types found in the directory, each separated by a comma. There will also be a comma at the beginning of the list, which is removed using the F\$EXTRACT function. The function receives a starting index of 1, thus excluding the first character, and a length of 9999 so that all remaining characters are included.

The list constructed by the previous example will contain duplicates when there is more than one file with the same type. Eliminating duplicates requires a little more code in the loop:

```

$   type_list = ""
$10:  file = f$search(directory+"*.*;")
$     if file .eqs. "" then goto 19
$     type = f$parse(file,,,"TYPE") - "."
$     if f$locate("."+type+",", type_list+",") .eq. -
        f$length(type_list)+1 then -
$       type_list = type_list + "," + type
$     goto 10
$     type_list = f$extract(1, 9999, type_list)
$19:

```

In order to determine whether a type is already in the list, the F\$LOCATE function is used to scan the list for a duplicate type. The F\$LOCATE function takes two arguments: a pattern string and a target string. The target string is scanned for an occurrence of the pattern string. If the pattern is found, its index in the target string is returned. If the pattern is not found, the length of the target is returned. Upon first consideration you might think it would suffice to use the file type as the pattern and the list as the target, thus scanning for the type in the list. However, this would fail in the case of two types where one was a substring of the other.

Take the file types DAT and STAR-DATA, for example. If the type list contained STAR-DATA and a file with type DAT was encountered, it would appear that the DAT type was already in the list because F\$LOCATE would find the "DAT" in "STAR-DATA". In order to prevent this problem, you must search for the file type surrounded by commas. Requiring the matches to include a comma on each end of the file type ensures that only complete file types are matched.

Once a list is constructed, you may want to split it apart into its individual components. The F\$ELEMENT function is perfect for this task, as illustrated by the following code, which displays the file types in the list, one per line:

```

$   i = -1
$20:  i = i + 1
$     type = f$element(i, ",", type_list)
$     if type .eqs. "," then goto 29
$     display type
$     goto 20
$29:

```

Imagine the type list as a sequence of types, each with a numerical index. The leftmost one is number 0, the next 1, and so on. The F\$ELEMENT function takes three arguments: the first is the index of the desired list element; the second, the delimiter character; and the third, the list itself. The function extracts the element specified by the index from the list and returns it. If the index specifies

an element that does not exist, the function returns the delimiter (it does not return a null string, because a null string is a perfectly valid list element).

The code uses `I` as the element index, initializing it to `-1` before the loop. Each iteration of the loop increments the index and extracts the next type from the list. If `F$ELEMENT` returns a comma, then the list is exhausted and the loop terminates. If the function does not return a comma, then it returns a file type, which is displayed. The following table shows the values returned by each call to `F$ELEMENT` when the type list is `"TXT,LSP,EXE,STAR-DATA,CLD,DAT"`:

Value of I	Return Value
0	"TXT"
1	"LSP"
2	"EXE"
3	"STAR-DATA"
4	"CLD"
5	"DAT"
6	", "

Do not be tempted to use `F$ELEMENT` to analyze or split apart such things as file specs. Assume a file spec is stored in the symbol `FILE_SPEC`. It is incorrect to extract the file type and version from the file spec with the following command:

```
$! Extract everything after the dot preceding the file type.  
$  
$ type_version = f$element(1, ".", file_spec)
```

The directory portion of the file spec might contain a dot, and the command would fail to perform as expected. Even something so foolproof as extracting the file name/type/version can backfire:

```
$! Extract everything after the directory's closing bracket.  
$  
$ name_type_version = f$element(1, "]", file_spec)
```

Remember, angle brackets (`<` `>`) are also valid directory delimiters. If they were present, the command would fail. File specs must be split up using the `F$PARSE` lexical function.

▷ Ch. 13

A **keyword** is a character string chosen from among a fixed set of valid strings. Keywords are used to represent commands or options when the repertoire of such choices is finite. A keyword is more meaningful than a number (imagine if DCL commands were chosen by number rather than by verb!). Keywords are used in lexical functions to control the behavior of the function (e.g., the F\$EDIT function requires keywords to control the kind of string editing performed). Keywords can also be used by DCL applications when asking the user to choose among several possible actions. For example, an application that creates many files in a directory might incorporate a feature to clean up the directory from time to time. The cleanup function can be controlled by a query such as

```
$ libcall ask xda_clean_option s -
    "Directory cleanup option (DELETE, PURGE, NONE):"
```

▷ Apx. C

The user must enter DELETE, PURGE, or NONE to choose the desired action. It is usually helpful if the application allows the keyword to be abbreviated. And a message should be displayed if the user enters an invalid option. All this “keyword manipulation” is embodied in the library subroutine LOOKUP_KEYWORD. This subroutine requires three arguments: a global symbol, which is set to the result of the lookup; the user’s original input; and a list of valid keywords in alphabetical order. If the user’s input is a full or abbreviated keyword in the list, the global symbol is set to the full keyword. If the user’s input is not a valid keyword or is an ambiguous abbreviation of a keyword, the global symbol is set to the null string.

Here is the cleanup example expanded to include keyword checking:

```

$10:  libcall ask xda_clean_option s -
      "Directory cleanup option (DELETE, PURGE, NONE):"
$     libcall lookup_keyword xda_clean_option -
      "'xda_clean_option'" DELETE,NONE,PURGE
$     if xda_clean_option .eqs. "" then goto 10
$
$     goto 20_'xda_clean_option
$
$20_DELETE:
      . delete files
      .
$     goto 29
$20_NONE:
$     goto 29
$20_PURGE:
      . purge files
      .
$29:

```

Once a valid option is obtained from the user, the keyword is used in a GOTO command to select the appropriate cleanup code. In the case of the NONE option, the cleanup code does nothing.

A GOTO command that employs apostrophe substitution to select one of a number of labels is called a **case statement**. The value of a symbol is substituted in the GOTO command to provide part or all of the label name. DCL then alters the procedure's flow of control to continue at the selected label. The procedure must be careful to ensure that the label that results after the substitution is present in the procedure. If not, an error is signaled. When keywords are used to form the labels, as in the previous example, the use of LOOKUP_KEYWORD guarantees that the keyword is valid and therefore that the GOTO will work as expected.

16.4

Dates and Times

Dates and times are important data items in many applications. VMS uses the terms **time** or **date** interchangeably to refer to dates, times, or combinations of the two. DCL provides facilities to manipulate times, which are represented as character strings in the following format:

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

In this format, *dd* is the day of the month, *mon* is a three-letter abbreviation for the month, *yyyy* is the year, *hh* is the hour in 24-hour form, *mm* is the minute, *ss* is the second, and *cc* is the hundredth of a second. This time is called an **absolute time** because it denotes a specific time, past, present, or future. There are also three special absolute times that are relative to the current time. Note that these are absolute times and therefore represent specific points in time, not time spans:

TODAY. The current day at 00:00:00 (midnight).

TOMORROW. The next day at midnight (24 hours after TODAY).

YESTERDAY. The previous day at midnight (24 hours before TODAY).

The F\$TIME lexical function returns the current absolute time. It requires no arguments and returns the current time as a character string:

```
$ display "The current time is: ", f$time()
```

This command could produce the following display:

```
The current time is 11-NOV-1988 08:54:36:29
```

Certain other functions can return times when given the appropriate arguments. For example, F\$FILE_ATTRIBUTES returns the creation time of a file when the "CDT" keyword is used:

```
$ create_time = f$file_attributes(file_spec,"CDT")
```

Occasionally you may want to compare two absolute times to determine which one is later. To continue the example with F\$FILE_ATTRIBUTES, you may want to determine which of two files was most recently modified. The modification time of a file can be obtained with F\$FILE_ATTRIBUTES, but it will be in absolute time format. Absolute format is not suitable for comparison because the fields are not in sort order: the year field is not first and the month field is alphabetic. The F\$CVTIME function can be used to convert an absolute time to a **comparison time**, which has the following format:

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

A comparison time differs from an absolute time in two ways. The fields are in sort order, from most significant to least significant. And the month is represented as a number rather than as an abbreviation. One comparison time A is greater than another comparison time B if and only if A is later in time than B.

The `F$CVTIME` function can accept a rather complex combination of arguments. In order to convert an absolute time to a comparison time, the first argument must be the absolute time. The second argument is a keyword string specifying the type of conversion to be performed; the "COMPARISON" keyword requests that the absolute time be converted to comparison format and returned. The modification times of two files can be compared as follows:

```
$ time1 = f$cvtime(f$file_attributes(file1,"RDT"),"COMPARISON")
$ time2 = f$cvtime(f$file_attributes(file2,"RDT"),"COMPARISON")
$ if time1 .gts. time2 then action when first later than second
```

Always convert times to comparison format before trying to compare them.

VMS provides another time format called the **delta time**. A delta time specifies the difference between two absolute times. Another way to look at it is that a delta time specifies a time span, a certain amount of time. The format of a delta time is as follows:

ddd-hh:mm:ss.cc

In this format, *ddd* denotes some number of days, *hh* some number of hours, and so on. Taken together, the fields specify an amount of time from one hundredth of a second up to 9,999 days and 24 hours.

Delta times are rarely used by themselves; rather, they are usually used in combination with absolute times. An absolute time and a delta time specified together is called a **combination time**. A combination time has one of the following formats:

"absolute+delta"
"absolute-delta"

In these formats, *absolute* represents an absolute time and *delta* represents a delta time. The combination time must often be enclosed in quotation marks, so it is best always to do so. The first format specifies a time that is "delta" later than the absolute time, that is, a specific time some interval after the absolute time. The second format specifies a time that is "delta" earlier than the absolute time.

The following example sets the expiration date of a file to be 30 days after midnight tomorrow:

```
$ set file/expiration="tomorrow+30-00:00:00.00" myfile.dat
```

The combination time consists of the absolute time TOMORROW plus the delta time 30-00:00:00.00. DCL has a complex set of rules to allow you to omit certain fields from times when they are unnecessary. In this example, the minutes, seconds, and hundredths fields can be omitted from the delta time:

```
⌘ set file/expiration="tomorrow+30-00" myfile.dat
```

The author recommends that you specify all time fields until you become familiar with the rules for forming and abbreviating times. The rules are given in the *VMS DCL Concepts Manual*.

The F\$CVTIME function can also be used to convert a combination time to an absolute time. The first argument is the combination time and the second argument is the keyword "ABSOLUTE". The following example asks the user for a new file expiration date based on the file's creation date:

```
$   created = f$file_attributes(file_spec,"CDT")
$   display "The file was created ", created
$   libcall ask xda_delta s -
           "How long until it expires (delta time):"
$   expires = f$cvtime(created+"+"+xda_delta, "ABSOLUTE")
$   display "The file will expire on ", expires
$   set file/expiration="'"expires'" file_spec
```

The file's creation time is obtained and displayed. Then the user is asked for a delta time, which is used as the file's lifetime. The absolute expiration date is determined using the F\$CVTIME function. Note how the combination time is constructed by concatenating the absolute time, a plus sign, and the delta time. The expiration date is then displayed and the file set.

The F\$CVTIME function has many other features, which are described in the *VMS DCL Dictionary*.

16.5 **Record Structures**

Chapter 15 presented sequential files whose records contain textual information. Chapter 17 will introduce indexed files whose records are structured. Both sequential and indexed files can contain structured records, although such records are more common in indexed files. A structured record, or simply **structure**, is a record whose contents are formatted into **fields**, each field containing a single item of information. All the fields, except perhaps for the last one, are of fixed

Table 16.1 Fields in a Record Structure

Position	Size	Type	Description
0	12 bytes	String	User name.
12	17 bytes	Date	Date this record was created, in the form <i>dd-mmm-yyyy hh:mm</i> .
29	17 bytes	Date	Date this record was last modified.
46	64 bytes	String	Device/directory owned by this user.
110	32 bits	Integer	Number of files in the directory.
114	32 bits	Integer	Block count of files in the directory.
118	8 bits	Boolean	1 if the user is a student, 0 if not.

size. The last field can have variable size if the records in the file have variable length. The sizes of all the fields except the last are fixed, and the last field occupies the rest of the variable-length record.

The records in an index file are almost always structured. The records in a sequential file can contain textual information, as was the assumption in Chapter 15, or they can be structured. Sequential files with structured records are not specifically addressed in this book because the techniques are the same as for indexed files.

Using structures in DCL is somewhat difficult because there is no direct support for them. However, with the careful use of features already described, you can create new structures and access the fields from existing ones. For purposes of illustration we will use the record structure illustrated in Table 16.1.

In order to access and store the fields in a structure, we need to know the position, size, and data type of each field. This information is represented using one or two symbols for each field. The value of the first symbol is the field's position; the value of the second symbol is the field's size. The name of the first symbol also contains a one-letter code that identifies the data type of the field. The conventions for structure symbols are explained below. Here is the code that sets up the symbols for the sample structure:

```

$   ufr_t_username = 0
$   ufr_s_username = 12
$   ufr_t_created  = 12
$   ufr_s_created  = 17
$   ufr_t_modified = 29
$   ufr_s_modified = 17
$   ufr_t_devdir   = 46
$   ufr_s_devdir   = 64
$   ufr_l_count    = 110*8
$   ufr_l_blocks   = 114*8
$   ufr_b_student  = 118*8

```

The record contains information about a user's files, so it is called a user file record, or UFR. This structure has four different types of fields: integer, text, date, and boolean. The definition symbols required for each type are as follows:

Text. A field containing a text string is described by two symbols. The first symbol name contains a T to identify a text field, and its value is the byte position of the beginning of the field. The value of the second symbol is the size of the field, and the symbol name contains an S to identify it as a size symbol.

Date. A field containing a date and time is described by two symbols. The first symbol name contains a T to identify a text field, and its value is the byte position of the beginning of the field. The second symbol again specifies the size of the field.

Integer. An integer is a 32-bit integer quantity. A field containing an integer is described by one symbol. The symbol name contains an L to identify a longword integer field, and its value is the *bit* position of the beginning of the field. The reason for using the bit position will become apparent later. No size symbol is needed, because an integer field is always a four-byte (32-bit) field.

Boolean. A field containing a boolean value is described by one symbol. The symbol name contains a B to identify a boolean field, and its value is the *bit* position of the field. No size symbol is needed, because a boolean field is always a one-byte (eight-bit) field. The field contains a 0 for false, or a 1 for true.

The letters used to identify the field type are dictated by the VAX modular programming standard, which is described in the *Guide to Creating VMS Modular Procedures*.

A record is created by storing the values of its constituent fields into a character string, one at a time, from first to last. The substring and bit-field assignment statements are used to store the data into the character string. The following code will construct one of our sample records:

```
$   date = f$extract(0,ufr_s_created,f$time())
$   record = ""
$   record[ufr_t_username,ufr_s_username] := "F_SHUBIN"
$   record[ufr_t_created,ufr_s_created]   := "'date'"
$   record[ufr_t_modified,ufr_s_modified] := "'date'"
$   record[ufr_t_devdir,ufr_s_devdir]     := "$DISK3:[F_SHUBIN]"
$   record[ufr_l_count,32]                = file_count
$   record[ufr_l_blocks,32]               = file_blocks
$   record[ufr_b_student,8]              = false
.
.   write record
.
```

Before the record is created, the current time is retrieved, truncated after the minutes field, and stored in the symbol DATE. The record will be the value of symbol RECORD, so the symbol is initialized by setting it to the null string. Then each field is stored in the symbol, beginning with the one at position zero and continuing until all fields have been stored. Text and date fields are stored using a substring assignment statement. The substring's byte position and length are specified by the two symbols that define the field. Note that each assignment automatically extends the record to the required length so that the field will fit in it. The value assigned to a text or date field is extended with spaces to occupy the entire field. In this manner, every character of the record is filled with good data.

Integer and boolean fields are stored using a bit-field assignment statement. The field's *bit* position is specified by the symbol that defines the field. The field's size is always 32 bits for an integer and eight bits for a boolean. Remember that a substring position is specified by the starting byte position, while a bit-field position is specified by the starting bit position. Once all the fields have been stored, a complete record exists in the symbol RECORD. The record can now be written to a file or manipulated in some other fashion.

The character string representing a structure is composed of both textual fields and binary fields. If you display the string on a terminal, the textual fields will be apparent, but the binary fields may cause the terminal to behave strangely. This

is because the binary fields may appear to the terminal as control sequences or other nontextual data.

16.5.2 **Extracting Fields from a Structure**

In order to use the information in an existing record, the data in the record's fields must be extracted and assigned to symbols. Once the data is assigned to symbols, it can be manipulated by all the normal DCL facilities. Lexical functions are used to extract the fields of a record:

```
.  
: read record as described in next chapter  
.  
$   user_name   = f$edit(f$extract(ufr_t_username,ufr_s_username,-  
                                record),"TRIM")  
$   create_date = f$extract(ufr_t_created,ufr_s_created,record)  
$   modify_date = f$extract(ufr_t_modified,ufr_s_modified,record)  
$   dev_dir     = f$edit(f$extract(ufr_t_devdir,ufr_s_devdir,-  
                                record),"TRIM")  
$   file_count  = f$cvsi(ufr_l_count,32,record)  
$   file_blocks = f$cvsi(ufr_l_blocks,32,record)  
$   student     = f$cvsi(ufr_b_student,8,record)
```

Text and date fields are extracted with the F\$EXTRACT function. The field's position and size are specified by the two symbols that define the field. If the data might not fill the field, as is the case with user name and device/directory, the F\$EDIT function is used to trim the trailing spaces present in the field.

Integer and boolean fields are extracted with the F\$CVSI function. The acronym CVSI stands for "convert signed bit-field to integer." This function requires three arguments: a bit position, a size, and a value. The value is treated as a sequence of bits, and the bit-field specified by the position and size is extracted, assumed to represent a signed integer, and converted to an integer. This is exactly what we need when extracting an integer or boolean field, because the field was stored as a series of bits with a bit-field assignment statement.

There is a companion to the F\$CVSI function, called F\$CVUI, which extracts unsigned bit-fields. The F\$CVUI function can be used in place of F\$CVSI when the field is known to be positive. It is not used in this book because all bit-fields are assumed to contain signed (potentially negative) integers.

Chapter 17



Indexed File Operations

This chapter continues the discussion of file operations begun in Chapter 15. It deals with indexed files, which have a much richer structure than sequential files.

An indexed file consists of a set of **records**, each of which is composed of individual **fields**. A field is a portion of a record containing one item of information, such as a user name or a social security number. An indexed file record can contain any number of fields. There is no requirement that all the records in a file have the same field layout; there can be various types of records, each with its own layout. If all the records in a file have the same length, then the indexed file can have **fixed-length records**. Otherwise the file must contain **variable-length records**.

What distinguishes indexed files from sequential files are the **key fields** in a record. A key field, or simply **key**, is a field whose values are to be indexed for fast lookup. An index for each key field is maintained in a section of the file separate from the data records. The index for a particular key contains an entry for each record in the file, the entry matching the record's key with the record's location in the file. The index is sorted such that a given key can be found quickly and the corresponding record or records retrieved. The indexes thus allow records to be retrieved randomly by key.

Every indexed file has at least one key, called the **primary key**. There may be up to 254 additional keys, called **alternate keys**. Information describing the keys

is kept at the beginning of the indexed file in the **file prologue**. The primary key has some important attributes associated with it:

Name. A string describing the key. It can be up to 32 characters in length.

Number. An integer that specifies the number of the index. The primary index is number 0, alternate indexes are numbered starting with 1.

Type. A keyword that specifies the data type of the key. The VMS record management system supports binary, decimal, and character string keys, but DCL can only handle files with character string keys.

Position. An integer specifying the starting position of the key in the record. The first position is zero.

Length. An integer that specifies the length of the key. A string key can be up to 255 characters in length.

Duplicates. A boolean value that controls whether duplicate keys are allowed. When true, there can be more than one record with the same key value.

There are additional attributes, not described here. A complete description of all indexed file attributes can be found in the *Guide to VMS File Applications*.

Alternate keys have all the attributes of the primary key, plus some additional ones:

Changes. A boolean value that controls whether the value of the key can be changed when a record is updated.

Null Key. A boolean value that controls whether a **null value** is allowed in the key field. If a null value is present in an alternate key field, that field is not entered into the index for the key.

Null Value. A character specifying the null value for the key field. The field is considered null if it is full of this character. Typical null values are the NUL and space characters.

Table 17.1 Fields in the Sample Indexed File

Name	Position	Size	Type	Description
XUF_T_USER	0	12	String	VMS user name, the primary key.
XUF_T_DEPT	12	16	String	Department name, an alternate key.
XUF_T_PROJECT	28	16	String	Project name.
XUF_L_PEOPLE	44	4	Integer	Number of direct reports, if user is a manager.

17.1 *Sample File*

The remainder of this chapter focuses on index file features that are supported by DCL. There are other features available in conventional programming languages that are not described here. The examples in this chapter are based on an indexed file named XDA_USER-FILE.DAT. The records in this file are formatted as illustrated in Table 17.1. The file has two keys. The primary key is a VMS user name, which is a string of 12 characters in length. The alternate key is the user's department, which is a string of 16 characters in length. Duplicate keys are not allowed for the user name key, but they are allowed for the department key.

▷ Ch. 16

In order to read and write records in the indexed file, a command procedure must define the structure of the records. The technique for defining the fields in a structure can be used to define the fields in a record. One or two assignment commands are required to define the symbols for accessing each field. These assignment commands are included in the initialization code of the application's main procedure:

```
#! Define the fields in a user file record. The symbols for each
#! field are prefixed with XUF, for "XDA User File".
$
$ xuf_t_user = 0
$ xuf_s_user = 12
$ xuf_t_dept = 12
$ xuf_s_dept = 16
$ xuf_t_project = 28
$ xuf_s_project = 16
$ xuf_l_people = 44*8
```

Reading an Existing File

In order to read an existing indexed file, you must first open it. This is accomplished with the OPEN command, as with sequential files:

```
$ open/read xda_user_file xda_system:xda_user-file.dat;
```

The OPEN command checks that the file exists, determines whether the process is allowed to read it, and prepares it for reading. The /READ qualifier requests read access to the file, so writing will not be allowed.

Once an indexed file is open, there are two ways to read records from it: randomly by key or sequentially. In order to read a record by one of its keys, you must specify the key index number and the key value. If a record exists with that key, it is read from the file:

```
$ libcall ask xda_user s "Which user:" "" u
$ user = f$fa0("#AS",xuf_s_user,xda_user)
$ read xda_user_file/index=0/key=""'user'" record /error=5
$ goto 9
$5: libcall signal xda e nosuchuser -
      "User ''xda_user does not exist."
$9:
      .
      .display information
      .
```

In this example, the user is prompted for a user name. The user name is supplied as the primary key value when reading a record. If the record is found, the procedure displays information contained in it. If the record is not found, the error branch is taken and an error is signaled. The procedure's error handler will be invoked in the latter case.

The /INDEX qualifier specifies the number of the index in which the key lookup is performed. The /KEY qualifier specifies the value of the key enclosed in quotation marks. In order to find a record whose key exactly matches the specified key value, the key value must be prepared according to the following rules:

- The key lookup is sensitive to the case of the key. In the sample file, user names are stored in uppercase letters, so the call to ASK includes the U option to convert the input to uppercase.

- The value specified with the /KEY qualifier must be of the same length as the key. In the preceding code, the F\$FAO function is used to pad the user name to a full 12 characters. This is accomplished with the sharp sign (#) character in the field-width position of the !AS directive. Instead of using an explicit width, the sharp sign indicates that the width is specified by the next argument to the lexical function. In this case, the field width is specified as the size of the user field: XUF_S_USER.

There are ways to perform key lookup other than by exact match. These are described in Section 17.2.1.

Once a record is retrieved by key, additional records can be read sequentially. When the READ command is used without the /INDEX and /KEY qualifiers, DCL will read the record whose key is next in alphabetical order in the same index. This is useful if you want to read some or all of the records in alphabetical order. It is also useful when an index has duplicate keys: you can read all the records having the same key. The following code prompts the user for a department and displays all the users in that department:

```

$ libcall ask xda_dept s "Which department:" "" u
$ dept = f$fao("#AS",xuf_s_dept,xda_dept)
$ read xda_user_file/index=1/key=""'dept'" record /error=5
$ goto 9
$5: libcall signal xda e nodept "There are no users in ''xda_dept."
$9:
$ display "Users in department ", xda_dept, ":"
$10: display f$extract(xuf_t_user,xuf_s_user,record)
$ read xda_user_file record /end_of_file=19
$ if f$extract(xuf_t_dept,xuf_s_dept,record) .eqs. dept then -
    goto 10
$19:

```

The user is prompted for a department name, which is converted to uppercase letters and padded with spaces to 16 characters. A random read is performed to obtain the first record with the requested department. If no such record is found, an error is signaled. If there is at least one such record, a loop is executed to read all of them. The loop first displays the user name from the previous record. It then reads the next record from the file. Because the alternate index has been established as the current one, RMS tries to read the next record in alphabetical order by department. There are three possible outcomes:

1. There are no more records in the file. The /END_OF_FILE qualifier causes a branch to the end of the loop.

2. There is another record and the department field matches the requested one. The IF command branches to the beginning of the loop.
3. There is another record but the department field does not match. The IF command does not branch and thus falls through to the end of the loop.

You must concern yourself with these three cases whenever you read duplicate records from an index.

If a procedure performs a sequential read without first establishing an index and key, RMS reads records according to the primary index, beginning with the first record in alphabetical order.

17.2.1 **Key Matching**

The method used by RMS to select a record by key depends upon the length of the supplied key and the value of the /MATCH qualifier. The supplied key may be equal in length to the record keys, or it may be shorter. An error is signaled if it is longer. The /MATCH qualifier takes the values EQ (equal), GE (greater than or equal), or GT (greater than). The default is /MATCH=EQ. There is no facility for matching records whose keys are less than the supplied key, because indexed files cannot be read backwards.

If the supplied key is equal in length to the record keys, RMS selects a record based on the full key. If /MATCH=EQ is specified, RMS selects the first (or only) record with an exactly matching key. If /MATCH=GE is specified, RMS selects the first record with an exactly matching key, or the next key in order if there is no exact match. Finally, if /MATCH=GT is specified, RMS selects the next key in order, *skipping any and all exact matches*.

If the supplied key is shorter than the record keys, RMS selects records based on the first n characters of the key, where n is the length of the supplied key. This is called a **generic match**. The /MATCH qualifier is used in the same way as full matches, except that “equal” and “greater than” comparisons are performed using only the first n characters of the record keys.

If no records match the specified key, an error or end-of-file condition is signaled.

Assume that the sample file contains records with the following department keys: DEVELOPMENT, SALES, SERVICE, SERVICE. Note that there are two records for the service department. Here are examples of READ commands that access records by the department index:

```

$! Here we read the record for the SALES department:
$
$ read xda_user_file /index=1/key="SALES      "/match=eq record

-or-

$! Here we read the record for the SALES department:
$
$ read xda_user_file /index=1/key="S          "/match=ge record

-or-

$! Here we read the first record for the SERVICE department:
$
$ read xda_user_file /index=1/key="SALES      "/match=gt record

-or-

$! Here we use a generic match and read the record for the
$! DEVELOPMENT department:
$
$ read xda_user_file /index=1/key="D"/match=eq record

-or-

$! Here we read the record for the SALES department:
$
$ read xda_user_file /index=1/key="E"/match=ge record

-or-

$! Here we read the first record for the SERVICE department:
$
$ read xda_user_file /index=1/key="SA"/match=gt record

```

17.3 **Creating a File**

Unlike sequential files, indexed files cannot be created with the OPEN command. DCL does not provide a way to specify all the information needed to create the file and construct its indexes. The DCL programmer must therefore use the File Definition Language Facility to characterize and create an indexed file. This facility is described in the *VMS File Definition Language Facility Manual*. The File Definition Language (FDL) is a language in which all the attributes of an indexed file can be specified. The specification resides in a text file, which is assigned the file type FDL by convention. The CREATE command can create an indexed file according to the specifications in such an FDL file.

A complete description of FDL is beyond the scope of this book. However, the following points summarize the steps that must be taken to create an indexed file in DCL:

1. Use the FDL editor to create the text file, which describes the desired indexed file. The EDIT/FDL command invokes the FDL editor. It will ask a series of questions about the indexed file and ultimately generate an FDL file to describe it.
2. Include the FDL file with your DCL application files.
3. When the application needs to create the indexed file, it uses the CREATE/FDL command. This command creates an empty indexed file according to the specifications in the FDL file.
4. The application can then open the empty indexed file and write records in it.

The CREATE/FDL command is used as follows:

```
$ create/fdl=xda_system:xda_user-file.fdl -
    xda_system:xda_user-file.dat
    :open file and write records
    :
```

The /FDL qualifier specifies the FDL file containing the specification of the indexed file. The parameter specifies the indexed file to be created. The new file has no records in it. The next section describes how to write records in an indexed file.

Here is the FDL description of the sample file:

```
TITLE "XDA Sample File"

IDENT " 9-MAY-1988 10:55:09  VAX-11 FDL Editor"

SYSTEM
SOURCE                                VAX/VMS

FILE
ORGANIZATION                          indexed

RECORD
CARRIAGE_CONTROL                      carriage_return
FORMAT                                 fixed
SIZE                                    48
```

AREA 0	
ALLOCATION	29
BEST_TRY_CONTIGUOUS	yes
BUCKET_SIZE	2
EXTENSION	6
AREA 1	
ALLOCATION	4
BEST_TRY_CONTIGUOUS	yes
BUCKET_SIZE	2
EXTENSION	2
AREA 2	
ALLOCATION	13
BEST_TRY_CONTIGUOUS	yes
BUCKET_SIZE	1
EXTENSION	3
KEY 0	
CHANGES	no
DATA_AREA	0
DATA_FILL	100
DATA_KEY_COMPRESSION	yes
DATA_RECORD_COMPRESSION	yes
DUPLICATES	no
INDEX_AREA	1
INDEX_COMPRESSION	yes
INDEX_FILL	100
LEVEL1_INDEX_AREA	1
NAME	"VMS Username"
PROLOG	3
SEGO_LENGTH	12
SEGO_POSITION	0
TYPE	string
KEY 1	
CHANGES	no
DATA_AREA	2
DATA_FILL	100
DATA_KEY_COMPRESSION	yes
DUPLICATES	yes
INDEX_AREA	2
INDEX_COMPRESSION	yes
INDEX_FILL	100
LEVEL1_INDEX_AREA	2
NAME	"Department"
SEGO_LENGTH	16
SEGO_POSITION	12
TYPE	string

17.4

Writing a File

Before an application can write records in an indexed file, it must open the file for write operations:

```
$ open/read/write xda_user_file xda_system:xda_user-file.dat;
```

The `/WRITE` qualifier specifies that write operations will be performed. The `/READ` qualifier is also necessary so that DCL will assume that the file already exists. If just the `/WRITE` qualifier is specified, DCL will create a *new sequential* file, superseding the indexed file. Always open an existing indexed file with an `OPEN/READ/WRITE` command.

▷ Ch. 16

In order to write a new record in an indexed file, you must first construct a character string containing the record's contents. Once the record is constructed, it is written with the `WRITE` command:

```
. obtain user name and department
.  
$ record = ""  
$ record[xuf_t_user,xuf_s_user] := "'user'"  
$ record[xuf_t_dept,xuf_s_dept] := "'dept'"  
$ record[xuf_t_project,xuf_s_project] := ""  
$ record[xuf_l_people,32] = 0  
$ write/symbol xda_user_file record
```

▷ Ch. 3

The record is constructed in the symbol `RECORD`. The symbol is first cleared to make sure that any previous data is removed. Substring and bit-field assignment commands are used to initialize the four fields in the record. The substring assignment command (`:=`) requires the use of apostrophe substitution to assign the user name and department to their respective fields. Once the record is constructed, the `WRITE` command adds it to the indexed file. The `/SYMBOL` qualifier allows the symbol `RECORD` to contain records of up to 2,048 bytes, rather than restricting it to 255 bytes. Although a particular indexed file may contain short records, it is best to specify the `/SYMBOL` qualifier to avoid any future problems with record length.

If a procedure attempts to write a record with a duplicate key, and the index for that key does not allow duplicates, an error is signaled.

17.5

Updating a File

The WRITE command can be used to **update** an existing record in an indexed file. A record is updated by first reading it, then altering the data in the record, and finally rewriting the record. The following example increases by 1 the number of people reporting to a particular manager:

```
$ libcall ask xda_user s "Manager with new person:" "" u
$ user = f$fao("#AS",xuf_s_user,xda_user)
$ read xda_user_file/index=0/key=""'user'" record /error=5
$ goto 9
$5: libcall signal xda e nosuchuser -
      "User ''xda_user does not exist."
$9:
$ people = f$cvui(xuf_l_people,32,record)
$ record[xuf_l_people,32] = people + 1
$ write/symbol/update xda_user_file record
```

Once the record is read, the people count is extracted with the F\$CVUI function. The people field is then increased by 1. Finally, the record is updated with the WRITE/UPDATE command. The /UPDATE qualifier specifies that the record just read should be rewritten with new information. An update operation always rewrites the record last read.

An alternate key field may be updated with a new value only if the corresponding index was created with the “changes” attribute. A primary key can never be changed.

17.6

Deleting Records

Records can be individually deleted from an indexed file without the need to copy the entire file. The READ/DELETE command first reads a record from the file and then deletes the record. Once a record is deleted, there is no way to restore it except by rewriting it. The following example deletes a user record from the sample file:

```
$ libcall ask xda_user s "User to delete:" "" u
$ user = f$fao("#AS",xuf_s_user,xda_user)
$ read/delete xda_user_file/index=0/key=""'user'" record/error=5
$ goto 9
$5: libcall signal xda e nosuchuser -
      "User ''xda_user does not exist."
$9:
```

You must be careful when writing a procedure to delete records from an indexed file. It is quite easy to delete the wrong record or, when deleting multiple records, to delete too many records. Avoid using generic matching to delete records; specify exact key matches whenever possible. The following example deletes all users in a given department:

```
$ libcall ask xda_dept s "Delete which department:" "" u
$ dept = f$fa0("#AS",xuf_s_dept,xda_dept)
$ count = 0
$10: read/delete xda_user_file /index=1/key=""'dept'" -
      record /error=19
$ count = count + 1
$ goto 10
$19:
$ display "Users deleted: ", count
```

This code uses a loop to delete each record with the given department. Note that exact key matching is used, even though multiple records are being deleted. As soon as an error occurs, it is presumed to be a “key not found” error and the loop is terminated.

17.7

File Sharing

RMS provides a facility for **file sharing**, which allows multiple processes to access and modify an indexed file simultaneously. If a file contains data that can potentially be accessed by more than one user at a time, that file must be shared among all users. It is the OPEN command that determines whether a file can be shared.

By default, a file is opened so as to disallow file sharing. To allow sharing, you must specify the /SHARE qualifier on the OPEN command. This qualifier takes a value, either READ or WRITE. If you specify /SHARE=READ, other users are allowed to read the file but not to modify it. If you specify /SHARE=WRITE, other users are allowed to read and modify the file:

```
$ open/read/write/share=write xda_user_file -
      xda_system:xda_user-file.dat;
```

This OPEN command allows other processes to read and modify the file. When a file is shared, every procedure that opens the file must cooperate by specifying the /SHARE qualifier. If some procedure does not cooperate, one of two things can happen:

- If the uncooperative procedure opens the file first, no other procedure will be able to open it. When a file is opened without sharing, no other process can open it without signaling an error.
- If the uncooperative procedure attempts to open the file after some other procedure has opened it, the former procedure will fail. When a file is opened with sharing, every process must open it with sharing.

Once a file is opened for sharing, many processes can access and modify it simultaneously. This opens up a host of possibilities for damaging the contents of the file. Suppose two processes attempt to update a record at the same time? One process will update the record first, but its update will be immediately lost as the second process performs another update. RMS provides a facility called **record locking**, which helps maintain the integrity of shared files.

The concept behind record locking is simple: when a record is read from the file, it is locked. A locked record cannot be read or written by any other process. When the record is updated by the original reader, it is unlocked, allowing other processes to access it again. This simple concept gets more complicated, however, when all the various combinations of record accessing are considered. The `/LOCK` qualifier specifies whether a record is to be locked after it is read. A record is locked if `/LOCK` is specified on the `READ` command (the default). It is not locked if `/NOLOCK` is specified. When you are reading a record that will not be updated, use `/NOLOCK` to prevent other processes from being locked out of the record.

Here is the record update example with record locking:

```
$ libcall ask xda_user s "Manager with new person:" "" u
$ user = f$fa0("!#AS",xuf_s_user,xda_user)
$ read/lock xda_user_file/index=0/key=""'user'" record /error=5
$ goto 9
$5: libcall signal xda e nosuchuser -
      "User ''xda_user does not exist."
$9:
$ people = f$cvui(xuf_l_people,32,record)
$ record[xuf_l_people,32] = people + 1
$ write/symbol/update xda_user_file record
```

The record is locked by the `READ` command and remains locked until updated by the `WRITE` command. No other processes can read or update the record while it is locked.

A record is locked when it is read with the READ/LOCK command (remember, /LOCK is the default). The record remains locked until the procedure takes one of the following actions:

- The record is updated with the WRITE/UPDATE command.
- The record is deleted with the READ/DELETE command.
- Another record is read, with or without locking.
- A new record is written to the file with the WRITE command.
- The file is closed.

When a process attempts to read a record that is locked by another process, one of two things happens. If the record is read with a READ/LOCK command, an error is signaled. If the record is read with a READ/NOLOCK command, the record is read in spite of the lock. This latter behavior allows a procedure to read a record regardless of other process activity, as long as the procedure does not intend to update the record. If the procedure will update the record, it *must* lock it first or the record may be corrupted.

An error is signaled if a process attempts to lock a record that is already locked. This presents a problem to the procedure that wants to wait until the record is unlocked. The /ERROR qualifier can be used on the READ command to detect the lock error and jump to an error handler. However, any error will jump to the error handler, not just lock errors. The simplest solution to this problem is to write procedures that do not wait for locked records but rather just allow the error to be signaled. If this is unacceptable for your application, you must write an error handler that distinguishes lock errors from other errors.

The error status for locked records is called RMS\$_RLK and has the hexadecimal value %X000182AA. Define this error status in the main procedure as follows:

```
$ rms$_rlk = %x000182aa
```

The error handler for the READ command can check for this status and retry the read operation:

```

$10:  read/lock xda_user_file /index=0/key=""'user'" -
      record /error=15
$      goto 19
$15:  status = $status
$      if status .eq. rms$_rlk then goto 10
$      libcall signal xda e nosuchuser -
      "User ''xda_user does not exist."
$19:
      .update the record
      .

```

The read loop will repeat until the record is successfully read and locked. Then the record can be updated.

The sample application in Appendix D illustrates the use of shared indexed files and record locking.

Chapter 18



File Protection

Data security is an important aspect of many applications, particularly those involving the confidential data of your organization. VMS provides a data protection facility, which allows you to control which users can access a collection of data and which operations those users can perform on the data. The data protection facility is composed of two independent protection mechanisms: **user identification code (UIC)** protection and **access control list (ACL)** protection.

The UIC-based protection mechanism was the original data protection mechanism in VMS. It is grounded in the idea that each protected object is owned by a particular UIC. The relation between the owner UIC and the UIC of the user who is attempting to access the object determines if and how the user can access the object. UIC-based protection can be applied to the following kinds of objects:

- Devices
- Data volumes
- Files and directories
- Logical name tables
- Queues
- Global sections

The ACL-based protection mechanism was introduced in VMS Version 3 to provide a more flexible data security facility. The acronym ACL stands for

access control list: an arbitrary list of associations between user identifiers and access capabilities. When the user who is attempting to access the object appears in the object's ACL, then the ACL determines if and how the user can access the object. An ACL can be associated with all the kinds of objects listed above except for data volumes.

This chapter describes those features of the two protection mechanisms that are most often needed by VMS users, and DCL programmers in particular. The discussion is restricted to the protection of files and directories because these objects are the ones most commonly manipulated by DCL applications. The protection of the other kinds of objects is quite similar and should not pose a problem for a DCL programmer familiar with the information in this chapter. The *Guide to VMS System Security* describes the VMS protection mechanism in complete detail.

18.1 **UIC-Based Protection**

▷ Ch. 2

Every VMS user has a user identification code assigned by the system manager and specified in the user's record in the user authorization file. Every file and directory has an associated UIC, called its **owner UIC**. It is the relation between the file's owner UIC and the UIC of the user who is attempting to access the file that determines whether the user can actually access the file and which operations can be performed on it.

Consider the relation between the file's owner UIC and the accessing user's UIC. VMS defines four **access categories** based on the possible relation:

System. The accessor is in a system UIC group or has special privileges (described below).

Owner. The accessor is the owner.

Group. The accessor is in the same group as the owner (but is not the owner).

World. The accessor is in a different group than the owner.

The system category is the most privileged, or highest, access category, and the world category is the lowest. VMS places the accessor in the highest possible category when determining access capability. For example, if an accessor is the owner of the file but is also in a system UIC group, VMS places the accessor in the system category.

Once the accessor is placed in the appropriate category, VMS determines the operations that the accessor can perform by consulting the **protection mask**

Table 18.1 Access Needed for Directory Operations

Operation	Grandparent Dirs.	Parent Dir.	Directory
Create directory	R or E	W	—
Create file	R or E	R or E	R and W
Look up single file	R or E	R or E	R or E
Use wildcards or list directory	R or E	R or E	R
Rename file	R or E	R or E	R and W
Change directory attributes	R or E	R or E	C
Delete directory	R or E	W	D

associated with the file. The protection mask contains four access flags for each of the four access categories, a total of 16 flags. The four access flags and the operations they permit are as follows:

Read. The accessor can read the file.

Write. The accessor can write or update the file.

Execute. The accessor can execute the file (pertinent to executable images and DCL procedures).

Delete. The accessor can delete the file.

The accessor can perform an operation on the file if the operation is permitted by the category to which the accessor was assigned. Furthermore, the accessor can perform the operation if it is permitted by any lower category. For example, if the accessor was assigned to the group category and wants to delete the file, the accessor can do so if either the group or the world category allows delete access. You must keep in mind, however, that it is not just the protection mask of the data file that matters. In order to get at the file, VMS must begin at the device, travel down through the directory hierarchy, and finally arrive at the file itself. The protection masks on the device, directories, and file all play a part in determining how the file can be accessed. Table 18.1 lists the operations that can be performed on a directory and describes how the various protection masks determine whether an accessor can perform the operation. Table 18.2 does the same for operations on data files.

There is a fifth type of access, called control access, which has no explicit flags in the protection mask. A user with control access to a file can change the file's protection or other characteristics just as the owner of the file can. It is inherent in

Table 18.2 Access Needed for File Operations

Operation	Grandparent Dirs.	Directory	File
Read file	R or E	R or E	R
Write or modify file	R or E	R or E	R and W
Execute file	R or E	R or E	R or E
Change file attributes	R or E	R or E	C
Delete file	R or E	W	D

the UIC-based protection scheme that users in the system and owner categories have control access, while users in the group and world categories do not. When you need to grant control access in any other fashion, you must use an access control list.

If the accessor has certain privileges, UIC-based protection checking is altered in fundamental ways. The following privileges affect the protection-checking methodology:

BYPASS. All protection checking is completely bypassed. The accessor can perform any operation whatsoever on the file.

GRPPRV. If the accessor is in the same group as the owner, the accessor is placed in the *system* category (not the group category).

READALL. The accessor can read and control the file, regardless of its protection.

SYSPRV. The accessor is placed in the system category regardless of UIC.

18.1.1 Protection Mask Format

The protection mask for a file is specified and displayed in the following format:

`(system:rwed, owner:rwed, group:rwed, world:rwed)`

The access categories are listed from highest to lowest. In each category, the letter R indicates read access, W write access, E execute access, and D delete access. A letter is present if its corresponding access is allowed in that category, absent if not. Remember that control access is always allowed in the system and owner categories and disallowed in the group and world categories.

When you specify a protection mask, you can abbreviate the access category name down to one letter (e.g., G for group). Here is a protection mask that

allows all access for the system and owner category, read/write access for the group category, and no access for the world category:

```
(s:rwed,o:rwed,g:rw,w)
```

When no access is allowed, the colon is omitted along with the access letters.

18.1.2 Setting the Protection Mask

Some commands allow the protection mask to be specified when a file is created. These commands include APPEND, BACKUP, CREATE, and COPY. The following example creates a new file with a specified protection mask:

```
$ create xda_songs.dat /protection=(s:rwed,o:rwed,g:rwe,w:r)
```

The /PROTECTION qualifier includes the protection mask that is assigned to the file.

If a command does not allow a protection mask to be specified, then it must be established separately with the SET FILE command. The OPEN command does not accept a protection mask:

```
$ open/write xda_file xda_songs.dat
$ set file xda_songs.dat /protection=(s:rwed,o:rwed,g:rwe,w:r)
```

The SET FILE command can be used to change a file's protection mask at any time. Table 18.2 describes the file access that is required to change its owner or protection mask.

18.1.3 Displaying the Protection Mask

The DIRECTORY command can be used to display the protection mask for a file. It is also useful to display the file's owner at the same time:

```
$ directory/owner/protection xda_songs.dat
```

This command will display the file spec, owner, and protection mask.

18.1.4 **Obtaining the Protection Mask**

▷ Ch. 13

A DCL procedure can obtain the protection mask for a file with the lexical function `F$FILE_ATTRIBUTES`. The function returns the mask as a character string in the following format:

```
"SYSTEM=rwed, OWNER=rwed, GROUP=rwed, WORLD=rwed"
```

Note the presence of a space after each comma. Also note the use of an equal sign (=) rather than a colon after the category names. When a procedure needs to determine whether a particular type of access is allowed for an access category, it must parse the protection mask. The following code determines whether delete access is allowed for the world category:

```
$ mask = f$file_attributes(file_spec,"PRO")
$ world = f$element(3, ",", mask) - "WORLD"
$ world_can_delete = f$locate("D",world) .ne. f$length(world)
```

▷ Ch. 16

The first line obtains the protection mask for the file whose file spec is stored in the symbol `FILE_SPEC`. The second line determines the information for the world category by extracting the final comma-separated element from the protection mask (remember that the `F$ELEMENT` function numbers elements beginning with zero, so the final one is number 3). The category name "WORLD" is removed from the element, leaving only the equal sign and the access flags. The third line attempts to locate a "D" in the flags. If present, the world category has delete access. If absent, the world category does not have delete access.

The author suggests that you avoid parsing protection masks if at all possible. The problem with doing so is that the protection mask does not tell the whole story about the protection of a file, and so the procedure may obtain a false picture of the access allowed a particular user. The protection mask does not tell the whole story because the file may also have an access control list.

18.2 **Access Control Lists**

The ACL-based protection mechanism is used to grant or deny access to a file in a more fine-grained fashion than that allowed by UIC-based protection. An access control list consists of one or more entries that specify the access allowed a particular user or set of users. Each entry is called an **access control entry** (ACE). What makes the ACL more flexible than the UIC-based protection mask is the way in which sets of users can be identified.

Individual users or sets of users are identified by **rights identifiers**. Rights identifiers, or simply identifiers, are defined and maintained by the system manager using the AUTHORIZE utility. There are four kinds of identifiers:

- Each user is assigned an identifier whose name is usually identical to the user name. The value of the identifier is the user's UIC.
- Each user group is assigned an identifier. The value of the identifier is the UIC [*group-number*, *].
- There are some special system identifiers that name the various environments in which programs can run. These include BATCH, DIALUP, INTERACTIVE, LOCAL, NETWORK, and REMOTE.
- The system manager can define additional identifiers that name various collections of users or that are associated with particular applications.

When a user logs in, VMS creates a process to run the user's programs. Various identifiers are associated with this process, just as the user's UIC is. The process identifiers always include the identifier assigned to the user name by the system manager. The process identifiers also include some of the environment identifiers. For example, a normal interactive process has the INTERACTIVE and LOCAL identifiers. In addition, the system manager can assign other identifiers, which name applications that the user can run or groups of files that the user can access. This last category of identifier is open-ended: the system manager can invent all kinds of identifiers to associate with users.

When a user attempts to access a file that has an ACL, the user's process identifiers are matched against the ACEs making up the ACL. The matching is performed from left to right, starting with the first ACE and ending with the last. The leftmost ACE whose identifiers *are all held by the process* is the ACE used by VMS to determine the access allowed the user. No other ACEs are considered once a matching one is found. It is possible that no ACEs match any of the process identifiers. There are three potential outcomes of the matching attempt:

- If there is a matching ACE and it allows the access requested by the user, then the access is granted.
- If there is a matching ACE and it does not allow the access, then the access is denied except in certain special cases. See Section 18.4 for a description of the special cases.
- If there are no matching ACEs, then the UIC-based protection mask is used to determine whether access is granted.

The following sections describe two types of ACE.

18.2.1

Identifier ACEs

An identifier ACE controls the type of access allowed to users with particular identifiers. The identifier ACE has the following format:

```
(IDENTIFIER=identifiers, ACCESS=access-types)
```

The *identifiers* portion of the ACE specifies one or more rights identifiers that this ACE matches. The identifiers can be specified in UIC format or as simple identifier names. The *access-types* portion of the ACE specifies the type of access allowed to users with the identifiers listed. The access types are READ, WRITE, EXECUTE, CONTROL, DELETE, and NONE. If multiple identifiers or access types are specified, they must be connected with plus signs (+).

Here are some examples of ACEs that might appear on a file. The following ACEs allow read, write, execute, and control access for all members of group DEVELOPMENT. A group can be specified in UIC format or simply as the name of the group.

```
(identifier=[development,*],access=read+write+execute+control)
```

--or--

```
(identifier=development,access=read+write+execute+control)
```

Assume that the system manager has defined an identifier called PAYROLL, which is associated with the payroll application and assigned to those users who have access to the application. The following ACE allows those users to read, write, and control a file:

```
(identifier=payroll,access=read+write+execute)
```

The payroll administrator decides that the payroll application can only be used by people logged in at the office. It cannot be used by people who have dialed in over a modem or logged in remotely from another node in the network. The following ACEs can be added to the files to prevent unauthorized access:

```
(identifier=network,access=none),  
(identifier=dialup,access=none),  
(identifier=remote,access=none),  
(identifier=payroll,access=read+write+execute)
```

The first three ACEs disallow access to network processes (i.e., to people trying to access the file from a remote node), users on dialup terminals, and users hosted from another network node. If an accessor does not match any of the first three ACEs, then the accessor is logged in at a local terminal. The final ACE matches users with the PAYROLL identifier, just as in the previous example. The order of these four ACEs is critical. If the payroll ACE were first, then any user with the PAYROLL identifier could access the payroll application regardless of how the user had logged in. Remember, VMS selects the first ACE whose identifiers match those of the accessor.

The following example allows any batch jobs to read a file, except that users JONES and SMITH cannot access the file at all:

```
(identifier=jones,access=none),  
(identifier=smith,access=none),  
(identifier=batch,access=read)
```

Again, the order of the ACEs is critical. Specific user identifiers must be matched before more general identifiers so that access is denied the specific users even if they have the general identifiers.

A special kind of identifier ACE, called a default identifier ACE, can be placed in the ACL associated with a directory. This ACE is automatically included on the ACLs of any files *subsequently* created in that directory. The format of a default identifier ACE is as follows:

```
(IDENTIFIER=identifiers,OPTIONS=DEFAULT,ACCESS=access-types)
```

It is identical to a normal identifier ACE except for the inclusion of the string OPTIONS=DEFAULT between the identifiers and the access types. Once this ACE is added to a directory's ACL, any new files created in the directory are automatically assigned the ACE (minus the DEFAULT option). If there are multiple ACEs with the DEFAULT option, the file is assigned all the default ACEs in order.

Other ACE options are described in the following section and in the *Guide to VMS System Security*.

18.2.2

Default Protection ACEs

A default protection ACE can reside only on the ACL of a directory. It specifies the UIC-based protection mask to be assigned to new files in the directory when no explicit mask is given. The default protection ACE applies to files that are created in the directory or in any of its subdirectories with no default protection ACE of their own. A default protection ACE has the following format:

```
(DEFAULT_PROTECTION, protection-mask)
```

The *protection-mask* portion of the ACE is specified in the standard system, owner, group, world format.

The following ACE, appearing on the ACL of a directory, will assign the specified protection mask to files created in that directory. The mask allows full access to the system and owner categories, read/write access to members of the group, and no access to others:

```
(default_protection, s:rwed, o:rwed, g:rw, w)
```

18.2.3

Modifying an ACL

There are two ways to modify the ACL of a file. The first method is to use the ACL editor. The ACL editor allows you to add, modify, and delete ACL entries interactively, in a manner similar to text editing. You can invoke the ACL editor with the following command:

```
$ edit/acl file-spec
```

The ACL editor is not described in detail in this book. See the *VMS Access Control List Editor Manual* for a description of the ACL editor.

The second method of modifying an ACL is to use the SET ACL command. The command requires one parameter, a file spec that identifies one or more files whose ACLs are to be modified. The command also accepts a host of qualifiers, which specify operations to be performed on the ACLs, along with existing or new ACEs involved in the operations. The SET ACL command is used in a DCL procedure to add new ACEs to an ACL or to delete existing ones. Table 18.3 describes many of the qualifiers accepted by the SET ACL command.

Some of the qualifiers to SET ACL specify ACEs that are to be removed from the ACL. In this case, only the identifier portion of the ACE need be included; the options and access types do not participate in selecting ACEs to be removed.

Table 18.3 SET ACL Command Qualifiers

Qualifier	Description
<i>/ACL=list-of-aces</i>	Specifies a list of ACEs to participate in the operation. If neither the <i>/DELETE</i> nor the <i>/REPLACE</i> qualifiers are included, the ACEs are added to the ACL at the position specified by the <i>/AFTER</i> qualifier.
<i>/AFTER=ace</i>	Specifies the position in the ACL at which the ACEs specified by the <i>/ACL</i> qualifier are inserted. They are inserted after the ACE whose identifiers are named by this qualifier. If no <i>/AFTER</i> qualifier is included, the ACEs are inserted at the beginning of the ACL.
<i>/DEFAULT</i>	If the file being modified is a directory, its ACEs are removed and replaced with the ACEs of its parent directory. If the file being modified is not a directory, its ACEs are removed and replaced with the <i>default</i> ACEs of its parent directory (those specified with <i>OPTIONS=DEFAULT</i>).
<i>/DELETE</i>	The ACEs specified by the <i>/ACL</i> qualifier are deleted.
<i>/NEW</i>	All existing ACEs are removed from the ACL before the operation specified by the other qualifiers is performed.
<i>/REPLACE=list-of-aces</i>	The ACEs specified by the <i>/ACL</i> qualifier are removed, and then the ACEs listed in this qualifier are inserted at the position occupied by the rightmost ACE removed.

In the following example, three ACEs are added to the ACL of a file. They are added to the beginning of the ACL because the */AFTER* qualifier is not included:

```
$ set acl/acl=((identifier=jones,access=none),-
              (identifier=smith,access=none),-
              (identifier=batch,access=read)) payroll.dat
```

If these are the first ACEs ever added to the file, everything works fine. However, assume that the payroll file already had the following ACL:

```
(identifier=taylor,access=read),
(identifier=payroll_dept,access=read+write+execute+control)
```

The SMITH and JONES ACEs can be added at the beginning of the ACL. However, the BATCH ACE must be added at the end so that people in the payroll department receive full access even when running batch jobs. The following two SET ACL commands are required to add the new ACEs correctly:

```
$ set acl/acl=((identifier=jones,access=none),-
              (identifier=smith,access=none)) payroll.dat
$ set acl/acl=(identifier=batch,access=read) -
              /after=(identifier=payroll_dept) payroll.dat
```

Here is the data file's ACL so far:

```
(IDENTIFIER=[AMCDEV, JONES], ACCESS=NONE),
(IDENTIFIER=[AMCDEV, SMITH], ACCESS=NONE),
(IDENTIFIER=[AMCDEV, TAYLOR], ACCESS=READ),
(IDENTIFIER=PAYROLL_DEPT, ACCESS+READ+WRITE+EXECUTE+CONTROL),
(IDENTIFIER=BATCH, ACCESS=READ)
```

If the SET ACL command specifies an ACE that already exists in the ACL, the old ACE is replaced with the new one. However, its position in the ACL is not maintained; you must specify the position with the /AFTER qualifier. The following example replaces the batch ACE to allow both read and execute access:

```
$ set acl/acl=(identifier=batch,access=read+execute) -
              /after=(identifier=payroll_dept) payroll.dat
```

And now the payroll file's ACL looks like this:

```
(IDENTIFIER=[AMCDEV, JONES], ACCESS=NONE),
(IDENTIFIER=[AMCDEV, SMITH], ACCESS=NONE),
(IDENTIFIER=[AMCDEV, TAYLOR], ACCESS=READ),
(IDENTIFIER=PAYROLL_DEPT, ACCESS+READ+WRITE+EXECUTE+CONTROL),
(IDENTIFIER=BATCH, ACCESS=READ+EXECUTE)
```

The payroll department decides to split its personnel into two categories, those who can read the payroll file and those who can write it. The PAYROLL_DEPT identifier is replaced with two new identifiers that are named PAYROLL_READ and PAYROLL_WRITE. The following command removes the old identifier and adds the new ones in the same position:

```
$ set acl/acl=(identifier=payroll_dept) -
              /replace=((identifier=payroll_read,access=read+execute),-
                       (identifier=payroll_write,-
                        access=read+write+execute+control)) -
              payroll.dat
```

The final ACL on the payroll data file is:

```
(IDENTIFIER=[AMCDEV, JONES], ACCESS=NONE),  
(IDENTIFIER=[AMCDEV, SMITH], ACCESS=NONE),  
(IDENTIFIER=[AMCDEV, TAYLOR], ACCESS=READ),  
(IDENTIFIER=PAYROLL_READ, ACCESS=READ+EXECUTE),  
(IDENTIFIER=PAYROLL_WRITE, ACCESS=READ+WRITE+EXECUTE+CONTROL),  
(IDENTIFIER=BATCH, ACCESS=READ+EXECUTE)
```

18.3 *Default File Protection*

Sections 18.3.1 and 18.3.2 describe how VMS determines the default protection mask and ACL for a new directory or file. A default protection mask is only required when you do not specify one explicitly in the command that creates the directory or file. A default ACL is always required, because there is no way to specify one in the commands that create directories and files.

18.3.1 *Directories*

The default protection mask for a new directory is determined as follows:

- It is always identical to the protection mask of its parent directory.

The default ACL for a new directory is determined as follows:

- It is identical to the ACL of its parent directory, except that ACEs with the NOPROPAGATE option are not included in the default ACL.

The NOPROPAGATE option can be included in a directory ACE to prevent it from being propagated to new subdirectories.

18.3.2 *Files*

The default protection mask for a new file is determined by following these steps:

1. If the file is a new version of an existing file, its protection mask is the same as the previous version.
2. Otherwise, if the parent directory has a default protection ACE, the protection mask is taken from the ACE.
3. Otherwise, the process default protection mask is used.

The process default protection mask is specified by the VMS system generation parameter RMS_FILEPROT, which is established by your system manager. You can determine its value by using the following command:

```
$ show protection
  SYSTEM=RWED, OWNER=RWED, GROUP=RE, WORLD=NO ACCESS
```

The protection mask shown in this example is the standard default. You can change the process default with the SET PROTECTION/DEFAULT command:

```
$ set protection=(s:rwed,o:rwed,g,w) /default
```

This new protection mask denies access to users in the same group. Once the default protection mask is set, it is used in step 3 above.

The author recommends that you do not change the process default protection in a command procedure. First, it is better to be explicit and to specify the protection mask on the command that creates the file. Second, if the procedure changes the default protection, then it is responsible for restoring it to its original value before the procedure exits. This is more trouble than it is worth. Remember, even if you do change the default protection mask, the file may still receive its mask from the parent directory's default protection ACE. If the procedure requires a particular protection mask, it must specify it explicitly.

The default ACL for a new file is determined by following these steps:

1. If the file is a new version of an existing file, its ACL is the same as the ACL of the previous version. However, ACEs with the NOPROPAGATE option are not included in the new ACL.
2. Otherwise, if the parent directory has any identifier ACEs with the DEFAULT option, these ACEs make up the ACL of the new file.
3. Otherwise, the new file has no ACL.

The NOPROPAGATE option can be included in an ACE to prevent it from being propagated to new versions of the file.

A special rule is applied when you create a file whose owner is not your own UIC (e.g., using the /OWNER_UIC qualifier on the CREATE command). In this case, the file receives an additional ACE, which grants the owner's access capabilities to your UIC. In addition, it grants control access to your UIC. Therefore, regardless of the owner of a file, its creator retains control over the file.

18.4

The Access Algorithm

The exact algorithm used by VMS to perform UIC-based and ACL-based protection checking is relatively complex. In particular, there are some not-so-obvious interactions between the two protection mechanisms. Assume that a particular user (the accessor) is requesting a particular type of access (e.g., write access) to a file. VMS takes the following steps to determine whether access is granted:

1. If the file has no ACL, go to step 2. Otherwise perform the ACL-based protection check:
 - a. If the ACL includes an identifier ACE that grants access, then access is granted.
 - b. If the accessor is not identified in the ACL, go to step 2.
 - c. The ACL includes an identifier ACE that denies access. However:
 - i. If the accessor is the owner of the file and the owner category in the protection mask grants access, then access is granted. If it does not grant access, then go to step 3.
 - ii. If the accessor is in the same group as the file's owner, then go to step 3.
 - iii. Go to step 3b.
2. Perform the UIC-based protection check:
 - a. If the accessor is the owner of the file and the owner category of the protection mask grants access, then access is granted.
 - b. If the world category of the protection mask grants access, then access is granted.
 - c. If the accessor is in the same group as the file's owner and the group category of the protection mask grants access, then access is granted. If it does not grant access, then go to step 3.
 - d. Go to step 3b.

3. Check various privileges that affect access:
 - a. If the accessor has GRPPRV privilege and the system category of the protection mask grants access, then access is granted.
 - b. If the accessor has system access and the system category of the protection mask grants access, then access is granted.
 - c. If the accessor has BYPASS privilege, then access is granted.
 - d. If the accessor has READALL privilege and wants only read and/or control access, then access is granted.
 - e. Access is denied.

Chapter 19



Devices

In classical terms, a **device** is a hardware component attached to a computer for purposes of data storage or input/output. Such devices include disk drives, magnetic tape drives, printers, terminals, and so on. In the VMS environment, the term *device* includes all these hardware components, along with **virtual devices** created by the VMS software. Two examples of virtual devices are mailboxes used for interprocess communication and windows on a VAXstation monitor.

The purpose of some devices, particularly disk and tape drives, is to store permanent information. The magnetic medium on which information is stored is called a **volume**. Tape drives and certain disk drives have removable volumes, so a drive may contain different volumes at different times. Many newer disk drives have fixed volumes, which cannot be removed.

DCL applications often describe or manipulate devices, usually hardware components. For example, an application might display information about the disks on a system or mount a tape volume on a tape drive. This chapter describes methods for obtaining device information and manipulating devices in DCL.

19.1

Device Names

Each device on a VAX system has a unique **device name**. The name is used to refer to the device in file specs and DCL commands that require a device name, such as MOUNT. In the early days of VMS, the format of device names was quite

simple. The name consisted of two letters, which identified the type of device; a single letter, which identified the device controller; and an integer, which identified the particular unit attached to the controller. For example, the device name TTB3 identified a hard-wired terminal (TT) attached to the B controller and having unit number 3. Users began to *assume* that all device names consisted of four characters and that the type of device could be deduced from the first two letters.

With the advent of VAXclusters, terminal servers, workstations, and other innovations, such assumptions about a device name are no longer valid. Here are some of the developments that render the assumptions obsolete:

- Not all devices have physical incarnations. A mailbox, for example, is a logical device used for communication between processes. A window on a VAXstation display screen is a logical device, which is similar to a terminal, but many such devices may be assigned to the same display screen.
- Devices that are available to all nodes in a VAXcluster have a segmented name. The first segment is the name of the node to which the device is connected or the allocation class of the device. The second segment is a conventional device name. The segments are separated by a dollar sign. A disk connected to node BIZET might have the name BIZET\$DUA2. A disk connected to two HSC controllers in allocation class 1 might have the name \$1\$DUA3.
- A terminal server supports many physical terminals that are not connected directly to a VAX. The terminal server is a single device that appears to the user as multiple devices.

It is not valid to assume that a device name is four characters long, that the first two characters accurately identify the type of device, or that the device is physically connected to the local VAX.

19.2

Device Information

The characteristics of a device cannot be determined from its name, but they can be determined using the `F$GETDVI` lexical function. This function requires two arguments. The first is a string containing the device name. The second is a keyword string specifying the item of information desired. The information is returned as an integer, string, or boolean, depending upon the item requested.

The device name is usually specified as the name by itself, but you can also specify a file spec or a logical name that includes the device. The lexical function

Table 19.1 F\$GETDVI Items

Keyword	Type	Description of Result
"ALL"	Boolean	True if the device is allocated to a user, false if it is available.
"DEVCLASS"	Integer	The class of the device. See the <i>VMS DCL Dictionary</i> for a list of classes.
"DEVNAM"	String	The name of the device.
"DEVTYPE"	Integer	The type of the device. See the <i>VMS DCL Dictionary</i> for a list of types.
"EXISTS"	Boolean	True if the device exists, false if not.
"FREEBLOCKS"	Integer	The number of available blocks on a disk device.
"FULLDEVNAM"	String	The fully qualified device name, which includes the node name in a VAXcluster environment.
"MAXBLOCK"	Integer	The number of blocks on a disk device.
"MNT"	Boolean	True if a volume is mounted on a disk or tape device, false if not.
"VOLNAM"	String	The name of the volume mounted on a disk or tape drive.

ignores everything but the device name. Table 19.1 describes a few of the item keywords; many more are available.

If F\$GETDVI is called with an invalid device name, it signals an error. If you are not absolutely sure that a device name is valid (e.g., it was entered by a user), call F\$GETDVI with the "EXISTS" keyword. This is the one exception to the rule that the function signals an error for invalid devices. Instead, it returns a true value if the device exists, a false value if it does not. If the device does not exist, you cannot use F\$GETDVI to obtain any other information about it.

Once you know that a device exists, you can use any of the item keywords to obtain information about that device. One important item is the device class ("DEVCLASS"). The device class denotes the general category of device, such as disk, tape, terminal, or line printer. Each class is represented by an integer; a complete list of device classes is available in the *VMS DCL Dictionary*. The device class is useful when the user has entered a device to be used for a specific purpose, such as the target of a backup operation. In this case, the procedure can verify that the device is a disk or tape.

Another important item is the device name ("DEVNAM"). It may seem silly to have an item that returns the device name, but remember that the first argument to F\$GETDVI need not be a device name by itself. It may be a logical name or a full file spec. In these cases, the device name item is used to extract the actual device name, by itself, without any other elements of the file spec.

Here is an example of the use of F\$GETDVI:

```
$10:  libcall ask xda_dev s "Device on which to perform backup:"
$      dev_ok = f$getdvi(xda_dev,"EXISTS")
$      if dev_ok
$      then
$          backup_dev_name = f$getdvi(xda_dev,"DEVNAM")
$          backup_dev_class = f$getdvi(xda_dev,"DEVCLASS")
$          dev_ok = backup_dev_class .eq. 1 .or. -
$                  backup_dev_class .eq. 2
$      if .not. dev_ok then libcall signal xda i baddevclass -
$                          "The device must be a disk or tape."
$      else
$          libcall signal xda i nosuchdev -
$                          "Device ''xda_dev does not exist."
$      endif
$      if .not. dev_ok then goto 10
$19:
```

This example uses a loop to repeatedly ask the user for a backup device until a valid disk or tape name is entered. Once the loop terminates, the symbol BACKUP_DEV_NAME contains the device name and BACKUP_DEV_CLASS contains its class number.

The F\$GETDVI lexical function can return an astounding amount of information about a device. You should consult the *VMS DCL Dictionary* for complete details.

19.3 Obtaining a List of the Devices on a System

You may find yourself implementing a particularly sophisticated DCL application, which needs to obtain a list of every device on the system or perhaps all the disks or terminals. For example, an application that monitors the free space on mounted disks needs a list of all disks. Unfortunately, there is no straightforward way to obtain such a list in DCL. You must obtain the list using a somewhat devious technique.

First of all, generate a list of all devices, using the `SHOW DEVICE` command. The output from the command can be directed to a file with the `/OUTPUT` qualifier. The `/MOUNTED` qualifier can be used to restrict the list to only those devices with volumes mounted on them. The file will contain a list in roughly the following format:

Device Name	Device Status	Error Count	Volume Label	Free Blocks	Trans Count	Mnt Cnt
DUA0:	Mounted	4	MICROVMS	12569	88	1
DUA1:	Mounted	0	VXMASTER	13449	7	1
DUA2:	Mounted	14	USER	39051	8	1

Device Name	Device Status	Error Count	Volume Label	Free Blocks	Trans Count	Mnt Cnt
MSA0:	Online	5				
:						
:						
XQA0:	Online	0				
XQA1:	Online	0				
XQA2:	Online	0				

Note that there are blank lines, heading lines, and lines describing devices. A list of devices is obtained by opening the file and reading the lines, one at a time, looking for those lines that contain a device description. The device name is extracted from the line and added to the list. The following code builds a list of all tape drives:

```
$ libcall unique_name xda_show sys$scratch:xda?.lis
$ show device/output='xda_show
$ open/read xda_show 'xda_show
$ tape_list = ""
$10: read xda_show line /end_of_file=19
$ i = f$locate(":", line)
$ if i .eq. f$length(line) then goto 10
$ dev = f$extract(0, i, line)
$ if f$getdvi(dev,"EXISTS") then -
    if f$getdvi(dev,"DEVCLASS") .eq. 2 then -
        tape_list = tape_list + dev + ","
$ goto 10
$19: close xda_show
$ delete 'xda_show;*
```

Notice how careful the code is to ensure that what it finds is indeed a tape device. If the line does not contain a colon, then it is blank or a heading and is ignored. Once the device name is extracted, `F$GETDVI` is used both to ensure that the name represents an existing device *and* that the device is a tape drive. The existence check may seem superfluous, but it serves two purposes. First, it guarantees

that a real device name has been extracted from the device listing. Second, it guards against a device that existed when the `SHOW DEVICE` was performed but has subsequently disappeared (e.g., a temporary mailbox or remote terminal). When the loop terminates, the symbol `TAPE_LIST` contains a list of the tape drives available to the system.

Do not extract any additional information from the device listing file; instead use `F$GETDVI`. The format of the listing file may change in a future release of VMS. We are taking enough of a chance that the device name begins in column zero and ends with a colon. Never extract information from listing files unless *absolutely necessary*. VMS makes no guarantee that the format of listings will remain the same.

The `SHOW DEVICE` command accepts a generic device name parameter, in which case it includes only those devices in the listing. For example, `SHOW DEVICE T` produces a listing of devices whose name begins with the letter T. It is tempting to use this command to produce a list of the terminals on the system. Don't forget, however, that not all terminal names begin with T; DECserver terminals begin with LT. Always use `F$GETDVI` to determine the class of a device.

19.4 **Terminal Characteristics**

Associated with each terminal device is information called the **terminal characteristics**. Some of the characteristics specify features of the terminal, for example, whether it accepts ANSI escape sequences or is connected to a modem. Other characteristics control the behavior of the terminal, for example, whether line editing is allowed or whether lines wrap when the cursor reaches the right margin. Terminal characteristics can be modified with the `SET TERMINAL` command, and there is an `F$GETDVI` item keyword that can be used to obtain each characteristic. Most terminal characteristics do not affect a DCL procedure, because DCL has only rudimentary facilities for dealing with terminal displays and keyboards.

One characteristic that a DCL application might use is the "ANSI CRT" characteristic ("`TT_ANSICRT`"). This is a boolean characteristic, which is true if the terminal accepts ANSI standard control sequences, false if not. An application that sends control sequences to the terminal for screen formatting should first check the terminal to determine whether the sequences will be obeyed:

▷ Ch. 6

```

$!  Make sure we have an ANSI terminal to display on.
$
$  output_dev = f$getdvi("sys$output","DEVNAM")
$  if f$getdvi(output_dev,"DEVCLASS") .eq. 66
$  then
$    if .not. f$getdvi(output_dev,"TT_ANSICRT") then -
        libcall signal xda f notansi -
            "This application only runs on ANSI terminals."
$  else
$    libcall signal xda f notterm -
        "This application requires a terminal."
$  endif

```

The code determines the device to which output will be displayed. It then checks to make sure the device is a terminal, in particular an ANSI terminal. If either check fails, a fatal error is signaled and the application terminates.

19.5 **Device Allocation**

A procedure can request exclusive use of a device by **allocating** the device. Once a device is allocated to a process, no other process can use the device until the original process deallocates it. The devices most commonly allocated are disk and tape drives. A procedure might allocate a disk drive in order to initialize a new disk volume and prepare it for use by some application. A procedure might allocate a tape drive for use in a disk backup operation.

A device is allocated with the `ALLOCATE` command. The `ALLOCATE` command has three forms, described in the following paragraphs.

In its simplest form, the `ALLOCATE` command accepts a list of devices, one of which it allocates. It checks the devices in the order specified and allocates the first one available. The command also accepts an optional second parameter, which must be a logical name. This logical name is defined to refer to the allocated device, so you can tell which one was actually allocated. The following example requests exclusive use of tape drive `MTA0` or `MTA1`:

```
$  allocate mta0,mta1 xda_tape_drive
```

VMS first tries to allocate `MTA0` and then `MTA1`. The first available drive is allocated to the process. If neither are available, an error is signaled. The process logical name `XDA_TAPE_DRIVE` is defined as `MTA0:` or `MTA1:`, as appropriate.

The `ALLOCATE` command can also accept a group of devices; it will try to allocate one of the devices in that group. A group can consist of all the devices with a

particular two-letter type (e.g., MT), or all the devices on a particular controller (e.g., DUA). The following example attempts to allocate an RX02 floppy diskette drive:

```
$ allocate dy xda_floppy_drive
```

VMS tries to allocate any device whose name contains the device type DY, that is, any RX02 floppy drive. If one is allocated, the logical name XDA_FLOPPY_DRIVE is defined to refer to it.

The first two letters of a device name do not always refer to a specific type of device. For example, DU is used for the name of the entire RA family of disks, including the RA80 and RA81. If you want to allocate a specific type of device, use the third form of the ALLOCATE command. Include the /GENERIC qualifier and a list of device types rather than device names:

```
$ allocate/generic ra81 xda_disk_drive
```

VMS tries to allocate any RA81 device, ignoring all other disk types, including the RA80.

The /LOG qualifier can be used to control information displayed by the ALLOCATE command. If /LOG is specified (the default), then the command displays the name of the allocated device. In addition, if the logical name is already defined, the command displays a message explaining that the logical name's value is being superseded. The /NOLOG qualifier suppresses both of these messages.

The following example asks the user for the type of a device to be allocated and performs the allocation:

```
$10: libcall ask xda_dev s "Type of device to allocate:"
$ set noon
$ allocate/generic/nolog 'xda_dev xda_device
$ status = $status
$ set on
$ if status then goto 19
$ libcall signal xda i cantalloc -
    "No ''xda_dev can be allocated." 'status
$ goto 10
$19:
$ display "Device ''f$trnlm("xda_device") was allocated."
```

The code contains a loop, which asks the user for a device type and attempts to allocate such a device. Once the allocation is successful, the procedure displays the name of the device by translating the logical name XDA_DEVICE. Note the use of the /NOLOG qualifier on the ALLOCATE command to prevent its status messages from being displayed.

A device is deallocated using the DEALLOCATE command. The logical name defined by the ALLOCATE command is specified as the name of the device to deallocate:

```
$ deallocate xda_device
```

The /ALL qualifier can be used to deallocate all devices allocated by the current process:

```
$ deallocate/all
```

All devices allocated to a process are deallocated when the process is deleted. In particular, all devices allocated during an interactive session are deallocated when the user logs out.

19.6 **Volume Initialization**

A new disk or magnetic tape volume must be **initialized** before data can be recorded on it. The INITIALIZE command prepares a volume for use by VMS. The command requires two parameters, the first of which is the name of the device on which the new volume is mounted. The second parameter is the **volume label** for the new volume. The volume label is limited to 12 characters for a disk volume, six for a magnetic tape volume. The label is recorded on the volume and identifies it for future use.

The INITIALIZE command formats the volume for use by VMS. By default, a disk is formatted according to “Files-11 Structure Level 2.” This is the standard format for disks that will contain VMS directories and files. Again by default, a tape is formatted according to “level 3 of the ANSI standard for magnetic tape labels and file structure for informational interchange” (ANSI X3.27-1978). This is the current standard format for magnetic tapes. Within these standards there are many options, which affect the logical format of the volume. These options can be specified using a large number of qualifiers to the INITIALIZE command. These qualifiers are described in the *VMS DCL Dictionary*.

The following example allocates a tape drive and initializes a new magnetic tape volume:

```
$ allocate mt xda_tape_drive
$ libcall ask xda_label s "Label for new tape volume:" "" u
$ display "Please load a new tape volume on drive ", -
    f$strnlm("xda_tape_drive")
$ libcall ask xda_ s "Press RETURN when ready:" continue
$ initialize/density=1600 xda_tape_drive 'xda_label
```

This code allocates an MT tape drive. It then asks the user for a label for the new volume and requests that the volume be mounted on the drive. Finally, the new volume is initialized. The /DENSITY qualifier specifies the density (in bytes per inch) at which the volume is written.

A volume cannot be initialized if it has already been mounted with the MOUNT command, described in the next section.

19.7 Volume Mounting

The term **mounting** refers to the process by which the system recognizes a data volume on a storage device, such as a disk or magnetic tape drive. Not until a volume is mounted can its files be created and manipulated by VMS programs. Before a volume can be mounted it must be physically **loaded** on the device by a human being. Be careful to distinguish the physical loading process from the system's mounting process.

The MOUNT command is used to mount a volume. It has the following general format:

```
$ mount device label logical-name
```

The *device* parameter specifies the device on which the volume is to be mounted. The parameter can name a specific device, or it can specify a device group, as in the ALLOCATE command. The *label* parameter specifies the label of the volume being mounted. This parameter is only required in some circumstances, described below. The *logical-name* parameter specifies a logical name, which is defined to refer to the device on which the volume is ultimately mounted. All future references to the device should be made through this logical name.

The MOUNT command accepts a number of qualifiers, some of which are described in Table 19.2. You should familiarize yourself with the qualifiers before reading further.

Table 19.2 MOUNT Command Qualifiers

Qualifier	Default?	Description
/ASSIST	✓	Operator intervention is requested if a failure occurs during the mount operation.
/NOASSIST		The user must intervene if a failure occurs.
/FOREIGN		The volume is not in standard ANSI format. No volume label or header checking is performed. Tapes used by the BACKUP utility are mounted foreign.
/GROUP		The volume is made available to other users in the same UIC group. This qualifier requires GRPNAM privilege.
/LABEL	✓	The volume is in standard ANSI format. Volume label and header checking are performed.
/OVERRIDE		This qualifier accepts keyword values, which specify steps in the mount sequence that are to be bypassed. The keyword IDENTIFICATION specifies that volume label checking is not performed. In this case, no label needs to be specified on the MOUNT command.
/SHARE		The disk volume is made shareable. The device is not allocated by the current process.
/SYSTEM		The volume is made public; all system users can access it. This qualifier requires SYSNAM privilege.

The MOUNT command performs the following steps:

1. If a specific device is named, the command ensures that the device is unallocated or already allocated to the current process. If a device group is specified, the command locates an available device in that group. (The command does not search for a device containing the volume with the requested label; it merely locates one that is unallocated.)
2. If the /GROUP, /SHARE, or /SYSTEM qualifiers are specified, the command deallocates the device if necessary. Otherwise it is allocated to the current process.
3. The command ensures that a volume is physically loaded on the device.
4. Unless the /FOREIGN or /OVERRIDE=IDENTIFICATION qualifiers are specified, the command checks the volume label against the label specified on the MOUNT command.
5. The command prepares the volume for input/output operations.

6. The command defines a logical name to refer to the mounted device. The logical name is defined in the process logical name table by default, the group table if /GROUP is specified, or the system table if /SYSTEM is specified. The logical name is chosen as follows:
 - If a logical name is specified in the MOUNT command, it is used.
 - If a logical name is not specified, the name DISK\$*label* or TAPE\$*label* is used as appropriate. The *label* portion of the name is identical to the volume label.

The MOUNT command has many more options than are described here. The *VMS Mount Utility Manual* describes the command in complete detail.

Chapter 20



Processes

The VMS environment in which DCL commands are executed and programs are run is called the **process**. A process provides the structure and information necessary to run the program images that make up DCL, VMS utilities, and application software. A component of VMS called the **scheduler** is responsible for periodically choosing a process to run on each VAX processor. The scheduler must ensure that all processes get their fair share of the VAX execution cycles. The following components are associated with every VMS process:

- A set of data structures containing control information for the process. These structures are maintained by VMS in an area of system memory called pool. Some of the control information is described in the next section.
- A private area of virtual memory called **P0 space**. A program image invoked with the RUN command or directly with its own command (e.g., PHONE) is executed in P0 space.
- A private area of virtual memory called **P1 space**. When a process is being controlled by the DCL command interpreter, the DCL program image runs in P1 space. Symbols and process logical names are maintained in P1 space. Information about executing command procedures is also maintained in P1 space.

VMS supports various kinds of processes, four of which you are likely to encounter in your adventures with VMS:

Interactive. An **interactive process** is created when you log in to VMS at a terminal. The DCL command interpreter resides in P1 space and reads commands from the terminal. VMS treats an interactive process as an autonomous entity, which remains in existence until you log out.

Batch. A **batch process** is created when you submit a batch job to a job queue with the SUBMIT command. The batch queues and jobs are controlled by the VMS job controller.

▷ Ch. 21

Network. A **network process** is created when a process on another DECnet node attempts to communicate with the local node. This communication may result from copying a file to or from the local node, sending mail to a user, or calling the user with the PHONE utility. A network process is also created when a remote user logs in to the local node with the SET HOST command. Network processes are not discussed in this book.

Subprocess. A **subprocess** is a separate VMS process, which is created and owned by another process. Subprocesses are described in this chapter.

Interactive, batch, and network processes are collectively called **detached processes** because they are not attached to or owned by any other process. A subprocess is not a detached process because it is owned by some other process.

20.1

Information about the Current Process

VMS maintains a plethora of information about each process. The information is used to identify the process and control its execution. Much of the important information can be obtained from within a DCL procedure using one of two techniques.

The primary technique for obtaining information about the current process is the F\$GETJPI lexical function. The acronym GETJPI stands for “get job and process information.” The lexical function requires two arguments. The first is a process identifier, which for the the current process is the null string. The second is a keyword string specifying the desired item of information. Table 20.1 describes some of the commonly used process information items. The F\$GETJPI function returns the current value of the requested item, either as an integer or a character string. The following code displays the user name, billing account, and login time for the current process:

Table 20.1 F\$GETJPI Items

Keyword	Type	Description of Result
"ACCOUNT"	String	The billing account, padded with spaces to eight characters. The billing account is stored in the user's record in the user authorization file (UAF).
"AUTHPRIV"	String	A list of the authorized privileges of the process. These are stored in the user's UAF record.
"CPUTIM"	Integer	The CPU time used by the process, in hundredths of a second.
"CURPRIV"	String	A list of the current privileges of the process.
"GRP"	Integer	The group number of the UIC. This is stored in the user's UAF record.
"LOGINTIM"	String	The date and time the process was created.
"MEM"	Integer	The member number of the UIC. This is stored in the user's UAF record.
"MODE"	String	The process mode: "INTERACTIVE", "BATCH", "NETWORK", or "OTHER".
"PRCNAM"	String	The process name.
"TERMINAL"	String	The terminal at which an interactive user logged in.
"USERNAME"	String	The user name, padded with spaces to 12 characters. This is the name used to log in to VMS and is the key to the user's UAF record.

```

$ display "User Name: ", f$getjpi("", "USERNAME")
$ display "Account:   ", f$getjpi("", "ACCOUNT")
$ display "Login Time: ", f$getjpi("", "LOGINTIM")

```

One of the items that F\$GETJPI can return is the **process mode**. The mode is a keyword string: "INTERACTIVE" for an interactive process and its subprocesses; "BATCH" for a batch process and its subprocesses; "NETWORK" for a network process and its subprocesses; "OTHER" for all other kinds of processes. The mode can be used in a procedure to determine, for example, whether an interactive user is available to answer questions:

Table 20.2 F\$ENVIRONMENT Items

Keyword	Type	Description of Result
"CAPTIVE"	Boolean	True if the process was created by logging in to a captive user account.
"DEFAULT"	String	The working device and directory, as established with the SET DEFAULT command.
"MESSAGE"	String	The current message settings, as established with the SET MESSAGE command.
"PROCEDURE"	String	The full file spec of the executing command procedure.
"VERIFY_IMAGE"	Boolean	True if the SET VERIFY=IMAGE command is in effect.
"VERIFY_PROCEDURE"	Boolean	True if the SET VERIFY=PROCEDURE command is in effect.

```

$   interactive = f$getjpi("", "MODE") .eqs. "INTERACTIVE"
$   .
$   .
$   if interactive
$   then
$       .
$       . ask some questions
$       .
$   endif

```

The F\$GETJPI function obtains the mode of the process and the assignment command sets the symbol INTERACTIVE to true if the mode is interactive, false otherwise. The symbol is tested later in the procedure to determine whether to ask questions of the interactive user.

Three items of information available with F\$GETJPI can also be obtained with other lexical functions. The process mode can be obtained with F\$MODE, the name with F\$PROCESS, and the UIC with F\$USER. These functions were available before F\$GETJPI was introduced into DCL. The author suggests that you use F\$GETJPI rather than the three other lexical functions, both for consistency and because the names F\$PROCESS and F\$USER are misleading.

Additional process information, mostly related to the DCL environment, can be obtained with the F\$ENVIRONMENT lexical function. This function takes one argument, a keyword string specifying the desired item. It returns the current value of the item as an integer or character string. Table 20.2 describes some

important items of environment information, most of which have already been described in previous chapters. The following example uses F\$ENVIRONMENT:

```
$ this_proc = f$environment("PROCEDURE")
$ proc_dir = f$parse(this_proc,,,"DEVICE") + -
            f$parse(this_proc,,,"DIRECTORY")
$ define xda_system 'proc_dir
  .
  .
  .
$ @xda_system:xda_another-proc
```

The F\$ENVIRONMENT function obtains the file spec of the executing procedure. The logical name XDA_SYSTEM is defined to refer to the device and directory containing the procedure. This logical name can then be used to refer to any file in the procedure directory, such as another procedure that must be invoked by the current one.

20.2

Information about Other Processes

The F\$GETJPI lexical function can be used to obtain information about processes other than the current one. Its first argument is a **process identifier**, or PID. A PID is an integer that uniquely identifies a process from among all the processes on the system. In fact, on a VAXcluster, the PID uniquely identifies a process from among all processes on all nodes of the cluster. VMS assigns a PID during the creation of the process.

If you know the PID of a process, you can obtain information about the process. However, it is rarely the case that a procedure can determine the PID of some arbitrary process. It could attempt to ask the user for one, but the user probably doesn't know either. Instead of trying to determine the PID of a specific process, the procedure uses the F\$PID lexical function to obtain information about *every* process, perhaps selecting some subset of the processes to display or alter.

The F\$PID function requires a single argument, called the **context symbol**. The context symbol must be initialized to the null string before the first call. On the first call, F\$PID returns the PID of the first process in the system's process table. The PID is returned as a *string* containing the external representation of the integer identifying the process (e.g., "%X02000024"). The fact that it is returned as a string is a holdover from the time when DCL represented integers as character strings. The F\$PID function also updates the context symbol, so on the next call it returns the PID of the second process in the table. This continues until there are no more processes, at which point the function returns the null string.

The `F$PID` function returns the identifiers of some or all of the processes on the system, depending upon the privileges of the requesting process. If the process has `WORLD` privilege, every PID is returned. If it has `GROUP` but not `WORLD` privilege, only the PIDs of processes in the same group are returned. If it has neither, then only the PIDs of processes with the same UIC are returned.

The `F$GETJPI` function expects a PID as its first argument (as a special case, it accepts the null string to identify the current process). The function can accept either an integer or a string as the PID, which allows the string returned by `F$PID` to be used as the first argument to `F$GETJPI`. In this way, `F$GETJPI` can obtain information about a process whose identifier was returned by `F$PID`.

Suppose you are writing a procedure to shut down the system. As part of the shutdown sequence, the procedure lists the processes that still exist on the system. However, it does not want to list system processes, those in group 0 or 1. The following code accomplishes this task:

```
$    display "Non-system processes:"
$    process_count = 0
$    context = ""
$10:  pid = f$pid(context)
$    if pid .eqs. "" then goto 19
$    if f$getjpi(pid,"GRP") .le. 1 then goto 10
$    display f$faos("!16AS (!AS)", f$getjpi(pid,"PRCNAM"), -
$                                     f$getjpi(pid,"MODE"))
$    process_count = process_count + 1
$    goto 10
$19:
$    if process_count .eq. 0 then -
$        display "There are no such processes."
$    if process_count .ne. 0 then -
$        display "Process count: ", process_count
```

When this example is run, it produces output something like the following:

Table 20.3 Commands to Set Process Environment

Environment Item	Command	Chapter
Default directory	SET DEFAULT	7
Message settings	SET MESSAGE	7
Privileges	SET PROCESS/PRIVILEGES= <i>list</i>	20
Process name	SET PROCESS/NAME=" <i>name</i> "	20
User identification code (UIC)	SET UIC	7
Verification	SET VERIFY	12

Non-system processes:

```
VXCAPTIVE1_1 (INTERACTIVE)
OSTERAAS (INTERACTIVE)
OSTERAAS_1 (INTERACTIVE)
BATCH_1922 (BATCH)
LES_1 (INTERACTIVE)
AMC_MAINT (INTERACTIVE)
VXCAPTIVE1 (INTERACTIVE)
GREEK (INTERACTIVE)
LES (INTERACTIVE)
BASE (INTERACTIVE)
BASE_1 (INTERACTIVE)
```

Process count: 11

The F\$PID and F\$GETJPI functions provide a powerful mechanism for determining the status of any processes on a VAX system or cluster. There is no way to obtain the DCL environment information for a process other than the current one (the F\$ENVIRONMENT function does not accept a PID).

20.3

Setting Process Information

VMS provides various commands to establish or alter the current process environment. These commands are listed in Table 20.3, along with the chapters in which they are described. All but two have been described in previous chapters.

The SET PROCESS/PRIVILEGES command can be used to set the current process privileges to a specified list of privileges. This command is rarely used in procedures because it does not provide a straightforward way to save the user's privilege settings, alter them, and then restore them to the original settings. The F\$SETPRV lexical function is designed to serve exactly this purpose. It should be used in place of the SET command to establish the privileges for a procedure.

▷ Ch. 7

The process name can be altered with the `SET PROCESS/NAME` command. The name is composed of any characters, including lowercase letters, but is restricted to 15 characters in length. VMS chooses a process name for each process as it is created. In the case of interactive processes, the process name is the same as the user name for the user's first process and the same as the terminal name for subsequent processes. In the case of batch processes, the process name consists of the word `BATCH` with a numeric suffix.

20.4

Subprocesses

A subprocess is a separate VMS process, created by an existing process and owned by that process. The owner is called the **parent** and the subprocess is called the **child**. A process that is not a subprocess is a detached process: the interactive process created when you log in is a detached process.

A DCL procedure creates a subprocess using the `SPAWN` command. The `SPAWN` command is complex, and you probably will become comfortable with it only after some experimentation. The following paragraphs describe the command in detail. Table 20.4 lists many of the qualifiers accepted by `SPAWN`. A process must have the `TMPMBX` or `PRMMBX` privilege in order to use the `SPAWN` command.

When a subprocess is created with the `SPAWN` command, it inherits some of the environment of the parent process. In particular:

- The child always inherits the parent's current process privileges.
- The child always inherits the working disk and directory.
- The child always inherits the message settings established with the DCL command `SET MESSAGE`.
- The child can inherit the parent's global, prompt-level, and procedure-level symbols, as controlled by the `/SYMBOLS` qualifier. The child never inherits the `$RESTART`, `$SEVERITY`, or `$STATUS` symbols.
- The child can inherit the process logical names and name tables, as controlled by the `/LOGICAL_NAMES` qualifier.

It is also important to note the environment items that are *not* inherited by the subprocess:

- The child never inherits the standard process-permanent files `SYS$COMMAND`, `SYS$INPUT`, `SYS$OUTPUT`, and `SYS$ERROR`. These files are established separately for the subprocess.

▷ Ch. 21

▷ Ch. 8

Table 20.4 SPAWN Command Qualifiers

Qualifier	Default?	Description
<i>/INPUT=file-spec</i>		Specifies a procedure file containing commands to be executed by the subprocess. If neither this qualifier nor a command string are included on the SPAWN command, input is taken from the terminal.
<i>/LOGICAL_NAMES</i>	✓	The child inherits the parent's logical names.
<i>/NOLOGICAL_NAMES</i>		The child does not inherit the parent's logical names.
<i>/OUTPUT=file-spec</i>		Specifies a file that receives the output from the subprocess. If not included, output is directed to the same destination as the parent's SYS\$OUTPUT.
<i>/PROCESS=name</i>		Specifies a name for the subprocess. If not specified, the name consists of the parent's name with an integer suffix.
<i>/SYMBOLS</i>	✓	The child inherits the parent's global, prompt-level, and procedure-level symbols.
<i>/NOSYMBOLS</i>		The child does not inherit the parent's symbols.
<i>/WAIT</i>	✓	The parent suspends execution until the child terminates.
<i>/NOWAIT</i>		The parent and child execute simultaneously.

- The child never inherits the parent's current DCL command table. It always uses the standard table unless changed by the subprocess itself.
- The child does not execute a LOGIN.COM procedure.

The SPAWN command creates a subprocess with the described environment. The child runs independently of the parent, executing commands determined by the parameters and qualifiers to the SPAWN command:

- The SPAWN command accepts an optional command string consisting of a single DCL command to be executed by the child. If the command string is present, the command is executed immediately after the child process is created and initialized.
- The SPAWN command accepts an optional */INPUT* qualifier, which specifies a procedure file. The procedure file becomes the child's process-permanent SYS\$INPUT file, thus causing the child to execute the procedure.

- If both the command string and the `/INPUT` qualifier are present, the command string is executed first, then the input procedure.
- If neither are present, the child reads commands from the terminal. In the case of a subprocess spawned from a batch job, however, the process terminates immediately, having no source of commands to execute.

Subprocess input and output is described in more detail in Section 20.5.

The subprocess executes its commands independently of the parent. The action taken by the parent during the execution of the child depends on the `/WAIT` qualifier. If `/WAIT` is specified on the `SPAWN` command (it is the default), then the parent ceases execution until the child is done. If `/NOWAIT` is specified, the parent and child execute simultaneously. When the `SPAWN` command is entered at the DCL prompt, both forms of the qualifier are useful. You may want to use the `/WAIT` qualifier to suspend the main process while you do some other work in a subprocess. Or you may want to use the `/NOWAIT` qualifier to execute a time-consuming command in a subprocess while continuing to edit a file in the main process. On the other hand, when the `SPAWN` command is used in a procedure, the `/NOWAIT` qualifier is almost always used so that the two processes can execute simultaneously.

The use of the `/NOWAIT` qualifier requires some synchronization between the parent and child processes; one method of synchronization is described in Sections 20.6 and 20.7.

20.4.1

Examples

▷ Ch. 9

The `SPAWN` command is one of the DCL commands that can be executed at the temporary command level created by a `CTRL/Y` interrupt. When you press `<CTRL/y>` during the execution of a program, a temporary command level is created by DCL. The `SPAWN` command can be entered at this level in order to create a subprocess. The subprocess can execute arbitrary DCL commands without affecting the parent process. When you log out of the subprocess and enter the `CONTINUE` command at the temporary command level, the command level is canceled and the original program resumes execution. This technique allows you to interrupt a lengthy program, perform some other activities (e.g., read a new mail message), and then resume the program.

Here is an example of a temporary command level:

```

$ search xda_system:*.com error
.
. output from search
.
Interrupt
$ spawn
%DCL-S-SPAWNED, process GREEK_1 spawned
%DCL-S-ATTACHED, terminal now attached to process GREEK_1
$ mail
.
. read your mail
.
MAIL> exit
$ logout
Process GREEK_1 logged out at 3-AUG-1988 14:21:47.47
%DCL-S-RETURNED, control returned to process GREEK
$ continue
.
. more output from search
.

```

During the search operation, the user presses <CTRL/y> to create a temporary command level. The SPAWN command creates a subprocess in which the MAIL utility can be used. The subprocess is ultimately terminated with the LOGOUT command, and the search operation is resumed with the CONTINUE command.

The /NOWAIT qualifier can be used on a SPAWN command entered at the DCL prompt in order to initiate the simultaneous execution of a time-consuming command:

```

$ spawn/nowait/output=error.sea search xda_system:*.com error

```

In this case, the same search operation is performed in a subprocess executing at the same time as the parent process. The /OUTPUT qualifier is used to direct the subprocess output to a file so that the output does not clutter the terminal screen and also so that it is collected in one place for future reference.

The SPAWN command can be used in a procedure to create a parallel flow of execution when an application needs to do two things at once. Section 20.7 presents a complete example of this technique. Here are a few examples of just the SPAWN command:

```

$! Create a subprocess to execute the background sampling
$! procedure. Don't let it inherit any symbols.
$
$ spawn/nowait/nosymbols @xda_system:xda_sampler

-or-

$! The same thing can be accomplished using the /INPUT qualifier,
$! but it really isn't as clear to the reader.
$
$ spawn/nowait/nosymbols /input=xda_system:xda_sampler

-or-

$! In particular, if the procedure requires any parameters, they
$! can only be passed using the first form of the command:
$
$ spawn/nowait/nosymbols @xda_system:xda_sampler sys$sysdevice -
1:00:00

```

20.4.2 *Inheriting Symbols and Logical Names*

When a subprocess is created with the SPAWN command, the subprocess normally inherits the symbols and logical names of its parent. The /NOSYMBOLS qualifier prevents symbols from being inherited, while the /NOLOGICAL_NAMES qualifier prevents logical names from being inherited.

Symbol inheritance should be avoided if at all possible. A procedure should not pass information to a subprocess by creating symbols with the information and then allowing the subprocess to inherit the symbols. This implicit method of passing information to the subprocess is not at all obvious to a reader of the procedure and may be difficult to maintain in the future. Instead, the parent procedure can pass information to the child procedure in the form of procedure parameters. Another reason to avoid symbol inheritance is that it takes significant time during subprocess creation to pass the symbols from the parent to the child.

Logical name inheritance is also dangerous and inefficient, although it is sometimes the case that the subprocess must rely on logical names defined by the parent. If at all possible, use the /NOLOGICAL_NAMES qualifier to prevent logical name inheritance and pass the necessary information as procedure parameters. Section 20.7 contains an example of passing parameters to a subprocess.

20.4.3

Subprocess Termination

A subprocess created with the SPAWN command terminates when it has no more commands to execute. There are three possibilities:

- When the SPAWN command includes a command to execute but does not include the /INPUT qualifier, the subprocess terminates after executing the single command. Note, however, that the command can be the at-sign command to invoke a procedure in the subprocess; in this case, the entire procedure executes.
- When the SPAWN command includes the /INPUT qualifier, the subprocess terminates when the specified command procedure exits.
- When neither a command or the /INPUT qualifier are included, the subprocess must be terminated with the LOGOUT command. In the case of a batch process, however, the spawned subprocess terminates immediately, because it has no source of commands.

The action taken by the parent process during the lifetime of the subprocess depends upon the /WAIT qualifier. If /WAIT was specified, the parent process suspends execution until the subprocess terminates. If /NOWAIT was specified, both processes execute simultaneously with no automatic synchronization.

20.5

Process-Permanent Files

A **process-permanent file** is a file that is opened by a process and that remains open until either it is explicitly closed or the process terminates. There are two common uses for process-permanent files (PPFs). First, the four standard logical names SYS\$COMMAND, SYS\$INPUT, SYS\$OUTPUT, and SYS\$error, which define the inputs and outputs for a process, are PPFs. Second, any file opened with the OPEN command is a PPF.

The four standard files listed above are automatically opened by VMS during the creation of a process. They remain open until the process terminates. The sources for SYS\$COMMAND and SYS\$INPUT and the destination of SYS\$OUTPUT and SYS\$error are determined by VMS using some standard rules along with optional information given by the creator of the process. The sources and destinations for these logical names are not always obvious, particularly in the case of a subprocess spawned from within a DCL procedure. Table 20.5 lists the PPFs for a procedure running in an interactive process. Table 20.6 lists the PPFs for a procedure running in a batch process. Remember that these tables pertain only to a process running a DCL procedure.

Table 20.5 Standard PPFs: Interactive Procedure

PPF	Detached Process	Subprocess
SYS\$COMMAND	The terminal.	(1) The terminal; or (2) the command procedure specified with the /INPUT qualifier on the SPAWN command.
SYS\$INPUT	The command procedure executing in the detached process.	The command procedure executing in the subprocess.
SYS\$OUTPUT	(1) The terminal; or (2) the file specified with the /OUTPUT qualifier on the at-sign command that invoked the procedure.	(1) the same as the parent's destination; or (2) the file specified with the /OUTPUT qualifier on the SPAWN command.
SYS\$ERROR	The terminal.	Same as SYS\$OUTPUT.

When a process-permanent file is specified as a source or destination for a program, the program performs input/output operations on the file that is already open. In particular, when a PPF is specified as an output file, data is appended to the existing file. No new file is created.

20.6

Jobs

VMS uses the term **job** to refer to a detached process and all of its subprocesses. The detached process can be an interactive, batch, or network process; the word *job* has nothing to do with the traditional term *batch job*. When a procedure uses the SPAWN command to create a subprocess, both the parent and the child are members of the same job.

Associated with each job is a **job logical name table**. Logical names defined in this table can be accessed by any process in the same job as the defining process. Thus, if a parent process defines a job logical name, any children created with SPAWN can access the logical name, and vice versa. The following command defines the job logical name XDA_JOB_INFO:

Table 20.6 Standard PPFs: Batch Procedure

PPF	Detached Process	Subprocess
SYS\$COMMAND	The command procedure executing in the detached process.	(1) The null device (NL:); or (2) the command procedure specified with the /INPUT qualifier on the SPAWN command.
SYS\$INPUT	Same as SYS\$COMMAND.	The command procedure executing in the subprocess.
SYS\$OUTPUT	(1) The log file; (2) the null device (NL:) if no log file was requested; or (3) the file specified with the /OUTPUT qualifier on the at-sign command that invoked the procedure (subprocedures only).	(1) the same as the parent's destination; or (2) the file specified by the /OUTPUT qualifier on the SPAWN command.
SYS\$ERROR	(1) The log file; or (2) the null device (NL:) if no log file was requested.	Same as SYS\$OUTPUT.

```
$ define/job xda_job_info step_1
```

The job logical name table is deleted when the job's detached process is deleted.

Job logical names can be used to communicate between parent and child subprocesses. The following section presents an example of the SPAWN command used in conjunction with the job logical name table.

20.7

An Example with SPAWN

Suppose you have a procedure that creates and deletes many files on a particular disk. It might be interesting to monitor the minimum number of free blocks on the disk as the procedure runs. A subprocess can be used to "watch" the disk as the parent process executes. At the end of the file manipulation, the subprocess can report the free block count to the parent. Note that the free block count might be affected by other users on the system.

Here is the command procedure that is run in the subprocess:

```

$! This is the XDA_WATCH-DISK procedure.
$! The first parameter is the disk to be watched.
$
$ min_free = 999999999
$10: if f$strnlm("xda_watch_disk_blocks") .eqs. "" then goto 19
$    wait 00:00:00.50
$    f = f$getdvi(p1,"FREEBLOCKS")
$    if f .lt. min_free then min_free = f
$    goto 10
$19:
$
$ define/job xda_watch_disk_blocks 'min_free
$ exit

```

Here is part of a DCL application that uses the disk watcher:

```

$ define/job/nolog xda_watch_disk_blocks in-progress
$ spawn/nosymbols/nological_names/nowait -
    @xda_system:xda_watch-disk 'disk
.
. perform file manipulation
.
$ undefine/job xda_watch_disk_blocks
$10: if f$strnlm("xda_watch_disk_blocks") .eqs. "" then goto 10
$ display "Minimum free blocks: ", -
    f$strnlm("xda_watch_disk_blocks")
$ undefine/job xda_watch_disk_blocks

```

The job logical name XDA_WATCH_DISK_BLOCKS provides the communication and synchronization between the parent process and the subprocess watching the disk. The following sequence of events occurs when this application runs:

1. The parent process defines the logical name XDA_WATCH_DISK_BLOCKS in the job logical name table, assigning it an arbitrary value such as the word IN-PROGRESS.
2. The parent spawns a subprocess. There is no reason for the child to inherit the symbols or logical names of the parent, so the /NOSYMBOLS and /NOLOGICAL_NAMES qualifiers are used. The /NOWAIT qualifier allows the parent and child to run simultaneously.
3. The parent begins to perform its file manipulation activities.
4. The XDA_WATCH-DISK procedure running in the child repeatedly performs the following steps:

- a. It checks to see whether the parent has deleted the job logical name. If so, this is the signal to stop watching the disk, report the result to the parent, and exit.
 - b. It waits one-half second.
 - c. It determines the number of free blocks on the disk and resets `MIN_FREE` if a new low has been attained.
5. Eventually the parent deletes the job logical name. The subprocess ceases executing the disk-watching loop and redefines the job logical name so that its value is the minimum number of free blocks. The procedure then exits and the child is deleted.
 6. After the parent deletes the job logical name, it must wait until the child redefines it. This is done with a simple one-line loop, which just iterates until the logical name is again defined.
 7. The parent determines the minimum free blocks by translating the logical name.
 8. The parent deletes the logical name so that it does not clutter the job logical name table.

Chapter 21



Batch Jobs

This chapter describes the batch facility of VMS, which allows a process to run “unattended,” without being connected to an interactive terminal. DCL applications can take advantage of the batch facility to perform time-consuming tasks without tying up the user’s terminal. It is important for the DCL programmer to understand the batch environment in order to coordinate the interactive and batch portions of an application and ensure their consistency.

21.1 The Batch Environment

A **batch job** is a process that runs in the VMS batch environment rather than in the interactive environment. The batch-processing facility is supported by a special program called the **job controller**, which runs in its own VMS process. The job controller is responsible for maintaining a set of **job queues** to which users submit batch and print jobs for processing. The remainder of this chapter discusses techniques for using the job controller and queues to run batch jobs.

A given VMS system can have one or more batch queues, each with its own unique name. Associated with a queue is a set of attributes, which determine how the job controller selects and runs jobs from the queue. It is up to your system manager to set up the batch queues and define their attributes; this book does not discuss the creation of queues. Almost every VMS system has a standard batch queue named SYS\$BATCH.

When you submit a batch job for execution, what you are submitting is a DCL procedure that the job controller is to execute in a batch process. A batch process is similar to the interactive process described in this book, but there are important differences. Once you understand the differences, you can write DCL applications that run both interactively and in batch. The following points describe the differences between a batch process and an interactive one:

- There is no terminal associated with a batch process. Its input comes from a command procedure and its output goes to a **log file**.
- The logical names `SYSD$COMMAND` and `SYSD$INPUT` refer to the DCL procedure being run in the batch process.
- The logical names `SYSD$OUTPUT` and `SYSD$ERROR` are directed to a log file. A log file is a text file whose purpose is to contain all the output generated by the batch job. The log file can be retained after the job completes.
- The `F$GETJPI` lexical function can be used to determine whether a procedure is being run interactively or in batch. When called with the item keyword "MODE", the function returns the string "INTERACTIVE" or "BATCH", depending upon the current environment.

▷ Ch. 20

A fundamental consequence of the batch environment is that a procedure cannot ask questions of the interactive user. Instead, any information the procedure requires to operate must be passed to it in the form of parameters.

One important similarity between the batch and interactive environments is the login procedure. If the procedure `LOGIN.COM` exists in your login directory, VMS automatically invokes it when you log in. The same is true when a batch job is submitted: VMS automatically invokes your `LOGIN.COM` immediately after it creates the batch process. If parts of the login procedure should be skipped in batch, use the `F$GETJPI` function to check for batch and exclude those parts.

21.2 Submitting a Batch Job

A batch job is submitted to a batch queue using the `SUBMIT` command. The `SUBMIT` command accepts many qualifiers, which control how and when the job is executed. The basic form of the command is as follows:

Table 21.1 SUBMIT Command Qualifiers (Part 1)

Qualifier	Default?	Description
<i>/AFTER=time</i>		The job is held in the queue and not run until the specified time.
<i>/DELETE</i>		The command procedure is deleted after the job finishes.
<i>/NODELETE</i>	✓	The command procedure is not deleted.
<i>/HOLD</i>		The job is held in the queue until it is explicitly released by the SET QUEUE/ENTRY command.
<i>/NOHOLD</i>	✓	The job is not held.
<i>/IDENTIFY</i>	✓	The job name, queue, and entry number are displayed after the job is submitted.
<i>/NOIDENTIFY</i>		The job information is not displayed.
<i>/KEEP</i>	(depends)	The log file is retained after the job completes. This is the default if <i>/NOPRINTER</i> is specified.
<i>/NOKEEP</i>	(depends)	The log file is deleted after the job completes. This is the default if <i>/PRINTER</i> is specified.
<i>/LOG_FILE=spec</i>	✓	Specifies the file to contain the job log. The default device/directory is the submitter's login directory. The default file name is the name of the job. The default file type is LOG.
<i>/NOLOG_FILE</i>		No job log file is produced.

\$ submit *procedure-spec*

The *procedure-spec* is a file spec that names the DCL procedure to be run in batch. The procedure must exist at the time the SUBMIT command is entered. Tables 21.1 and 21.2 describe many of the important qualifiers for the SUBMIT command.

Here are a few things to note about the SUBMIT command:

- Unless you specify the */AFTER* qualifier, the job is run as soon as possible, depending on the jobs queued before it.
- By default, a log file is generated, printed to the SYS\$PRINT queue, and deleted. It is usually easier to review the log file with an editor. This is specified by the following combination of qualifiers:

Table 21.2 SUBMIT Command Qualifiers (Part 2)

Qualifier	Default?	Description
<code>/NAME="name"</code>		Specifies the name of the job. The name can contain any characters but is limited to 39 characters in length. The default name is the name of the command procedure.
<code>/NOTIFY</code>		A message is broadcast to your terminal when the job completes.
<code>/NONOTIFY</code>	✓	No completion message is broadcast.
<code>/PARAMETERS</code>		Provides up to eight parameters to the command procedure. The parameters are separated by commas and enclosed in parentheses. If a parameter contains any characters other than letter or digits, it must be enclosed in double quotes (").
<code>/PRINTER=queue</code>	✓	The log file is queued for printing. The default print queue is <code>SYS\$PRINT</code> . <i>Note that this qualifier is present by default.</i>
<code>/NOPRINTER</code>		The log file is not printed.
<code>/QUEUE=queue</code>		Identifies the batch queue on which the job is queued. The default queue is <code>SYS\$BATCH</code> .
<code>/RESTART</code>		Specifies that the batch job is restartable (see Section 21.4).
<code>/USER=user</code>		Names the user on whose behalf the job is run. The default is the submitting user. The <code>CMKRNL</code> privilege is required to submit a job for another user.

```
$ submit /log_file=file-spec/keep/noprinter procedure
```

▷ Ch. 15

If the log file spec does not include a directory, the log file is placed in your login directory. The log file is in VFC (variable with fixed control) format, not in standard text format.

- If the `/LOG_FILE` qualifier does not include a log file name, the name of the command procedure is used. However, if the `/NAME` qualifier is present, then the job name is used. In the latter case, the job name must be a valid file name.
- To use the `/USER` qualifier to submit a job on behalf of another user, you must have `CMKRNL` (change mode to kernel) privilege and read access to the user authorization file.

The SUBMIT command is used by a DCL application to run a portion of an application in batch. For example, an application that generates accounting reports might do so in batch because the ACCOUNTING utility takes a long time to format reports from large accounting files. The application could use the following SUBMIT command:

```
$    submit /name=xda_accounting_reports -  
        /log_file=xda_logs:xda_accounting_reports.log -  
        /keep/noprinter -  
        xda_system:xda_accounting /parameters=(all,summary)
```

21.3

Sending Mail about Job Status

The success or failure of a batch job can always be determined by saving a log file and checking it after the job completes. However, because a batch log is nothing more than a snapshot of the output produced by the job, it can be difficult for a naive user to understand. In addition, it may contain a lot of extraneous information that is not of direct concern to the user. Another way to summarize the status of a batch job is for the job to produce a succinct log of important events and mail the log to the user who submitted the job. This status log can be customized for the task being performed, using terminology and summarizing events that the user understands.

A status log is relatively easy to produce. The command procedure creates a temporary file at the beginning of the job. As it proceeds, it writes summary information to the file. When the procedure is finished, it closes the file, mails it to the submitter, and deletes it. The following command procedure is designed to run in batch. It accepts a wildcard file spec, which identifies a set of files whose internal structure is to be checked with the ANALYZE utility. The output produced by each analysis is collected in a status log and mailed to the user.

```

$! @XDA_ANALYZE file-spec
$
$ analyze = "analyze"
$ delete = "delete"
$ mail = "mail"
$ xda__status = %x10428000
$ xda__success = xda__status + %x0001
$ xda__ctrly = xda__status + %x000c
$ status = xda__success
$ log_file = ""
$ on control_y then goto control_y
$ on warning then goto error
$
$ libcall = "@xda_system:subroutine-library"
$ libcall unique_name xda_sys$scratch:xda_?.log;
$ log_file = xda_
$ open/write xda_log 'log_file
$ write xda_log ""
$ write xda_log "Analysis performed with ANALYZE/RMS_FILE/CHECK."
$
$10: file = f$search(p1)
$ if file .eqs. "" then goto 19
$ set noon
$ analyze/rms_file/check/output=xda_log 'file
$ set on
$ goto 10
$19:
$
$ close xda_log
$ set noon
$ mail 'log_file 'f$getjpi("", "USERNAME") -
    /subject="Batch analysis of 'p1'"
$ set on
$ delete 'log_file'*
$
$
$control_y:
$ status = xda__ctrly
$ goto exit
$error:
$ status = $status
$ goto exit
$exit:
$ set noon
$ close/nolog xda_log
$ if log_file .nes. "" then -
    if f$search(log_file) .nes. "" then delete 'log_file'*
$ exit status .or. %x10000000

```

The status log is opened with the logical name XDA_LOG. Any number of WRITE commands can be used to add lines to the status log. In addition, the logical name can be specified in the /OUTPUT qualifier of a command so that the output produced by a utility is directed to the status log. This is the case with the ANALYZE command in the procedure. The output from each use of the ANALYZE utility is appended to the status log.

This job is submitted using a command such as the following:

```
$ submit/nolog_file -  
      xda_system:xda_analyze /parameter=user_disk:[smith]*.dat
```

The SUBMIT command specifies that no log file is to be produced; the status log mailed by the job is sufficient. The /PARAMETER qualifier specifies the files that are to be analyzed.

21.4 **Restartable Batch Jobs**

In some applications, it is imperative that a batch job run to completion even if it is terminated prematurely. There are at least two reasons why a batch job can terminate prematurely without running its cleanup code. First, the VAX system might crash. Second, the operator or some other user might cancel the batch job. A batch job can be made **restartable** so that it will always run to completion.

The job controller guarantees to resubmit a restartable job if the job does not complete its execution. The job will be resubmitted if the system crashes while it is executing. The job will also be resubmitted if the operator cancels it, as long as it is canceled with the following command:

```
$ stop/queue/entry=number/requeue queue-name
```

The /REQUEUE qualifier requests that the job be restarted.

A batch job is made restartable by including the /RESTART qualifier on the SUBMIT command. This single qualifier is all that is required to ensure that the job is resubmitted if it does not complete the first time. However, if the batch job does terminate prematurely and restarts, it will run again from the beginning, repeating the code that was already executed during the original submission. This may be harmless, but in many cases it is not. The programmer must design the batch procedure to be restartable, skipping those portions of the code that have already been executed and that must not be reexecuted.

DCL includes some simple features to aid the programmer in designing restartable procedures. Two special symbols are involved:

\$RESTART. This symbol is always defined. It has a boolean value, false if this is the original execution of a batch job, true if this is a restarted job. The symbol does not specify whether the job is restartable, but rather whether it has in fact been restarted.

BATCH\$RESTART. This symbol is only defined during the execution of a restarted job. Its value is determined by the procedure during its *original execution*, as explained below.

A procedure can test the **\$RESTART** symbol to determine whether it has been restarted. If it has, it can use the value of **BATCH\$RESTART** to control which parts are executed and which are skipped. The value of **BATCH\$RESTART** is established with the **SET RESTART_VALUE** command. Upon restart, the symbol has the last value that was set during the original execution of the procedure. Using the **SET RESTART_VALUE** command, the procedure can be divided into steps as follows:

```
      . procedure initialization
      .
$step1:
$   set restart_value=step1
      . perform first step of procedure
      .
$
$step2:
$   set restart_value=step2
      . perform second step
      .
$
$step3:
$   set restart_value=step3
      . perform third step
      .
$   goto exit
$
$exit:
$   set restart_value=exit
      . cleanup code
      .
```

Each step begins with a **SET RESTART_VALUE** command, which saves the step name. The step name is included as a label at the beginning of the code that performs the step. When the procedure is about to exit, a **SET RESTART_VALUE**

command saves the label EXIT as an indication that the procedure is effectively complete. All you need to make this procedure restartable is some code to test the value of \$RESTART and to skip to the appropriate step:

```
.  
:procedure initialization  
.  
$   if $restart then  
$     if f$type(batch$restart) .nes. "" then goto 'batch$restart  
$  
$step1:  
$   set restart_value=step1  
.  
:perform first step of procedure  
.  
$  
$step2:  
$   set restart_value=step2  
.  
:perform second step  
.  
$  
$step3:  
$   set restart_value=step3  
.  
:perform third step  
.  
$   goto exit  
$  
$exit:  
$   set restart_value=exit  
.  
:cleanup code  
.
```

After procedure initialization, the value of \$RESTART is tested. If it is true, the procedure has been restarted. In this case, the F\$TYPE lexical function is used to determine whether BATCH\$RESTART has a value from a SET RESTART_VALUE command. If so, the procedure skips to that label, effectively bypassing the steps completed during the original execution of the batch job. If BATCH\$RESTART does not exist, then the original job did not even get to the first step, and execution takes up at the beginning. In the event that the procedure almost finished during its original execution, BATCH\$RESTART contains the label EXIT and execution skips immediately to the cleanup code and the procedure exits.

The only difficult aspect of designing restartable jobs is in breaking the procedure into steps. Here are a few guidelines:

- Do not assume that BATCH\$RESTART exists, because the first step might not be reached before the system crashes. Test BATCH\$RESTART with the F\$TYPE function to make sure it is defined.
- Each step must be restartable. When the system crashes in the middle of a step, the job is restarted at the beginning of that step. It must be harmless to reexecute the step from the beginning.
- If a step of the procedure cannot be rerun, then you must arrange for the procedure never to restart at the beginning of that step. Assume that step 3 cannot be rerun. Instead of performing a SET RESTART_VALUE=STEP3 at the beginning of step 3, you can perform a SET RESTART_VALUE=STEP4 so that step 3 is skipped in the event of a restart.
- After the last step, make sure to set BATCH\$RESTART to indicate that the procedure is effectively complete. Performing a SET RESTART_VALUE=EXIT command will ensure that the procedure immediately exits if it is restarted after the last step.

21.5 **Periodic Batch Jobs**

Some applications require that a batch job run periodically, perhaps once an hour or once a day. The VMS job controller does not provide a facility for easy implementation of periodic batch jobs, so the batch job must take the responsibility for resubmitting itself. The simplest technique is for the batch job to resubmit itself just before it exits:

```

$! Procedure to monitor system resources every 4 hours.
$
$! Remember the time at which the job started.
$
$ start_time = f$time()
.
. :additional initialization
.
$
.
. :procedure activities
.
$exit:
$ set noon
$ submit/nolog_file/after="''start_time+0-04:00" -
    'f$environment("PROCEDURE")
.
. :more cleanup
.
$ exit status .or. %x10000000

```

At the very beginning of the procedure, the current time is saved in the symbol `START_TIME`. The cleanup code includes a `SUBMIT` command to resubmit the job for execution four hours after the starting time. The `SUBMIT` command is included in the cleanup code so that it is guaranteed to be executed regardless of the reason for procedure exit. A combination time is specified on the `/AFTER` qualifier: it includes the job starting time as its absolute time component and the delta time “four hours from now” as its delta time component. The resulting absolute time is four hours from the original job starting time.

This job rescheduling technique is almost foolproof, but it has one flaw. If the batch job terminates before it has had a chance to resubmit itself, perhaps because of a system crash, the periodicity of the job is broken. In order to reduce the chances of this happening, you could reorganize the procedure to resubmit itself at the beginning rather than at the end. This may indeed significantly reduce the “window” in which termination prevents resubmission, but it does not eliminate it. The system might crash at the very beginning of the job. In addition, resubmitting the job at the beginning of the procedure opens up the possibility that the first job is not finished before the second one begins (what if the first job occasionally runs for more than four hours?).

You can guarantee that the batch job will resubmit itself by making it restartable. The technique used to make the procedure restartable is a variation on the scheme presented in the previous section. Here is the preceding procedure modified to guarantee that it is resubmitted:

```
#! Procedure to monitor system resources every 4 hours.
$
#! Remember the time at which the job started.
$
$ start_time = f$time()
$
#! If this is a restart and we already resubmitted the procedure,
#! exit immediately. Otherwise skip to the cleanup code and
#! resubmit.
$
$ if $restart
$ then
$   if f$type(batch$restart) .nes. "" then -
$     if batch$restart .eqs. "RESUBMITTED" then exit
$   goto exit ! To resubmit the job.
$ endif
```

```

        .additional initialization
    $
        .procedure activities
    $
$exit:
$   set noon
$   submit/nolog_file/after="'start_time+0-04:00'/restart -
        'f$environment("PROCEDURE")
$   set restart_value=resubmitted
        .more cleanup
    $   exit status .or. %x10000000

```

During the normal execution of this procedure, nothing special happens until after the job is resubmitted by the cleanup code. Once the job is resubmitted, the restart value is set to "RESUBMITTED" to indicate that the resubmittal has occurred. If the job then exits normally, the restart value is never employed. Note that the SUBMIT command includes the /RESTART qualifier so that the next batch job is also restartable.

If the system crashes while the procedure is executing, the batch job is restarted. Here is what happens:

1. The starting time is saved.
2. The procedure detects that it was restarted.
 - a. If the symbol BATCH\$RESTART exists and has the value "RESUBMITTED", then the original job got far enough to resubmit itself. There is nothing left to do, so the restarted job exits immediately.
 - b. If the symbol BATCH\$RESTART does not exist or has some other value, then the original job did not get far enough. The procedure skips to the exit code to resubmit itself.
3. Under no circumstances does the job execute its initialization code or procedure activities.

With the addition of only a few lines of code, the procedure can guarantee that the batch job is executed every four hours regardless of how each job terminates.

The previous section demonstrated a method to ensure that a periodic batch job is resubmitted exactly once every period. This does not rule out the possibility of duplicate jobs, however, because a user might accidentally submit the job while it is already active or queued for its next run. Or an operator might submit it after a system crash, not realizing that the job is restarted after crashes. If you want to ensure that only one copy of a batch job can ever run at one time, additional code is needed in the procedure.

The additional code is quite straightforward. It makes use of the group or system logical name table. The fundamental idea is that a particular group or system logical name will always be defined while the job is running. This logical name acts as a “lock” on the batch procedure. If another job detects that the logical name exists when it starts up, it immediately cancels itself.

The group logical name table can be used if the batch procedure is used only by the members of a particular UIC group. If anyone can submit the procedure, then the system logical name table must be used so that all submitters can access the logical name. The only reason to use a group logical name is that it requires only GRPNAM privilege, while use of a system logical name requires SYSNAM privilege.

Here is the periodic batch job from the previous section with the addition of a logical name lock:

```

$! Procedure to monitor system resources every 4 hours.
$
$! If this procedure is already running, exit immediately.
$! Otherwise define the system logical name to lock the
$! procedure.
$
$ if f$trnlrm("xda_batch_lock") .nes. "" then exit
$ define/system xda_batch_lock running ! Value doesn't matter.
$
$! Remember the time at which the job started.
$
$ start_time = f$time()
$
$! If this is a restart and we already resubmitted the procedure,
$! jump down to EXIT2 to unlock the procedure and quit.
$! Otherwise skip to the cleanup code and resubmit.
$

```

```

$   if $restart
$   then
$       if f$type(batch$restart) .nes. "" then -
$           if batch$restart .eqs. "RESUBMITTED" then goto exit2
$       goto exit ! To resubmit the job.
$   endif
$
$       .additional initialization
$       .
$
$       .procedure activities
$       .
$
$exit:
$   set noon
$   submit/nolog_file/after="''start_time+0-04:00"/restart -
$       'f$environment("PROCEDURE")
$   set restart_value=resubmitted
$
$       .more cleanup
$       .
$exit2:
$   undefine/system xda_batch_lock ! Remove the logical name lock.
$   exit status .or. %x10000000

```

The first thing the procedure does is check to see if the XDA_BATCH_LOCK logical name is defined. If so, it assumes that another batch job is running the same procedure and exits immediately. If the logical name is not defined, the procedure defines it. These two commands act as a lock to prevent more than one copy of the batch job from running at the same time. Any copies other than the first simply fade away without resubmitting themselves.

The cleanup code must delete the logical name so that it does not prevent future batch jobs from running. If the system crashes during the job, the system logical name will not be defined after the system reboots. Therefore the restarted batch job will run normally. The only problem arises when the operator cancels the job without restarting it. When the job runs again, the logical name is still defined and the job exits immediately. In this case someone must delete the logical name by hand.

If the GRPNAM and SYSNAM privileges are not available to the users of a batch job that must be locked, a file can be used instead of a logical name. The presence of the “lock file” signifies that the batch job is running; its absence signifies that the job is not running.

Chapter 22



Arrays

In most programming languages, an **array** is a collection of data elements of the same type. In the C language, for example, you can create an array of integers, an array of floating-point numbers, or an array of characters. Two items of information are needed to refer to an individual element of an array. The first is the name of the array, and the second is an identifier for the particular element in question. This identifier is called an **index** or **subscript**. Most languages, C included, require that an index be an integer. A few languages, MUMPS being a notable example, allow an index to be any value whatsoever. The array is elevated from a simple ordered sequence of elements to an associative array that maps one value (the index or key) to another value (the data item).

DCL does not provide built-in array facilities. However, using apostrophe substitution, you can simulate an array with a collection of symbols. A portion of the symbol name serves as the array name and a portion serves as the index. There is no requirement that all the array elements be of the same type, because each symbol can contain any type of data.

22.1 Arrays with Integer Indexes

Most programming languages provide arrays whose index values are integers. This can be simulated in DCL with a collection of symbols, one for each array element, whose names consist of a fixed alphabetic part (the array name) and an integer part (the index). For example, an array of file specifications might

use the symbols FILE_SPEC1, FILE_SPEC2, FILE_SPEC3, and so on. The array name is FILE_SPEC and the indexes are the integer suffixes 1, 2, and 3. If the symbol I is set to the desired index, the correct array symbol is obtained using the following form of apostrophe substitution:

```
... file_spec'i ...
```

When the symbol I is set to the value 2, the symbol named after substitution is

```
... file_spec2 ...
```

Note carefully that the apostrophe substitution results in the *name* of the array symbol containing the desired value. The substitution does not produce the value itself. Here is a complete example, which builds a file spec array from all the data files in the system manager's directory:

```
$ i = 0
$10: file = f$search("sys$manager:*.dat;")
$ if file .eqs. "" then goto 19
$ i = i + 1
$ file_spec'i = file ! Create an array element.
$ goto 10
$19:
$ file_count = i ! Remember the size of the array.
```

The symbol I is used as an index into the FILE_SPEC array. It is initialized to zero and incremented each time an array element is created. Therefore, the array elements will be named FILE_SPEC1, FILE_SPEC2, and so on, as desired. The command that creates an array element is an example of an assignment statement containing apostrophe substitution; array manipulation is one of the few operations that necessitates substitution in the left-hand side of an assignment statement.

The following loop scans the file specs in the array and eliminates any spec containing a dollar sign in its file name by replacing the spec with a placeholder string:

```
$ i = 0
$10: i = i + 1
$ if i .gt. file_count then goto 19
$ name = f$parse(file_spec'i,,,"NAME")
$ if f$locate("$",name) .ne. f$length(name) then -
    file_spec'i = "(system file)"
$ goto 10
$19:
```

As in the first example, the symbol `I` is used to index the array. The loop terminates when `I` becomes greater than the number of file specs in the array, as specified by `FILE_COUNT`. The name portion of the file spec is extracted using the `F$PARSE` lexical function and then examined for a dollar sign. If a dollar sign is found, the array element is set to the string `"(system file)"`.

It cannot be overemphasized that apostrophe substitution produces the name of the symbol containing an array element, not the value of the array element. After substitution in the preceding examples, a command includes the name of the symbol containing a file spec. It does not include the file spec itself, nor does it include a string literal containing the file spec. In these examples, it is convenient that substitution results in the name of the symbol, because it is precisely the name that is required on the left-hand side of an assignment statement or as an argument to a lexical function. When you want to assign a value to an array element or use an existing value in an expression, it is the name of the array element symbol that must appear after substitution.

But what if you need an array element in a command that does not accept an expression? If the command does not accept an expression, then it will not take the name of the array element symbol and automatically extract its value. It must directly receive the value of the element. You could decide that two substitutions are required, one to produce the array symbol name and then another one to substitute the symbol's value in the command. With this in mind, you might think that the following command would work:

```
$ delete 'file_spec''i
$!      ^-----^      left substitution
$!      ^^              right substitution
```

Unfortunately, the left substitution is performed first, so DCL attempts to substitute the value of the symbol `FILE_SPEC`. This symbol has nothing to do with the array elements, which are named `FILE_SPEC1`, `FILE_SPEC2`, and so forth. The symbol `FILE_SPEC` probably does not exist, or if it does, it contains a value that is unrelated to the problem at hand. There is no way to force DCL to perform the right substitution (of `I`) first. The solution is to use two commands:

```
$ file = file_spec'i
$ delete 'file
```

The appropriate array element is copied to the temporary symbol `FILE`, and that symbol is substituted in the `DELETE` command.

The contrast between substitution in expressions and substitution outside of expressions can be demonstrated clearly using procedure parameters. One array that is always present in a procedure is the parameter array. Because the procedure parameters are named P1–P8, they naturally form an array of eight character strings. The following procedure will display the values of its parameters:

```
#! Display all the parameters passed to this procedure.
$
$ i = 0
$10: i = i + 1
$ if i .gt. 8 then goto 19
$ if p'i .nes. "" then display "P", i, ": ", p'i
$ goto 10
$19:
```

The second IF command displays the value of a parameter if it is not null. On the third cycle through the loop, for example, when I has the value 3, the IF command becomes:

```
$ if p3 .nes. "" then write sys$output "P", i, ": ", p3
```

Because the IF and WRITE commands accept expressions, the *value* of P3 is used where its name is specified.

Compare the previous procedure to the following one, which assumes its parameters are file specs and displays the contents of the files:

```
#! Display the contents of the files specified as parameters.
$
$ i = 0
$10: i = i + 1
$ if i .gt. 8 then goto 19
$ file = p'i
$ if file .nes. "" then type/page 'file
$ goto 10
$19:
```

The TYPE command expects a file spec as its parameter, not an expression. Therefore, two commands are required: one to assign the parameter value to a temporary symbol, and a second one to test the symbol and type the file. An attempt to combine the two commands in a fashion similar to the first example:

```
$ if p'i .nes. "" then type/page p'i
```

would result in the following line after substitution:

```
$ if p3 .nes. "" then type/page p3
```

The TYPE command will attempt to display a file named P3, rather than the file specified by the third procedure parameter. It does not evaluate P3 because it does not accept an expression.

22.2 *Arrays with String Indexes*

In the previous section, substitution was used to insert the names of array symbols into commands. The symbol names consisted of a fixed alphabetic portion and a variable integer portion, the index. There is no reason why the index has to be an integer. It can be an arbitrary string, as long as it is composed of the alphanumeric characters valid in a symbol name. An array whose indexes are strings is called a **symbolic array**, because it associates the index string, which is presumably meaningful, or symbolic, with some other data item. The index string is often called the “key” and the associated data item the “value.”

A symbolic array can be used to maintain information about the CPU time consumed by users currently logged in to VMS. The information can be stored in symbols of the form CPU_XXX, where XXX is the name of a particular user. Because user names are composed of alphanumeric characters, they will always form a valid symbol name. The following loop collects the elapsed CPU time for each user on the system (assuming the user has WORLD privilege):

▷ Ch. 20

```
$ context = 0
$10: pid = f$pid(context)
$ if pid .eqs. "" then goto 19
$ if f$getjpi(pid,"GRP") .le. 1 then goto 10
$ username = f$edit(f$getjpi(pid,"USERNAME"), "TRIM")
$ if f$type(cpu_'username) .eqs. "" then cpu_'username = 0
$ cpu_'username = cpu_'username + f$getjpi(pid,"CPUTIM")
$ goto 10
$19:
```

The procedure uses the F\$PID lexical function to obtain the process IDs for all processes on the system, one at a time. System processes in groups 0 and 1 are ignored. In turn, the process ID is used to obtain the user name, say "JONES". If the symbol CPU_JONES does not exist, it is created and initialized to zero. The value of the symbol CPU_JONES is then incremented by the elapsed CPU time for the process. After the loop is complete, the total CPU time for each user will be accumulated, regardless of the number of times each user is logged in.

There are two problems connected with symbolic arrays, which reduce their usefulness. The first is that the key can be composed only of letters, digits, dollar sign, and underscore. DCL syntax errors occur if any other characters creep in. A file name, for example, could be used as a key prior to VMS Version 4.4 but not thereafter. With the advent of Version 4.4, hyphens were allowed in file names but not in symbol names. If a file name containing a hyphen were to be used in a symbol representing one of the symbolic array's elements, the symbol name would contain a hyphen and a DCL syntax error would result.

The second problem has to do with the collection of symbols that are created for a symbolic array. In the preceding example, each distinct user name results in a symbol whose name includes that user name. Because the user names are arbitrary, there is no way to predict which symbols are created. Therefore, when you want to cycle through the array later on, say to print a report, you don't know which symbols to inspect. How do you know which user names were encountered? There are two possible solutions. The first is to search all the processes again and use the resulting user names to index the array. Unfortunately, in the meantime, some users may have logged out and new ones logged in. There is a mismatch between your array and reality.

The other solution is to build a separate list of user names as the array is created. This list can be used later to cycle through the array. Here is the preceding example, modified to keep a list:

```
$   user_list = ""
$   context = 0
$10: pid = f$pid(context)
$   if pid .eqs. "" then goto 19
$   if f$getjpi(pid,"GRP") .le. 1 then goto 10
$   username = f$edit(f$getjpi(pid,"USERNAME"), "TRIM")
$   if f$type(cpu_'username) .eqs. ""
$   then
$       cpu_'username = f$getjpi(pid,"CPUTIM")
$       user_list = user_list + username + ","
$   else
$       cpu_'username = cpu_'username + f$getjpi(pid,"CPUTIM")
$   endif
$   goto 10
$19:
```

The logic is similar, but with the addition of one action. Each time a new user is encountered, the user name is appended to USER_LIST, along with a comma. When the loop terminates, USER_LIST will contain a list of all the distinct user names encountered as the processes were searched. For example,

if JONES, SMITH, and SNORK were logged in, USER_LIST will contain the string "JONES,SMITH,SNORK,". Note the trailing comma.

The following code will print a simple report showing the total CPU times for the users:

```
$    display f$fao("!12AS !10AS", "User", " Total CPU")
$    i = 0
$10:  i = i + 1
$    username = f$element(i, ",", user_list)
$    if username .eqs. "" then goto 19
$    display f$fao("!12AS !10UL", username, cpu_'username)
$    goto 10
$19:
```

After a heading is printed, the symbol I is used to index the list of users. The F\$ELEMENT function extracts each user name, which is assigned to USERNAME. When USERNAME finally contains the null string, every user name has been examined and the loop terminates. The value of USERNAME is used in two ways. First, it is displayed so that the user name is included in the report. Second, it is used as the index in the CPU array to extract the user's total CPU time and display it. The USER_LIST symbol contains the definitive list of users who have an entry in the CPU array.

Symbolic arrays are a powerful tool but must be used with care.

22.3

The DCL Symbol Table

All symbols you create using DCL are kept in an area of memory called the **symbol table**. Even though VMS is a virtual memory system, the amount of memory set aside for the symbol table is fixed. This amount is determined by the system generation (SYSGEN) parameter CLISYMTBL and is measured in units of 512-byte pages. Whenever VMS creates a process, it uses the parameter to determine the number of pages to set aside for the symbol table. Once the process is created, the symbol table cannot be enlarged.

When you develop a complex DCL application that requires many symbols, you may find that the symbol table space is exhausted during execution of the application. In this case, you have two choices: reduce the number of symbols, or ask the system manager to increase the value of the SYSGEN parameter CLISYMTBL. Any new processes created after the system manager increases the parameter will have a larger symbol table.

Appendix A



Hexadecimal Notation

Modern digital computers operate in the binary number system, so they represent data internally as quantities in base 2. Each binary digit is called a **bit**. DCL supports integer quantities, which occupy 32 bits. In the VAX architecture, a collection of 32 bits is called a **longword**, so DCL supports longword integers.

Programming languages such as DCL allow integers to be represented externally as decimal numbers, performing the necessary conversion to binary when the programs are prepared for execution. Decimal integers are appropriate for most situations, but occasionally the actual bit patterns within the integer are important. An example of this is the VMS status code, a longword integer that is divided into several bit fields. When the bit patterns are important, the decimal number system is not a particularly useful external representation.

▷ Ch. 8

Picture the longword integer 96,877 in binary:

```
000000000000000010111101001101101
```

A programming language might allow integers to be specified in binary, but doing so would require the programmer to type a sequence of 32 ones and zeros. The chance of making an error in the number is high. In order to reduce the length of the digit sequence, the base 16, or hexadecimal, number system is employed. Since the hexadecimal base, 16, is a power of the binary base, 2, the conversion

from binary to hexadecimal is nothing more than a grouping operation. Begin by separating the binary digits into groups of four:

0000	0000	0000	0001	0111	1010	0110	1101
------	------	------	------	------	------	------	------

Each group contains four binary digits and has one of 16 possible values from 0 through 15. This is exactly the number of possible values of one digit in a base-16 number. So a 32-digit binary number can be represented as an eight-digit hexadecimal number.

A single printable character must be chosen to represent each of the 16 hexadecimal digit values. The digit characters 0 through 9 are chosen for the digit values 0 through 9, and the letters A through F are chosen for the digit values 10 through 15. Now the groups of binary digits can be replaced by their equivalent hexadecimal digits:

0	0	0	1	7	A	6	D
---	---	---	---	---	---	---	---

The hexadecimal number 00017A6D is equivalent to the decimal number 96,877. DCL allows a number to be represented in hexadecimal by preceding its hexadecimal digits with a percent sign (%) and an X:

```
$ value = %x00017a6d
```

As with decimal numbers, leading zero digits are not required:

```
$ value = %x17a6d
```

Both of these assignment commands set the symbol VALUE to the hexadecimal value %X00017A6D. DCL converts the hexadecimal value to binary before setting the symbol.

Appendix B



DEC Multinational Character Set

This appendix contains two tables illustrating the complete DEC Multinational Character Set. Table B.1 lists the 128 characters in the standard seven-bit ASCII character set. Table B.2 lists the additional 128 characters in the eight-bit extended character set. Blank slots in the table have no assigned character and are reserved for future use by DIGITAL.

To determine the hexadecimal code for a particular character, begin by finding the character in the tables. The first digit of the hexadecimal code depends on the column containing the character; look up at the column heading to find the first digit. The second digit depends on the row containing the character; look to the left at the row heading to find the second digit. For example, the hexadecimal code for the question mark (?) is 3F, because the question mark is in column 3, row F.

A detailed description of the characters in the DEC Multinational Character Set can be found in the *Guide to Using VMS*.

Table B.1 Standard ASCII Character Set

digit 1:	0	1	2	3	4	5	6	7
digit 2:								
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table B.2 DEC Multinational Extension

digit 1:	8	9	A	B	C	D	E	F
digit 2:								
0		DCS		°	À		à	
1		PU1	ı	±	Á	Ñ	á	ñ
2		PU2	¢	²	Â	Ò	â	ò
3		STS	£	³	Ã	Ó	ã	ó
4	IND	CCH			Ä	Ô	ä	ô
5	NEL	MW	¥	μ	Á	Õ	á	õ
6	SSA	SPA		¶	Æ	Ö	æ	ö
7	ESA	EPA	§	·	Ç	Œ	ç	œ
8	HTS		⊗		È	Ø	è	ø
9	HTJ		©	¹	É	Ù	é	ù
A	VTS		ª	º	Ê	Ú	ê	ú
B	PLD	CSI	«	»	Ë	Û	ë	û
C	PLU	ST		¼	Ì	Ü	ì	ü
D	RI	OSC		½	Í	Ý	í	ý
E	SS2	PM			Î		î	
F	SS3	APC		¿	Ï	ß	ï	

Appendix C



Subroutine Library

This appendix contains a complete listing of the subroutine library described and used throughout this book. The subroutine library resides in a single, self-sufficient procedure containing a collection of DCL subroutines that can be used by DCL applications to perform common but nontrivial operations. The library is yours to modify and expand as required by your applications.

Each subroutine includes a block comment describing its function, parameters, and any return value. When the library is used for production applications, the comments should be moved to the end of the procedure so that they do not slow down its execution.

The general format of a call to a subroutine is as follows:

```
$    @directory:subroutine-library name parameter ...
```

The *directory* portion specifies the disk and directory containing the subroutine library. The first parameter, *name*, is the name of the desired subroutine. Additional *parameters* are required to specify the exact operation to be performed.

The examples in this book assume that the personal command LIBCALL is defined in the application's initialization code. The command is defined as follows:

```
$    libcall = "@directory:subroutine-library"
```

This personal command improves the readability of a subroutine call and takes up less space on the command line:

```
$ libcall name parameter ...
```

The subroutine library is self-sufficient and does not depend on any symbols or logical names defined by its caller.

File *SUBROUTINE-LIBRARY.COM*

```
$ sublib__status = %x10000000
$ sublib__success = sublib__status + %x0001
$ on control_y then exit sublib__status + %x0004
$ on warning then exit $status .or. %x10000000
$
$ display = "write sys$output"
$ goto 'p1
```

```

$! Title:      Ask a Question
$!
$! Synopsis:   This subroutine asks the user a question and returns
$!             the answer. The prompt for the question is composed
$!             of a query string and optionally a default answer.
$!
$! Parameters: P2: A global symbol to receive the answer.
$!             P3: The data type of the answer. B for boolean
$!                 (yes/no); I for integer; S for string.
$!             P4: The query string for the question. It must end
$!                 with a punctuation character and no space.
$!             P5: The default answer (optional; if not specified
$!                 then an answer must be entered).
$!             P6: A comma-separated list of options:
$!                 H: Display help before asking question.
$!                 S: Skip a line before asking question.
$!                 U: Uppcase the input string.
$!                 Z: Allow CTRL/Z as an answer.
$!             P7: The help specifier (optional). It must be in
$!                 the form "procedure [parameter...]". The
$!                 procedure is invoked with the at-sign command.
$!
$! Result:     For Boolean data type, a 0 (no) or 1 (yes). For
$!             Integer data type, the integer. For String data
$!             type, the string. If CTRL/Z is allowed and entered,
$!             the string "^Z" is returned.
$
$ASK:
$
$   signal = "@" + f$environment("PROCEDURE") + " signal ask"
$   if p3 .eqs. "B" .and. p5 .nes. "" .and. -
$       f$type(p5) .eqs. "INTEGER" then -
$       p5 = f$element(p5, "/", "NO/YES")
$   if p5 .nes. "" then p4 = f$extract(0, f$len(p4)-1, p4) + -
$       "[" + p5 + "]" + f$extract(f$len(p4)-1, 1, p4)
$   if f$locate("S", p6) .ne. f$length(p6) then display ""
$   if f$locate("H", p6) .ne. f$length(p6) then @'p7
$
$a10:   read sys$command /prompt="'p4 " input /end_of_file=a_eof
$   if input .eqs. "" then input = p5
$   input = f$edit(input, "TRIM")
$   if input .eqs. ""
$   then
$       signal w inputreq -
$           "Please enter a value; there is no default."

```

```

$      else if input .eqs. "?"
$      then
$          if p7 .nes. "" then @'p7
$          if p7 .eqs. "" then display -
$              "There is no help for this question."
$      else
$          goto a_'p3
$a_B:      input = f$edit(input,"UPCASE")
$          if f$locate(input,"YES") .eq. 0 .or. -
$              f$locate(input,"NO") .eq. 0
$          then
$              input = input .and. 1
$              goto a19
$          else
$              signal w yesnoreq "Please answer YES or NO."
$          endif
$          goto a15
$
$a_I:      if f$type(input) .eqs. "INTEGER"
$          then
$              input = f$integer(input)
$              goto a19
$          else
$              signal w intreq "The input must be an integer."
$          endif
$          goto a15
$
$a_S:      if f$locate("U",p6) .ne. f$length(p6) then -
$          input = f$edit(input,"UPCASE")
$          goto a19
$a15:
$          endif
$          endif
$          goto a10
$a_eof:
$          input = "^Z"
$          if f$locate("Z",p6) .ne. f$length(p6) then goto a19
$          signal i invctrlz "End-of-file is not a valid response."
$          goto a10
$a19:
$          'p2 == input
$          exit sublib__success

```

```

$! Title:      Look Up a Keyword in a List
$!
$! Synopsis:   This subroutine looks up a keyword or its
$!             abbreviation in a list of valid keywords.
$!             If the keyword exists in the list and is
$!             unique, the full keyword is returned.
$!
$! Parameters: P2: A global symbol to receive the result.
$!             P3: The keyword or a unique abbreviation thereof.
$!             P4: A comma-separated list of valid keywords.
$!
$! Result:     If the (abbreviated) keyword is valid and unique,
$!             the full keyword is returned.  If the keyword is
$!             invalid or null, the null string is returned.
$
$LOOKUP_KEYWORD:
$
$   'p2 == ""
$   if p3 .eqs. "" then exit sublib__success
$   p3 = "," + f$edit(p3,"UPCASE")
$   p4 = "," + f$edit(p4,"UPCASE") + ","
$   p4_tail = f$extract(f$locate(p3,p4)+1, 999, p4)
$   if f$locate(p3,p4_tail) .eq. f$length(p4_tail) then -
$       'p2 == f$element(0,"",p4_tail)
$   exit sublib__success

```

```

$! Title:          Signal an Informational or Error Message
$!
$! Synopsis:       This subroutine "signals" a message, producing one
$!                or more message lines in the standard VMS format.
$!                It also exits with a status whose severity matches
$!                that of the message.
$!
$! Parameters:     P2: The message facility code.
$!                P3: The message severity (S, I, W, E, or F).
$!                P4: The message identification.
$!                P5: The message text.
$!                Pn: Optional message lines or status codes whose
$!                    corresponding message lines are to be included.
$!
$! Status:         The severity of the exit status is equal to the
$!                message severity, except in the case of warnings.
$!                If the message severity is W, an informational
$!                severity is included in the status so that the
$!                caller's error handler is not invoked.
$
$SIGNAL:
$
$    prefix = f$fao("%!AS-!AS-!AS, ", p2, p3, p4)
$    i = 4
$s10:    i = i + 1
$        if i .gt. 8 then goto s19
$        if p'i .eqs. "" then goto s19
$        text = p'i
$        if f$type(text) .eqs. "INTEGER" then -
$            text = f$message(text)
$        if f$ext(0,1,text) .nes. "%" then text = prefix + text
$        if i .gt. 5 then text[0,1] := "-"
$        display text
$        goto s10
$s19:
$        if p3 .eqs. "W" then p3 = "I"
$        exit sublib__status + f$locate(p3,"WSEIF")

```

```

$! Title:      Generate a Unique Name
$!
$! Synopsis:   This subroutine generates a unique name suitable for
$!             use in creating a temporary file.
$!
$! Parameters: P2: The global symbol to receive the result.
$!             P3: The pattern specifying the format of the unique
$!                 name. It must contain a question mark (?), which
$!                 is replaced with a unique number.
$!
$! Result:     A unique name consisting of the pattern with the
$!             question mark replaced with a ten-digit number.
$!             The number is composed of eight digits of time and a
$!             two-digit counter.
$
$UNIQUE_NAME:
$
$   if f$type(sublib_counter) .eqs. "" then sublib_counter == 0
$   sublib_counter == (sublib_counter+1) - -
$                   (sublib_counter+1)/100*100
$   'p2 == f$fao("!AS!8AS!2ZL!AS", f$element(0,"?",p3), -
$           f$extract(12,11,f$time())-":"-":"-".", -
$           sublib_counter, f$element(1,"?",p3))
$   exit sublib__success

```

```

$! Title:          Check the Validity of a Symbol
$!
$! Synopsis:       This subroutine checks the validity of a string
$!                 representing the name of a symbol.  It checks the
$!                 length and the individual characters.
$!
$! Parameters:     P2: The global symbol to receive the result.
$!                 P3: The symbol to be checked.
$!                 P4: Optional maximum length of symbol (default 31).
$!                 P5: Optional valid characters in addition to the
$!                     standard letters, digits, dollar, and underscore.
$!
$! Result:         The null string if the symbol is okay.  Otherwise a
$!                 message fragment describing the error.
$
$VERIFY_SYMBOL:
$
$      p3 = f$edit(p3,"TRIM,UPCASE")
$      if p4 .eqs. "" then p4 = 31
$      'p2 == f$fao("is limited to !UL characters", f$integer(p4))
$      if f$length(p3) .gt. p4 then exit sublib__success
$      v = "$0123456789ABCDEFGHIJKLMNQRSTUUVWXYZ_" + p5
$      'p2 == ""
$      i = -1
$vs10:   i = i + 1
$      if i .ge. f$length(p3) then exit sublib__success
$      if f$locate(f$extract(i,1,p3),v) .lt. f$length(v) then -
$          goto vs10
$      'p2 == f$fao("contains the invalid character ""!AS"", -
$                  f$extract(i,1,p3))
$      exit sublib__success

```

Appendix D



Sample Application

The Manager for Organized Distribution Lists (MODL) is a DCL application that helps you manage distribution lists for the MAIL utility. A **distribution list** is a text file containing a list of users who are to receive mail about a particular topic. For example, you might have a distribution list for a software project; the list contains the user names of all the developers working on the project. You might have another list containing the names of all the managers in your organization. Distribution lists tend to proliferate, and the same people can appear in multiple lists. It quickly becomes difficult to manage the updating of distribution lists, and it can even take some time to locate the particular list you need.

The MODL system introduces the idea of a single distribution file containing a record for each user in the organization. The organization might be a project group, a department, or the entire company. The larger the organization encompassed by a MODL distribution file, the fewer distribution files there are. In addition to user records, the MODL file contains records for keywords, which are mnemonic identifiers for particular groups or projects. A keyword can be associated with one or more users who are members of the group or project for which the keyword stands. For example, the keyword **MANAGER** might signify people who are managers in the department. Once this keyword is associated with the users who are managers, mail can be sent to the managers by asking MODL to “send mail to all users with keyword **MANAGER**.” Another example is the keyword **CONSULTANT**. This keyword can be associated with the users who are outside consultants rather than employees. Then when you are sending mail

about a new company policy, you can ask MODL to “send mail to all users who are not a CONSULTANT.”

MODL distribution files are created and maintained through a set of commands supported by the MODL system.

D.1 Using MODL

The MODL system provides a command-oriented interface, which makes MODL behave like other command-oriented utilities such as MAIL. In order to invoke MODL, enter the MODL command:

```
$ modl
```

```
    Manager for Organized Distribution Lists
```

```
MODL>
```

MODL responds with a prompt. At the prompt you may enter one of the following:

- A MODL command. A command begins with a command verb and may require some parameters. If a parameter is omitted, MODL will prompt for it. See Section D.2 for a description of MODL commands.
- A VMS command. If the line you enter begins with a dollar sign (\$), MODL assumes the rest of the line contains a VMS command. The command is executed.
- A blank line. A blank line is ignored.
- A <CTRL/z>. When <CTRL/z> is pressed, MODL exits. The <CTRL/z> is equivalent to the EXIT command.

Lines entered at the MODL> prompt may contain comments. A comment begins with an exclamation point (!); the exclamation point and the rest of the line are ignored:

```
MODL> modl add manager shubin ! Phred Shubin just got promoted.
```

You can also use MODL to execute a single command and then immediately exit. If, at the DCL prompt, you include a command following the MODL verb, MODL executes the command and then immediately exits without beginning an interactive session.

```
$ modl send policy.txt "not consultant" "New vacation policy."
```

D.2 MODL Commands

The following sections describe the commands available with MODL. The commands are described in alphabetical order.

D.2.1 The ADD Command

The ADD command assigns keywords to users.

```
ADD keyword, ... user, ...
```

The *keyword* parameter is a single keyword or a list of keywords to be added to users. The keywords must have been previously defined with the REGISTER command.

The *user* parameter is a single user or a list of users to receive the keywords. The users must have been previously defined with the REGISTER command. The word SELF can be used in the list to specify the current user.

D.2.2 The CLOSE Command

The CLOSE command closes the current MODL distribution file.

```
CLOSE
```

The current MODL distribution file is closed. After this command is issued, no MODL file is selected.

D.2.3 The CREATE Command

The CREATE command creates a new MODL distribution file.

```
CREATE file-spec "title"
```

The *file-spec* parameter provides the specification of the new MODL file. If no device and directory are included, the file is created in the working directory. MODL provides a default file type of MODL_FILE.

The *title* parameter specifies a descriptive title for the new distribution file.

The new distribution file contains no registered users or keywords.

D.2.4

The DEREGISTER Command

The DEREGISTER command removes the definitions of users or keywords from the MODL distribution file.

```
DEREGISTER USER user [NOVERIFY]
```

The *user* parameter is the name of a user to be removed from the distribution file. The word SELF can be used to signify the current user.

MODL normally verifies the deregistration of a user. The optional parameter NOVERIFY specifies that MODL should not verify the operation. If the parameter is not specified, MODL does not prompt for it.

```
DEREGISTER KEYWORD keyword
```

The *keyword* parameter is a keyword to be removed from the distribution file. MODL requires that the keyword not be assigned to any users.

D.2.5

The DISPLAY Command

The DISPLAY command displays a list of all the users who match given selection criteria.

```
DISPLAY "selection"
```

The *selection* parameter is a boolean expression that specifies which users are to be displayed. All users matching the selection are displayed at the terminal.

The *selection* parameter is a boolean expression composed of keywords and operators. The keywords must be registered in the distribution file. The boolean operators are AND, OR, and NOT. The NOT operator is applied first, followed by AND, followed by OR. Parentheses may be used to alter the order in which operators are applied. Unless the *selection* parameter is a single keyword, it must be enclosed in double quotes.

Here are some examples:

```

MODL> ! Select users who are developers.
MODL> display developer

-or-

MODL> ! Select users who are developers or managers.
MODL> display "developer or manager"

-or-

MODL> ! Select users who are not managers.
MODL> display "not manager"

-or-

MODL> ! Select users who are developers, but exclude consultants.
MODL> display "developer and not consultant"

-or-

MODL> ! Invite all developers along with the marketing managers.
MODL> display "developer or marketing and manager"

-or-

MODL> ! The same expression clarified with parentheses.
MODL> display "developer or (marketing and manager)"

-or-

MODL> ! Invite all development and marketing managers.
MODL> display "(developer or marketing) and manager"

```

D.2.6 *The EXIT Command*

The EXIT command causes MODL to exit back to DCL.

```
EXIT
```

D.2.7 *The LIST Command*

The LIST command creates a VMS MAIL distribution list containing users who match given selection criteria.

```
LIST "selection" output-file "heading"
```

The *selection* parameter is a boolean expression that selects the users in the MODL distribution file who are to be included in the MAIL distribution list. See the DISPLAY command for a description of the *selection* parameter.

The *output-file* parameter is the file spec for the MAIL distribution list. The default device/directory is the working directory, and the default file type is DIS.

The *heading* parameter is a text string to be used as a heading in the MAIL distribution list.

D.2.8 *The MODIFY Command*

The MODIFY command updates information about users or keywords.

```
MODIFY USER user "new full name" "new title" new-mail-address
```

The *user* parameter specifies the name of the user to be updated. The word SELF can be used to signify the current user.

The *new full name*, *new title*, and *new-mail-address* parameters specify the new information about the user. See the REGISTER command for a description of these items.

```
MODIFY KEYWORD keyword "new title"
```

The *keyword* parameter specifies the keyword to be updated.

The *new title* parameter specifies a new title for the keyword. See the REGISTER command for a description of these items.

D.2.9 *The OPEN Command*

The OPEN command selects a MODL distribution file and prepares it for use.

```
OPEN file-spec
```

The *file-spec* parameter provides the specification of an existing MODL distribution file to be opened.

D.2.10

The REGISTER Command

The REGISTER command registers a new user or keyword in the MODL distribution file.

```
REGISTER USER name "full name" "title" mail-address
```

The *name* parameter specifies the name to be associated with the new user. The name will usually be identical to the person's VMS user name.

The *full name*, *title*, and *mail-address* parameters specify information about the user. The *full name* is a string containing the person's full name. The *title* is a string containing the person's business title or other identifying information. The *mail-address* is the person's DECnet mail address.

```
REGISTER KEYWORD keyword "title"
```

The *keyword* parameter specifies a string that identifies the keyword being registered. The keyword can be composed of letters, numbers, dollar sign (\$), underscore (_), and hyphen (-).

The *title* parameter specifies a string that describes the meaning of the keyword.

D.2.11

The REMOVE Command

The REMOVE command removes keywords from users.

```
REMOVE keyword,... user,...
```

The *keyword* parameter is a single keyword or a list of keywords to be removed from users.

The *user* parameter is a single user or a list of users from whom the keywords are to be removed. The users must have been previously defined with the REGISTER command. The word SELF can be used in the list to specify the current user. If this parameter is specified as an asterisk (*), the keywords are removed from all registered users.

D.2.12 *The SEND Command*

The SEND command sends mail to a group of users who match given selection criteria.

```
SEND file-spec "selection" "subject"
```

The *file-spec* parameter is the spec of the file containing the message to be sent. The default device/directory is the working directory. The default file type is TXT.

The *selection* parameter is a boolean expression that specifies the users in the MODL distribution file who will receive the message. See the DISPLAY command for a description of the *selection* parameter.

The *subject* parameter is a text string specifying the subject of the message.

The method used to send the message is determined by the SET AGENT and SET SEND commands.

D.2.13 *The SET Command*

The SET command establishes various operating characteristics for MODL.

```
SET AGENT "string"
```

This command establishes the mailing agent for future SEND commands. The *string* parameter is the code for the mailing agent followed by a percent sign (%). For example, the Zipnet agent might be specified as "zp%". The mailing agent is stored in the MODL distribution file, so all users of the file will use the same agent.

```
SET NOAGENT
```

This command specifies that future SEND commands will not use a mailing agent. When there is no mailing agent, the SEND command sends the message by invoking the MAIL utility once for each recipient.

When a new MODL distribution file is created, no mailing agent is stored in it.

```
SET SEND copy-self? batch?
```

This command establishes the operating characteristics for future SEND commands. The *copy-self?* parameter specifies whether an additional copy of each message should be sent to the user initiating the message. The *batch?* parameter specifies whether message sending should be performed in batch or interactively. This command establishes the operating characteristics for the remainder of the current session; it does not save the characteristics in the MODL distribution file.

If no SET SEND command has been issued during a terminal session, the default for *copy-self?* is false, the default for *batch?* is true.

D.2.14 *The SHOW Command*

The SHOW command displays information about various aspects of the MODL system.

SHOW FILE

This command displays information about the current MODL distribution file, including its file spec, title, and agent.

SHOW KEYWORD *keyword*

This command displays information about a registered keyword. The *keyword* parameter specifies the keyword in question. The information displayed includes the keyword and its title. The parameter may also be specified as an asterisk (*), in which case information about all registered keywords is displayed.

SHOW SEND

This command displays information about the operating characteristics established with the SET command. The information includes the mailing agent, copy-self status, and batch status.

SHOW USER *name*

This command displays information about a registered user. The *name* parameter specifies the user in question. The information displayed includes the user name, full name, title, mailing address, and keyword list. The parameter can also be specified as an asterisk (*), in which case information about all registered users is displayed.

D.3

File Layout

A MODL distribution file is an indexed file containing three types of records. The file has a single key, which occupies the first field in every record.

A MODL file has a header record, which describes the file and contains information that pertains to the entire file, such as the mailing agent. Table D.1 describes the fields in the header record.

There is a record for each keyword registered in the file. The record contains the keyword's title. Table D.2 describes the fields in a keyword record.

There is a record for each user registered in the file. The record contains information that describes the user, the user's mailing address, and a list of the keywords associated with the user. Table D.3 describes the fields in a user record.

Table D.1 MODL File: Header Record

Position	Size	Type	Description
0	16	String	The key of the header record is !HEADER!.
16	12	String	The user name of the user who created the file.
28	17	String	The date and time when the file was created.
45	32	String	The title of the file.
77	32	String	The mailing agent, or blank if no agent is used.

Table D.2 MODL File: Keyword Record

Position	Size	Type	Description
0	16	String	The key of the keyword record consists of a sharp sign (#) followed by the keyword.
16	12	String	The user name of the user who registered the keyword.
28	17	String	The date and time when the keyword was registered.
45	32	String	The title of the keyword.

Table D.3 MODL File: User Record

Position	Size	Type	Description
0	16	String	The key of the user record consists of the user name.
16	12	String	The user name of the user who registered this user.
28	17	String	The date and time when the user was registered.
45	32	String	The user's business title.
77	32	String	The user's full name.
109	64	String	The user's mailing address.
173	up to 200	String	A comma-separated list of the keywords associated with the user.

File MODL.COM

```

$      define = "define/nolog"
$      delete = "delete"
$      mail   = "mail/noself"
$      submit = "submit"
$
$
$      saved_message = f$environment("MESSAGE")
$      set message/facility/severity/identification/text
$
$      modl__status = %x11088000
$      modl__success = modl__status + %x001
$      modl__ctrly   = modl__status + %x00c
$      status = modl__success
$      on control_y then goto control_y
$      on warning then goto error
$
$
$      check_open_file = -
$          "if f$trnlm("modl_file") .eqs. "" then " + -
$          "signal e nofileopen "No MODL file has been opened.""
$      display          = "write sys$output"
$      false            = 0
$      libcall          = "@modl_system:subroutine-library"
$      modlcall         = "@modl_system:modl_subroutines.com"
$      read_lock        = "read/lock modl_file"
$      read_lock_header = "read/lock modl_file /key=""!HEADER!"
$      read_nolock      = "read/nolock modl_file"
$      read_nolock_header = -
$          "read/nolock modl_file /key=""!HEADER!"
$      this_user        = f$edit(f$getjpi("", "USERNAME"), "TRIM")
$      true             = 1
$      undefine         = "deassign"
$      update_unlock    = "write/update/symbol modl_file"
$      write_unlock     = "write/symbol modl_file"
$      ask              = libcall + " ask"
$      signal           = libcall + " signal modl"
$
$
$!     Establish defaults for the copy-self? and batch? settings.
$
$
$      if f$type(modl_batch) .eqs. "" then modl_batch == true
$      if f$type(modl_copy_self) .eqs. "" then -
$          modl_copy_self == false

```

```

$
$ mf_t_key = 0 ! Key: !HEADER! for the header;
$ mf_s_key = 16 ! #keyword for a keyword record;
$ ! Username for a user record.
$ mf_t_doer = 16 ! User who created/updated record.
$ mf_s_doer = 12
$ mf_t_time = 28 ! Date/time of creation/update.
$ mf_s_time = 17
$ mf_t_title = 45 ! Title (all records).
$ mf_s_title = 32
$ mf_t_agent = 77 ! Mailing agent (header).
$ mf_s_agent = 32
$ mf_t_name = 77 ! Full name (user).
$ mf_s_name = 32
$ mf_t_address = 109 ! Mailing address (user).
$ mf_s_address = 64
$ mf_t_keylist = 173 ! List of keywords (user).
$ mf_s_keylist = 200
$
$
$! If a MODL distribution file has been OPENed by the user,
$! then open the RMS file so we can read and write it.
$
$ if f$strnlm("modl_opened_file") .nes. "" then -
    open/read/write/share=write modl_file -
        modl_opened_file /error=5
$
$ goto 9
$5: @modl_system:modl_close
$ signal f errselfile -
    "Error opening MODL file ''f$strnlm("modl_opened_file")'' -
    '$status
$9:
$ command_list = "ADD,CLOSE,CREATE,DEFINE,DEREGISTER," + -
    "DISPLAY,EXIT,HELP,LIST,MODIFY,OPEN," + -
    "REGISTER,REMOVE,SEND,SET,SHOW,UNDEFINE"
$
$ if p1 .nes. ""
$ then ! The command line includes a MODL command.
$ libcall lookup_keyword modl_ ''p1'' 'command_list
$ if modl_ .eqs. "" then signal f unkcnd -
    "The command ''p1 is unknown or ambiguous."
$ @modl_system:modl_'modl_ ''p2'' ''p3'' ''p4'' -
    ''p5'' ''p6'' ''p7'' ''p8''

```

```

$
$ else ! No command, so start a session.
$ display ""
$ display " Manager for Organized Distribution Lists
$ display ""
$10: read sys$command/prompt="MODL> " command /end_of_file=19
$ command = f$edit(command,"TRIM,UNCOMMENT")
$ if command .eqs. "" then goto 10
$ if f$extract(0,1,command) .eqs. "$"
$ then ! It's a VMS command.
$ set noon
$ define/user_mode sys$input sys$command
$ 'f$extract(1,999,command)
$ if f$getdvi("SYS$INPUT","DEVCLASS") .ne. 1 then -
$ undefine/user_mode sys$input ! Undo DEFINE/USER.
$ set on
$ display ""
$
$ else ! It's a MODL command.
$ verb = f$element(0," ",command)
$ libcall lookup_keyword modl_ ''verb'' 'command_list
$ if modl_ .eqs. "EXIT" then goto 19
$ if modl_ .eqs. ""
$ then
$ signal i unkcmdverb -
$ "The command ''verb is unknown or ambiguous."
$
$ else
$ set noon
$ @modl_system:modl_'modl_'f$string(command - verb)
$ set on
$ if f$trnlrm("modl_file") .nes. "" then -
$ read_nolock_header junk ! Unlock all records.
$ endif
$ endif
$ goto 10
$19:
$ endif
$ goto EXIT

```

```

$
$
$control_y:
$     status = modl__ctrly
$     goto EXIT
$
$error:
$     status = $status
$     goto EXIT
$
$exit:
$     set noon
$     close/nolog modl_file
$     set message 'saved_message
$     exit status .or. %x1000000
$
$! Title:      Manager for Organized Distribution Lists (MODL)
$!
$! Synopsis:   MODL is a system that helps manage distribution
$!             lists for the MAIL utility.  A MODL distribution
$!             file is an indexed file that cross-references
$!             users and keywords.  A mail message can be sent
$!             to some or all of the registered users by specifying
$!             a keyword or boolean combination of keywords.  MODL
$!             selects the users who fit the keyword selection and
$!             sends the message to them.
$!
$! Usage:      MODL is run by invoking this procedure.  It is best
$!             to define a personal command:
$!
$!             modl == "@modl_system:modl"
$!
$!             When MODL is invoked with no parameters, it
$!             initiates a command loop which reads a command
$!             and executes it.  When MODL is invoked with
$!             parameters, it executes the single command
$!             specified by the parameters and then exits.
$!
$! Author:     Paul C. Anagnostopoulos
$! Created:    July 1988

```

File MODL_ADD.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      new_keys = p1
$      users = p2
$      if new_keys .eqs. "" then ask modl_ s "_Keyword List:"
$      if new_keys .eqs. "" then new_keys = modl_
$      if users .eqs. "" then ask modl_ s "_User List:"
$      if users .eqs. "" then users = modl_
$      new_keys = f$edit(new_keys,"COLLAPSE,UPCASE")
$      users = f$edit(users,"COLLAPSE,UPCASE")
$
$      i = -1
$10:    i = i + 1
$      keyword = f$element(i,",",new_keys)
$      if keyword .eqs. "," then goto 19
$      key = f$fao("!<#!AS!>", mf_s_key, keyword)
$      read_nolock /key="''key'" rec /error=15
$      goto 10
$15:    signal e undefkey "Keyword ''keyword is not defined."
$19:
$
$      i = -1
$20:    i = i + 1
$      username = f$element(i,",",users)
$      if username .eqs. "," then goto 29
$      if username .eqs. "SELF" then username = this_user
$      key = f$fao("!#AS", mf_s_key, username)
$      read_lock /key="''key'" rec /error=25
$      key_list = f$edit(f$extract(mf_t_keylist,mf_s_keylist,-
$                               rec), "TRIM")
$
$      gosub add_keywords
$      rec[mf_t_keylist,mf_s_keylist] := "''key_list'"
$      update_unlock rec
$      goto 20
$25:    signal w unreguser "User ''username is not registered."
$29:
$      exit modl__success
$
$
$add_keywords:
$      k = -1
$80:    k = k + 1
$      keyword = f$element(k,",",new_keys)
$      if keyword .eqs. "," then goto 89
```

```

$      if f$locate(","+keyword+",", " ","+key_list+",") .eq. -
          f$length(key_list)+2
$      then
$          if key_list .nes. "" then key_list = key_list + ","
$          key_list = key_list + keyword
$      else
$          signal i userhaskey -
              "User ''username already has keyword ''keyword."
$      endif
$      goto 80
$89:   return modl__success
$
$! Module:      Process the ADD Command
$!
$! Synopsis:   This module processes the ADD command, which is
$!             used to associate keywords with users. Both the
$!             keywords and users must be previously registered
$!             with the REGISTER command.
$!
$! Format:     ADD keyword,... user,...
$!
$! Parameters: keyword:  A registered keyword to be associated with
$!                       the specified user(s).
$!             user:    A registered user with whom the keyword is
$!                       associated. The word SELF can be used to
$!                       specify the current user.
$!
$! Notes:

```

File MODL_CLOSE.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      close/nolog modl_file
$      undefine modl_opened_file
$      exit modl__success
$
$! Module:      Process the CLOSE Command
$!
$! Synopsis:    This module processes the CLOSE command, which
$!              closes the current MODL file. After the command
$!              completes, there is no current MODL file.
$!
$! Format:      CLOSE
$!
$! Parameters:  none
$!
$! Notes:
```

File MODL_CREATE.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$
$      file_spec = p1
$      title = p2
$      if file_spec .eqs. "" then ask modl_ s -
$                               "_MODL Distribution File:"
$      if file_spec .eqs. "" then file_spec = modl_
$      if title .eqs. "" then ask modl_ s "_Title:"
$      if title .eqs. "" then title = modl_
$
$      file_spec = f$parse(file_spec,".modl_file",,,"SYNTAX_ONLY")
$      if f$search(file_spec) .nes. "" then signal e fileexists -
$                               "The MODL file 'file_spec already exists."
$      create/fdl=modl_system:modl_file.fdl 'file_spec
$
$      open/read/write modl_file 'file_spec
$      hdr = ""
$      hdr[mf_t_key,mf_s_key]      := "!HEADER!"
$      hdr[mf_t_time,mf_s_time]    := "'f$time()'"
$      hdr[mf_t_doer,mf_s_doer]    := "'this_user'"
$      hdr[mf_t_title,mf_s_title] := "'title'"
$      hdr[mf_t_agent,mf_s_agent] := " "
$      write_unlock hdr
$      close modl_file
$
$      @modl_system:modl_open 'file_spec
$      exit modl__success
$
$! Module:      Process the CREATE Command
$!
$! Synopsis:    This module processes the CREATE command, which is
$!              used to create a new MODL distribution file. The
$!              new file has no registered users or keywords.
$!              There is no mailing agent stored in the file.
$!              The file is automatically opened after creation.
$!
$! Format:      CREATE file-spec "title"
$!
$! Parameters: file-spec:  The file spec for the new MODL file.
$!                  The default file type is MODL_FILE.
$!                  title:   A title for the new MODL file.
$!
$! Notes:
```

File MODL_DEREGISTER.COM

```
$      on control_y then exit modl__ctrl
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      what = p1
$      if what .eqs. "" then ask modl_ s -
$                               "_Deregister KEYWORD or USER:"
$      if what .eqs. "" then what = modl_
$      libcall lookup_keyword modl_ ''what'' KEYWORD,USER
$      if modl_ .eqs. "" then signal e unkwhat -
$                               "The item ''what is unknown or ambiguous."
$      goto 'modl_
$
$
$
$KEYWORD:
$      keyword = p2
$      if keyword .eqs. "" then ask modl_ s "_Keyword:"
$      if keyword .eqs. "" then keyword = modl_
$      keyword = f$edit(keyword,"TRIM,UPCASE")
$
$      key = f$fao("!#<#!AS!>", mf_s_key, keyword)
$      read_nolock /key="''key''" rec /error=k15
$      goto k19
$k15:  signal e keynotreg "Keyword ''keyword is not registered."
$k19:
$      read_nolock /key="$"/match=ge rec /end_of_file=k29
$k30:  key_list = f$edit(f$extract(mf_t_keylist,mf_s_keylist,-
$                               rec), "TRIM")
$      if f$locate(", "+keyword+",", ", "+key_list+",") .ne. -
$      f$length(key_list)+2 then -
$      signal e keyinuse -
$      "At least one user has keyword ''keyword." -
$      "Use the REMOVE command first."
$      read_nolock rec /end_of_file=k39
$      goto k30
$k39:
$      read_nolock/delete /key="''key''" rec
$      exit modl__success
$
$
$
$USER:
$      username = p2
$      verify = p3 .nes. "NOVERIFY"
$      if username .eqs. "" then ask modl_ s "_User:"
$      if username .eqs. "" then username = modl_
$      username = f$edit(username,"TRIM,UPCASE")
$
```

```

$      if username .eqs. "SELF" then username = this_user
$      key = f$fa0("#AS", mf_s_key, username)
$      read_lock /key=""key' rec /error=u15
$      goto u19
$u15:  signal e usernotreg "User ''username is not registered."
$u19:
$
$      if verify
$      then
$          ask modl_ b -
$              "Are you sure you want to deregister ''username?" no
$          if .not. modl_ then exit modl__success
$      endif
$
$      read_nolock/delete /key=""key' rec
$      exit modl__success
$
$! Module:      Process the DEREGISTER Command
$!
$! Synopsis:    This module processes the DEREGISTER command, which
$!              is used to remove registered users and keywords from
$!              the MODL file.
$!
$!              When a user is deregistered, the user record is
$!              removed from the MODL file. No information about
$!              the user remains.
$!
$!              When a keyword is deregistered, the command first
$!              checks to ensure that the keyword is not associated
$!              with any users. Then the keyword record is removed.
$!
$! Format:      DEREGISTER USER user-name [NOVERIFY]
$!
$! Parameters:  user-name:  The user name of the user.
$!              NOVERIFY:   If this parameter is not specified, the
$!                          command prompts to ensure that you
$!                          want to deregister the user. If it is
$!                          specified, no prompt occurs.
$!
$! Format:      DEREGISTER KEYWORD keyword
$!
$! Parameters:  keyword:    The name of the keyword.
$!
$! Notes:

```

File MODL_DISPLAY.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      selection = p1
$      if selection .eqs. "" then ask modl_ s "_Selection:"
$      if selection .eqs. "" then selection = modl_
$
$      modlcall parse_selection modl_selection "'selection'"
$      read_nolock_header hdr
$      agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),"TRIM")
$
$      fao = "!12AS !32AS !32AS"
$      display ""
$      display f$fao(fao, "User", "Full Name", "Mailing Address")
$      display f$fao(fao, "----", "---- ----", "----- -----")
$      count = 0
$      read_nolock /key="$"/match=ge rec /end_of_file=19
$10:      kl = "," + -
           f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
                 "TRIM") + ","
$      if 'modl_selection then gosub display_user
$      read_nolock rec /end_of_file=19
$      goto 10
$19:
$      display f$fao("!/Users displayed: !UL!/", count)
$      exit modl__success
$
$
$display_user:
$      user = f$edit(f$extract(mf_t_key,mf_s_key,rec),"TRIM")
$      name = f$edit(f$extract(mf_t_name,mf_s_name,rec),"TRIM")
$      address = f$edit(f$extract(mf_t_address,mf_s_address,rec),-
                       "TRIM")
$      display f$fao(fao, user, name, agent + address)
$      count = count + 1
$      return
$
```

\$! Module: Process the DISPLAY Command
\$!
\$! Synopsis: This module processes the DISPLAY command, which is
\$! used to display a list of the users who satisfy a
\$! given selection expression. The users are displayed,
\$! but no mail message is sent. The display includes
\$! the user name, full name, and mailing address.
\$!
\$! Format: DISPLAY "selection"
\$!
\$! Parameters: selection: The selection expression specifying the
\$! users to be displayed.
\$!
\$! Notes: See the User's Guide for a description of the
\$! selection expression.

File MODL_HELP.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$
$      define/user_mode sys$input sys$command
$      help/library=modl_system:modl_helplib.hlb -
          /page/noinstruction/prompt -
          'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$      exit modl__success
$
$! Module:      Process the HELP Command.
$!
$! Synopsis:    This module processes the HELP command, which
$!              provides online help for MODL. The help text
$!              is stored and displayed using the VMS HELP.
$!
$! Format:      HELP [topic ...]
$!
$! Parameters:  topic:  A topic or topic hierarchy that selects
$!                    specific help information to be displayed.
$!                    If this parameter is missing, the help
$!                    facility displays the top-level help screen.
$!
$! Notes:      The help text is not included in this listing.
```

File MODL_LIST.COM

```
$      status = modl__success
$      on control_y then goto CONTROL_Y
$      on warning then goto ERROR
$      check_open_file
$
$      selection = p1
$      output_file = p2
$      heading = p3
$      if selection .eqs. "" then ask modl_ s "_Selection:"
$      if selection .eqs. "" then selection = modl_
$      if output_file .eqs. "" then ask modl_ s "_Output File:"
$      if output_file .eqs. "" then output_file = modl_
$      if heading .eqs. "" then ask modl_ s "_Heading:"
$      if heading .eqs. "" then heading = modl_
$
$      modlcall parse_selection modl_selection "'selection'"
$      read_nolock_header hdr
$      agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),"TRIM")
$      output_file = f$parse(output_file, ".DIS",,, "SYNTAX_ONLY")
$
$      create 'output_file
$      open/append modl_list 'output_file
$      write modl_list "! ", heading
$      write modl_list ""
$      write modl_list "! This is a MODL distribution list."
$      write modl_list "! File:      ", -
$              f$strlnm("modl_opened_file")
$      write modl_list "! Selection: ", selection
$      write modl_list ""
$
$      read_nolock /key="$"/match=ge rec /end_of_file=19
$10:      kl = ", " + -
$              f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
$              "TRIM") + ", "
$      if 'modl_selection then gosub list_user
$      read_nolock rec /end_of_file=19
$      goto 10
$19:
$      goto EXIT
$
$CONTROL_Y:
$      status = modl__ctrly
$      goto EXIT
$ERROR:
$      status = $status
$      goto EXIT
```

```

$EXIT:
$      set noon
$      close/nolog modl_list
$      exit status .or. %x1000000
$
$
$list_user:
$      name = f$edit(f$extract(mf_t_name,mf_s_name,rec),"TRIM")
$      address = agent + -
$                  f$edit(f$extract(mf_t_address,mf_s_address,rec),-
$                          "TRIM")
$
$      s = f$length(address)
$      if s .lt. 32 then s = 32
$      write modl_list f$fao("#AS !! !AS", s, address, name)
$      return
$
$
$! Module:      Process the LIST Command.
$!
$! Synopsis:    This module processes the LIST command, which is
$!              used to create a MAIL distribution list.  The users
$!              included in the list are those who satisfy a given
$!              selection expression.  The distribution list is
$!              created, but no mail message is sent.
$!
$! Format:      LIST "selection" output-file "heading"
$!
$! Parameters:  selection:      The selection expression specifying
$!                              the user to be included in the list.
$!                              output-file:    The spec of the distribution list.
$!                              The default file type is DIS.
$!                              heading:        A heading to be included at the
$!                              beginning of the distribution file.
$!
$! Notes:      See the User's Guide for a description of the
$!              selection expression.

```

File MODL_MODIFY.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      what = p1
$      if what .eqs. "" then ask modl_ s "_Modify KEYWORD or USER:"
$      if what .eqs. "" then what = modl_
$      libcall lookup_keyword modl_ "'what'" KEYWORD,USER
$      if modl_ .eqs. "" then signal e unkwhat -
$                               "The item 'what' is unknown or ambiguous."
$      goto 'modl_
$
$
$
$KEYWORD:
$      keyword = p2
$      new_title = p3
$      if keyword .eqs. "" then ask modl_ s "_Keyword:"
$      if keyword .eqs. "" then keyword = modl_
$      keyword = f$edit(keyword,"TRIM,UPCASE")
$      key = f$fao("!#<#!AS!>", mf_s_key, keyword)
$      read_lock /key="'key'" rec /error=15
$      goto 19
$15:   signal e keynotreg -
$       "The keyword 'keyword' is not registered."
$19:
$      old_title = f$edit(f$extract(mf_t_title,mf_s_title,rec),-
$                       "TRIM")
$      if new_title .eqs. "" then ask modl_ s "_New Title:" -
$                                           "'old_title'"
$      if new_title .eqs. "" then new_title = modl_
$      rec[mf_t_title,mf_s_title] := "'new_title'"
$      update_unlock rec
$      exit modl__success
$
$
$USER:
$      username = p2
$      new_full_name = p3
$      new_title = p4
$      new_address = p5
$      if username .eqs. "" then ask modl_ s "_User Name:"
$      if username .eqs. "" then username = modl_
$      username = f$edit(username,"TRIM,UPCASE")
$      if username .eqs. "SELF" then username = this_user
$      key = f$fao("!#AS", mf_s_key, username)
$      read_lock /key="'key'" rec /error=55
$      goto 59
```

```

$55:   signal e usernotreg "User ''username is not registered."
$59:
$      old_full_name = f$edit(f$extract(mf_t_name,mf_s_name,rec),-
$                               "TRIM")
$      old_title = f$edit(f$extract(mf_t_title,mf_s_title,rec),-
$                               "TRIM")
$      old_address = f$edit(f$extr(mf_t_address,mf_s_address,rec),-
$                               "TRIM")
$      if new_full_name .eqs. "" then ask modl_ s -
$                               "_New Full Name:" ""''old_full_name''
$      if new_full_name .eqs. "" then new_full_name = modl_
$      if new_title .eqs. "" then ask modl_ s "_New Title:" -
$                               ""''old_title''
$
$      if new_title .eqs. "" then new_title = modl_
$      if new_address .eqs. "" then ask modl_ s "_New Address:" -
$                               ""''old_address''
$
$      if new_address .eqs. "" then new_address = modl_
$      rec[mf_t_name,mf_s_name]      := ""''new_full_name''
$      rec[mf_t_title,mf_s_title]    := ""''new_title''
$      rec[mf_t_address,mf_s_address] := ""''new_address''
$      update_unlock rec
$      exit modl__success
$
$! Module:      Process the MODIFY Command
$!
$! Synopsis:   This module processes the MODIFY command, which is
$!             used to modify the information about registered
$!             users and keywords.
$!
$! Format:     MODIFY USER user-name "new full name" "new title" -
$!             new-address
$!
$! Parameters: user-name:      The name of the registered user
$!             to be updated.
$!             new full name:  The full name of the user.
$!             new title:     The business title of the user.
$!             new address:   The VMS mailing address of the user.
$!
$! Format:     MODIFY KEYWORD keyword "new title"
$!
$! Parameters: keyword:      The name of the registered keyword
$!             to be updated.
$!             new title:    A descriptive title for the keyword.
$!
$! Notes:     As usual, if a parameter is omitted, MODL prompts
$!             for it. The default answer is the previous value
$!             of the field.

```

File MODL_OPEN.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$
$      file_spec = p1
$      if file_spec .eqs. "" then ask modl_ s -
$                               "_MODL Distribution File:"
$      if file_spec .eqs. "" then file_spec = modl_
$
$      file_spec = f$parse(file_spec, ".modl_file",,, "SYNTAX_ONLY")
$      if f$search(file_spec) .eqs. "" then signal e filnotfnd -
$                               "MODL file ''file_spec does not exist."
$
$      if f$strnlm("modl_opened_file") .nes. "" then -
$          @modl_system:modl_close
$
$      open/read/write/share=write modl_file 'file_spec /error=5
$      goto 9
$5:    signal e filnotopen "Cannot open MODL file ''file_spec'" -
$          '$status
$9:
$
$      read_nolock_header hdr /error=15
$      goto 19
$15:   close modl_file
$      signal e notmodlfile "File ''file_spec is not a MODL file."
$19:
$      define modl_opened_file 'file_spec
$      exit modl__success
$
$! Module:      Process the OPEN Command
$!
$! Synopsis:    This module processes the OPEN command, which is
$!              used to select a MODL distribution file and prepare
$!              it for use. The MODL file must have been previously
$!              created using the CREATE command.
$!
$! Format:      OPEN file-spec
$!
$! Parameters:  file-spec:      The file spec of the MODL
$!                          distribution file.
$!
$! Notes:      Most other MODL commands require that a MODL
$!              distribution file already be open.
```

File MODL_REGISTER.COM

```
$      on control_y then exit modl__ctrl_y
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      what = p1
$      if what .eqs. "" then ask modl_ s -
$                               "_Register USER or KEYWORD:"
$
$      if what .eqs. "" then what = modl_
$      libcall lookup_keyword modl_ "'what'" KEYWORD,USER
$      if modl_ .eqs. "" then signal e unkwat -
$                               "The item 'what' is unknown or ambiguous."
$
$      goto 'modl_
$
$
$
$KEYWORD:
$      keyword = p2
$      title = p3
$      if keyword .eqs. "" then ask modl_ s "_Keyword:"
$      if keyword .eqs. "" then keyword = modl_
$      if title .eqs. "" then ask modl_ s "_Title:"
$      if title .eqs. "" then title = modl_
$
$      keyword = f$edit(keyword,"TRIM,UPCASE")
$      libcall verify_symbol modl_ "'keyword'" -
$                               'f$integer(mf_s_key-1) "-"
$
$      if modl_ .nes. "" then signal e invkey -
$                               "Keyword 'keyword' 'modl_."
$
$      key = f$fao("!#<#!AS!>", mf_s_key, keyword)
$      read_nolock /key="'key'" rec /error=19
$      signal e keyalrddydef "Keyword 'keyword' is already defined."
$19:
$      rec = ""
$      rec[mf_t_key,mf_s_key]      := "'key'"
$      rec[mf_t_time,mf_s_time]   := "'f$time()'"
$      rec[mf_t_doer,mf_s_doer]   := "'this_user'"
$      rec[mf_t_title,mf_s_title] := "'title'"
$      write_unlock rec
$      exit modl__success
$
$
$
```

```

$USER:
$   user = p2
$   full_name = p3
$   title = p4
$   address = p5
$   if user .eqs. "" then ask modl_ s "_User Name:"
$   if user .eqs. "" then user = modl_
$   if full_name .eqs. "" then ask modl_ s "_Full Name:"
$   if full_name .eqs. "" then full_name = modl_
$   if title .eqs. "" then ask modl_ s "_Title:"
$   if title .eqs. "" then title = modl_
$   if address .eqs. "" then ask modl_ s "_Mailing Address:"
$   if address .eqs. "" then address = modl_
$
$   user = f$edit(user,"TRIM,UPCASE")
$   if user .eqs. "SELF" then user = this_user
$   if f$length(user) .gt. mf_s_key then -
$       signal e userlen "User name ''user is too long."
$   if f$length(address) .gt. mf_s_address then -
$       signal e addrlen "Mailing address ''address is too long."
$
$   key = f$fao("#AS", mf_s_key, user)
$   read_nolock /key=""'key" rec /error=59
$   signal e useralrdyreg "User ''user is already registered."
$59:
$   rec = ""
$   rec[mf_t_key,mf_s_key]           := ""'key'"
$   rec[mf_t_time,mf_s_time]        := ""'f$time()'"
$   rec[mf_t_doer,mf_s_doer]        := ""'this_user'"
$   rec[mf_t_title,mf_s_title]      := ""'title'"
$   rec[mf_t_name,mf_s_name]        := ""'full_name'"
$   rec[mf_t_address,mf_s_address] := ""'address'"
$   write_unlock rec
$   exit modl__success
$
$! Module:      Process the REGISTER Command
$!
$! Synopsis:    This module processes the REGISTER command, which is
$!              used to register information about users and
$!              keywords. Duplicate users/keywords are not allowed.
$!
$!

```

```
$! Format:      REGISTER USER user-name "full name" "title" address
$!
$! Parameters: user-name:      The name of the user.
$!              full name:     The full name of the user.
$!              title:         The business title of the user.
$!              address:       The VMS mailing address of the user.
$!
$! Format:      REGISTER KEYWORD keyword "title"
$!
$! Parameters: keyword:        The name of the keyword.
$!              title:         The descriptive title for keyword.
$!
$! Notes:
```

File MODL_REMOVE.COM

```
$      on control_y then exit modl__ctrl_y
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      remove_keys = p1
$      users = p2
$      if remove_keys .eqs. "" then ask modl_ s "Keyword List:"
$      if remove_keys .eqs. "" then remove_keys = modl_
$      if users .eqs. "" then ask modl_ s "_User List:"
$      if users .eqs. "" then users = modl_
$      remove_keys = f$edit(remove_keys,"COLLAPSE,UPCASE")
$      users = f$edit(users,"COLLAPSE,UPCASE")
$
$      if users .eqs. "*"
$      then
$          read_lock /key="$"/match=ge rec /end_of_file=19
$10:      kl = f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
$          "TRIM")
$          modlcall remove_keywords modl_ "'kl'" "'remove_keys'"
$          rec[mf_t_keylist,mf_s_keylist] := "'modl_'"
$          update_unlock rec
$          read_lock rec /end_of_file=19
$          goto 10
$19:
$      else
$          i = -1
$20:      i = i + 1
$          username= f$element(i,",",users)
$          if username .eqs. "," then goto 29
$          if username .eqs. "SELF" then username = this_user
$          key = f$fao("!#AS", mf_s_key, username)
$          read_lock /key="'key'" rec /error=25
$          kl = f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
$          "TRIM")
$          modlcall remove_keywords modl_ "'kl'" "'remove_keys'"
$          rec[mf_t_keylist,mf_s_keylist] := "'modl_'"
$          update_unlock rec
$          goto 20
$25:      signal w unreguser "User "'username is not registered."
$          goto 20
$29:
$      endif
$      exit modl__success
$
```

```
$! Module:      Process the REMOVE Command
$!
$! Synopsis:   This module processes the REMOVE command, which is
$!             used to disassociate keywords from users.
$!
$! Format:     REMOVE keyword,...{user,... | *}
$!
$! Parameters: keyword:  A keyword to be disassociated from the
$!                   specified user(s).  A list is allowed.
$!                   user:   A registered user from whom the keyword(s)
$!                   are removed.  A list is allowed.  If an
$!                   asterisk (*) is specified, the keywords
$!                   are removed from all registered users.
$!
$! Notes:
```

File MODL_SEND.COM

```
$      status = modl__success
$      on control_y then goto control_y
$      on warning then goto error
$      check_open_file
$      dc$_mailbox = 160
$
$      file_spec = p1
$      selection = p2
$      subject = p3
$      copy_self = p4
$      batch = p5
$      if file_spec .eqs. "" then ask modl_ s "_File to Send:"
$      if file_spec .eqs. "" then file_spec = modl_
$      if selection .eqs. "" then ask modl_ s "_Selection:"
$      if selection .eqs. "" then selection = modl_
$      if subject .eqs. "" then ask modl_ s "_Subject:"
$      if subject .eqs. "" then subject = modl_
$      if copy_self .eqs. "" then copy_self = modl_copy_self
$      if batch .eqs. "" then batch = modl_batch
$      if f$getjpi("", "MODE") .eqs. "BATCH" then batch = false
$
$!     Parse the message file spec and make sure the file exists.
$!     Allow the spec to be NL: by checking for a mailbox device.
$
$      file_spec = f$parse(file_spec, "sys$disk:[] .txt;")
$      if f$getdvi(file_spec, "DEVCLASS") .ne. dc$_mailbox then -
$          if f$search(file_spec) .eqs. "" then -
$              signal e filnotfnd "File 'file_spec does not exist."
$
$      if batch
$      then
$!         Submit a batch job to send the message. First parse the
$!         selection expression to check for errors. Then create a
$!         little procedure that opens the current MODL file, sends
$!         the message, and exits. Submit the procedure so that it
$!         is deleted when done.
$
$      modlcall parse_selection modl_ '''selection'''
$      libcall unique_name modl_ sys$scratch:modl_?.tmp
$      temp_file = modl_
$      open/write modl_temp_file 'temp_file
$      write modl_temp_file "$ modl open ", -
$          f$trnlm("modl_opened_file")
$      write modl_temp_file "$ modl send ", file_spec, " """, -
$          selection, "" """, subject, "" """, copy_self
$      write modl_temp_file "$ exit"
```

```

$      close modl_temp_file
$      submit/identify/nolog/notify 'temp_file/delete
$
$
$      else
$!     Send the message immediately. Determine whether there is
$!     a mailing agent and call the appropriate subroutine. Then
$!     send a self-copy if requested.
$
$      signal i sendfile "File to be sent: ''file_spec'"
$      read_nolock_header hdr
$      agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),-
$                  "TRIM")
$      if agent .eqs. "" then gosub send_no_agent
$      if agent .nes. "" then gosub send_agent
$      if copy_self then mail 'file_spec 'this_user -
$                  /subject=""''subject (MODL self copy)"
$
$      endif
$      goto exit
$
$
$control_y:
$      status = modl__ctrl_y
$      goto exit
$error:
$      status = $status
$      goto exit
$exit:
$      set noon
$      close/nolog modl_temp_file
$      close/nolog modl_log_file
$      exit status .or. %x10000000
$
$
$!     This subroutine sends a message without using an agent.
$!     It does so by invoking MAIL once for each user, thus
$!     preventing one problem from messing up the other recipients.
$!     As it goes, it maintains a log file of the results, which
$!     is mailed back to the sender at the end.
$
$SEND_NO_AGENT:
$      modlcall parse_selection modl_selection ""'selection'"
$      libcall unique_name modl_sys$scratch:modl_?.log;
$      log_file = modl_
$      open/write modl_log_file 'log_file
$      fao = "!12AS !20AS !40AS"
$      write modl_log_file -
$          f$fao(fao, "User", "Mailing Address", "Status")

```

```

$ write modl_log_file -
    f$fao(fao, "-----", "-----", "-----")
$ set message sys$message:cliutlmsg
$
$ read_nolock /key="$"/match=ge rec /end_of_file=59
$50: kl = "," + -
    f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
        "TRIM") + ","
$ if 'modl_selection
$ then
$ user = f$edit(f$extract(mf_t_key,mf_s_key,rec),"TRIM")
$ address = f$edit(f$extr(mf_t_address,mf_s_address,rec),-
    "TRIM")
$
$ set noon
$ mail 'file_spec 'address /subject=""'subject'
$ message = f$message($status)
$ set on
$ write modl_log_file f$fao(fao, user, address, -
    f$edit(f$element(1, ",", message), "TRIM"))
$ endif
$55: read_nolock rec /end_of_file=59
$ goto 50
$59:
$ close modl_log_file
$ send_log = true
$ if f$getjpi("","MODE") .eqs. "INTERACTIVE"
$ then
$ ask modl_b "Do you want a copy of the mailing log?" no
$ send_log = modl_
$ endif
$ if send_log then mail 'log_file 'this_user -
    /subject=""'subject (MODL mailing log)"
$ delete 'log_file'*
$ return
$
$
$! This subroutine sends a message using an agent. It creates
$! a MAIL distribution list with the selected users and then
$! sends the message to them with one invocation of MAIL.
$
$SEND_AGENT:
$ libcall unique_name modl_ sys$scratch:modl_?.dis;
$ temp_dis = modl_
$ @modl_system:modl_list ''selection' 'temp_dis -
    "Temporary list for sending by agent."
$ mail 'file_spec "@''temp_dis'" /subject=""'subject'"

```

```

$      delete 'temp_dis'*
$      return
$
$! Module:      Process the SEND Command
$!
$! Synopsis:    This module processes the SEND command, which is
$!              used to send a message to users who satisfy a given
$!              selection expression.  There are four modes for
$!              sending the message:
$!
$!              Interactive, no agent:  The message is sent while
$!              the user waits.  A separate invocation of MAIL is
$!              used for each recipient.  Errors appear on the
$!              terminal and a log file can be saved.
$!              Interactive, agent:  The message is sent while the
$!              user waits.  A distribution list is created and a
$!              single invocation of MAIL is used.  Errors appear
$!              on the terminal.
$!              Batch, no agent:  A batch job is submitted to send
$!              the mail.  In batch, a separate invocation of MAIL
$!              is used for each recipient.  An error log is
$!              mailed back to the sender.
$!              Batch, agent:  A batch job is submitted to send the
$!              mail.  In batch, a distribution list is created and
$!              a single invocation of MAIL is used.  Errors are
$!              reported by the agent.
$!
$! Format:      SEND message-file "selection" "subject"
$!
$! Parameters:  message-file:  The file spec of the text file with
$!                          the message to be sent.
$!              selection:      The selection expression specifying
$!                          the users to receive the message.
$!              subject:        The subject heading for the message.
$!
$! Notes:      See the User's Guide for a description of the
$!              selection expression.

```

File MODL_SET.COM

```
$      on control_y then exit modl__ctrl
$      on warning then exit $status .or. %x1000000
$      check_open_file
$
$      what = p1
$      if what .eqs. "" then ask modl_ s -
$                               "_What (AGENT, NOAGENT, SEND):"
$      if what .eqs. "" then what = modl_
$      libcall lookup_keyword modl_ ""'what'" AGENT,NOAGENT,SEND
$      if modl_ .eqs. "" then signal e unkwhat -
$                               "The item ''what is unknown or ambiguous."
$      goto 'modl_
$
$
$
$AGENT:
$      agent = p2
$      read_lock_header hdr
$      old_agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),-
$                          "TRIM")
$      if agent .eqs. "" then ask modl_ s "_Agent:" ""'old_agent'"
$      if agent .eqs. "" then agent = modl_
$      hdr[mf_t_agent,mf_s_agent] := ""'agent'"
$      update_unlock_hdr
$      exit modl__success
$
$
$NOAGENT:
$      read_lock_header hdr
$      hdr[mf_t_agent,mf_s_agent] := ""
$      update_unlock_hdr
$      exit modl__success
$
$
$SEND:
$      copy_self = p2
$      batch = p3
$      if copy_self .eqs. "" then ask modl_ b "_Copy Self?" -
$                               'modl_copy_self
$      if copy_self .eqs. "" then copy_self = modl_
$      if batch .eqs. "" then ask modl_ b "_Send in Batch?" -
$                               'modl_batch
$      if batch .eqs. "" then batch = modl_
$      modl_copy_self == copy_self
$      modl_batch == batch
$      exit modl__success
$
```

```

$! Module:      Process the SET Command
$!
$! Synopsis:    This module processes the SET command, which is
$!              used to specify various parameters for MODL.
$!
$!              The SET AGENT command specifies the mailing agent
$!              for future SEND commands. The agent is stored in
$!              the MODL distribution file. The SET NOAGENT command
$!              specifies that no mailing agent is used.
$!
$!              The SET SEND command specifies various things about
$!              how mail messages are sent. These things are
$!              specific to the current user and are only remembered
$!              for the remainder of the terminal session.
$!
$! Format:      SET AGENT  "string"
$!              SET NOAGENT
$!
$! Parameters:  string:   The string that identifies the mailing
$!                      agent (e.g., "zp%"). This string is
$!                      prefixed on each mailing address before
$!                      mail is sent to it.
$!
$! Format:      SET SEND copy-self? batch?
$!
$! Parameters:  copy-self?:  A boolean specifying whether an
$!                          additional copy of a message should
$!                          be mailed to the sender.
$!              batch?:      A boolean specifying whether mailing
$!                          should be done with a batch job.
$!
$! Notes:

```

File MODL_SHOW.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$      check_open_file
$
$      what = p1
$      if what .eqs. "" then ask modl_ s -
$                               "_What (FILE, KEYWORD, SEND, USER):"
$      if what .eqs. "" then what = modl_
$
$      libcall lookup_keyword modl_ "'what'" -
$                               FILE,KEYWORD,SEND,USER
$      if modl_ .eqs. "" then signal e unkwhat -
$                               "The item 'what' is unknown or ambiguous."
$      goto 'modl_
$
$
$
$FILE:
$      read_nolock_header hdr
$      doer = f$edit(f$extract(mf_t_doer,mf_s_doer,hdr),"TRIM")
$      time = f$edit(f$extract(mf_t_time,mf_s_time,hdr),"TRIM")
$      title = f$edit(f$extract(mf_t_title,mf_s_title,hdr),"TRIM")
$      agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),"TRIM")
$      if agent .eqs. "" then agent = "(none)"
$      display ""
$      display "MODL file:  ", f$strnlm("modl_opened_file")
$      display "Title:      ", title
$      display "Agent:       ", agent
$      display "(Created by: ", doer, ", on: ", time, ")"
$      display ""
$      exit modl__success
$
$
$KEYWORD:
$      keyword = p2
$      if keyword .eqs. "" then ask modl_ s "_Keyword or *:"
$      if keyword .eqs. "" then keyword = modl_
$      keyword = f$edit(keyword,"TRIM,UPCASE")
$      if keyword .eqs. "*"
$      then
$          count = 0
$          read_nolock /key="#" /match=ge rec /end_of_file=19
$10:      if f$extract(mf_t_key,1,rec) .nes. "#" then goto 19
$          gosub display_keyword
$          count = count + 1
$          read_nolock rec /end_of_file=19
$          goto 10
$19:
```

```

$      display f$fao("!/Keywords displayed: !UL!/", count)
$      else
$      key = f$fao("!#<#!AS!>", mf_s_key, keyword)
$      read_nolock /key="''key'" rec /error=25
$      gosub display_keyword
$      display ""
$      goto 29
$25:      signal e keynotdef "Keyword ''keyword is not defined." -
          '$status
$29:
$      endif
$      exit modl__success
$
$display_keyword:
$      keyword = f$extract(mf_t_key,mf_s_key,rec) - "#"
$      doer = f$edit(f$extract(mf_t_doer,mf_s_doer,rec),"TRIM")
$      time = f$edit(f$extract(mf_t_time,mf_s_time,rec),"TRIM")
$      title = f$edit(f$extract(mf_t_title,mf_s_title,rec),"TRIM")
$      display ""
$      display "Keyword:      ", keyword
$      display "Title:        ", title
$      display "(Defined by: ", doer, ", on: ", time, ")"
$      return
$
$
$SEND:
$      read_nolock_header hdr
$      agent = f$edit(f$extract(mf_t_agent,mf_s_agent,hdr),"TRIM")
$      if agent .eqs. "" then agent = "(none)"
$      display ""
$      display "Agent:        ", agent
$      display "Copy Self: ", -
          f$element(modl_copy_self,"|","NO|YES")
$      display "In Batch:  ", f$element(modl_batch,"|","NO|YES")
$      display ""
$      exit modl__success
$
$
$USER:
$      user = p2
$      if user .eqs. "" then ask modl_ s "_User Name or *:" -
          'this_user
$
$      if user .eqs. "" then user = modl_
$      user = f$edit(user,"TRIM,UPCASE")
$      if user .eqs. "*"
$      then

```

```

$          count = 0
$          read_nolock /key="$"/match=ge rec /end_of_file=39
$30:       gosub display_user
$          count = count + 1
$          read_nolock rec /end_of_file=39
$          goto 30
$39:
$          display f$fao("!/Users displayed: !UL!/", count)
$          else
$          if user .eqs. "SELF" then user = this_user
$          key = f$fao("!#AS", mf_s_key, user)
$          read_nolock /key="''key'" rec /error=45
$          gosub display_user
$          display ""
$          goto 49
$45:       signal e usernotreg "User ''user is not registered." -
           '$status
$49:
$          endif
$          exit modl__success
$
$display_user:
$          user = f$extract(mf_t_key,mf_s_key,rec)
$          doer = f$edit(f$extract(mf_t_doer,mf_s_doer,rec),"TRIM")
$          time = f$edit(f$extract(mf_t_time,mf_s_time,rec),"TRIM")
$          title = f$edit(f$extract(mf_t_title,mf_s_title,rec),"TRIM")
$          full_name = f$edit(f$extract(mf_t_name,mf_s_name,rec),-
           "TRIM")
$          address = f$edit(f$extract(mf_t_address,mf_s_address,rec),-
           "TRIM")
$          keylist = f$edit(f$extract(mf_t_keylist,mf_s_keylist,rec),-
           "TRIM")
$          display ""
$          display "User:           ", user
$          display "Full name:      ", full_name
$          display "Title:         ", title
$          display "Mailing address: ", address
$          if keylist .nes. "" then -
$          display "Keyword list:   ", keylist
$          display "(Registered by:  ", doer, ", on: ", time, ")"
$          return
$
$! Module:      Process the SHOW Command
$!
$! Synopsis:    This module processes the SHOW command, which is
$!              used to display the contents of a MODL distribution
$!              file and various other environmental items.
$!

```

\$! The SHOW FILE command is used to display information
\$! about the current MODL distribution file. The
\$! information includes the full file spec, title, and
\$! mailing agent.
\$!

\$! The SHOW KEYWORD and SHOW USER commands are used to
\$! display information about registered keywords and
\$! users. The information includes the title in both
\$! cases, and the full name, mailing address, and
\$! keyword list for a user.
\$!

\$! The SHOW SEND command is used to display the mailing
\$! parameters established with the SET SEND command.
\$! The information includes the send-self and batch
\$! parameters.
\$!

\$! Format: SHOW FILE
\$!

\$! Format: SHOW USER{user | *}
\$!

\$! Parameters: user: The name of a registered user, or an
\$! asterisk (*) to display all users.
\$!

\$! Format: SHOW KEYWORD{keyword | *}
\$!

\$! Parameters: keyword: The name of a registered keyword, or an
\$! asterisk to display all keywords.
\$!

\$! Format: SHOW SEND
\$!

\$! Notes:

File MODL_SUBROUTINES.COM

```
$      on control_y then exit modl__ctrly
$      on warning then exit $status .or. %x10000000
$
$      goto 'p1
$
$
$! Title:      Parse a Selection Expression
$!
$! Synopsis:   This subroutine parses a selection expression and
$!             builds the corresponding DCL boolean expression.
$!             Each keyword in the expression turns into:
$!             (F$LOC(",keyword,",KL) .NE. F$LEN(KL))
$!             under the assumption that the list of keywords for a
$!             particular user is stored in the symbol KL.
$!             Each boolean operator in the keyword is converted to
$!             a DCL operator by surrounding it with dots.
$!             Parentheses are simply copied to the DCL expression.
$!
$! Parameters: P2: The symbol to receive the resulting DCL
$!             expression.
$!             P3: The selection expression to be parsed.
$!
$! Returns:    The DCL expression representing the selection
$!             expression.
$
$PARSE_SELECTION:
$
$      selection = f$edit(p3,"COMPRESS,TRIM,UPCASE") + "~"
$      if selection .eqs. "*"~"
$      then
$          'p2 == "true"
$          exit modl__success
$      endif
$
$      result = ""
$      i = 0
$ps10:   char = f$extract(i,1,selection)
$      if f$locate(char," ()~") .ne. 4
$      then
$          if char .eqs. "~" then goto ps19
$          result = result + char
$          i = i + 1
$      else
$          keyword = ""
$ps20:   keyword = keyword + char
$          i = i + 1
```

```

$          char = f$extract(i,1,selection)
$          if f$locate(char," ()~") .eq. 4 then goto ps20
$          if keyword .eqs. "AND" .or. keyword .eqs. "OR" .or. -
$             keyword .eqs. "NOT"
$          then
$             result = f$fao("!AS.!AS.", result, keyword)
$          else
$             gosub check_keyword
$             result = -
$                 f$fao("!AS(f$loc("",!AS,"",kl) .ne. f$len(kl))",-
$                     result, keyword)
$             endif
$          endif
$          goto ps10
$ps19:
$
$          'p2 == result
$          exit modl__success
$
$check_keyword:
$          key = f$fao("!#<#!AS!>", mf_s_key, keyword)
$          read_nolock /key="''key'" rec /error=ps95
$          return modl__success
$ps95: signal e undefkey "Keyword ''keyword is not defined."

```

```

$! Title:      Remove Keywords from a List
$!
$! Synopsis:   This subroutine removes one or more keywords from an
$!             original list of keywords.
$!
$! Parameters: P2: The symbol to receive the resulting list.
$!             P3: The original comma-separated list of keywords.
$!             P4: The list of keywords to be removed.
$!
$! Returns:    The original list with specified keywords removed.
$
$REMOVE_KEYWORDS:
$
$      original = "," + p3 + ","
$      k = -1
$rk10:  k = k + 1
$      keyword = f$element(k, ",", p4)
$      if keyword .eqs. "," then goto rk19
$      l = f$locate(","+keyword+",", original)
$      if l .ne. f$length(original) then -
$          original = f$extract(0, l, original) + -
$                  f$extract(l+f$length(keyword)+1,9999,original)
$      goto rk10
$rk19:
$      if f$extract(0,1,original) .eqs. "," then -
$          original = original - ","
$      'p2 == f$extract(0,f$length(original)-1, original)
$      return modl__success

```

File MODL_FILE.FDL

TITLE	"MODL File Description"	
IDENT	"18-APR-1988 12:36:23	VAX-11 FDL Editor"
SYSTEM		
	SOURCE	VAX/VMS
FILE		
	ORGANIZATION	indexed
RECORD		
	CARRIAGE_CONTROL	carriage_return
	FORMAT	variable
	SIZE	400
AREA 0		
	ALLOCATION	35
	BEST_TRY_CONTIGUOUS	yes
	BUCKET_SIZE	2
	EXTENSION	8
AREA 1		
	ALLOCATION	4
	BEST_TRY_CONTIGUOUS	yes
	BUCKET_SIZE	2
	EXTENSION	2
KEY 0		
	CHANGES	no
	DATA_AREA	0
	DATA_FILL	100
	DATA_KEY_COMPRESSION	yes
	DATA_RECORD_COMPRESSION	yes
	DUPLICATES	no
	INDEX_AREA	1
	INDEX_COMPRESSION	no
	INDEX_FILL	100
	LEVEL1_INDEX_AREA	1
	NAME	"Record Key"
	PROLOG	3
	SEGO_LENGTH	16
	SEGO_POSITION	0
	TYPE	string

Glossary

- Absolute time.** A string representing an exact point in time. The full VMS format is:
dd-mon-yyyy hh:mm:ss.cc.
- Access category.** A level at which a process accesses a file or other object. The four levels are system, owner, group, and world.
- Access control entry (ACE).** An individual entry in an access control list.
- Access control list (ACL).** A list of access control entries that specify detailed protection information for VMS objects.
- Access mode.** The VAX privilege mode at which a process runs or makes requests. The four access modes are user, supervisor, executive, and kernel.
- Allocation.** See Device allocation.
- Alternate key.** A secondary key field in an indexed file.
- Ampersand substitution.** An explicit form of substitution in a DCL command, which is specified by the ampersand (&).
- Apostrophe substitution.** An explicit form of substitution in a DCL command, which is specified by the apostrophe (').
- Append.** See Open for append.
- Argument.** A value that serves as input data for a particular operation. For example, a lexical function may require one or more arguments to specify the operation it is to perform. See also Optional argument.
- Arity.** The number of operands required by an operator.
- Array.** A collection of data items stored together under one name. Associated with each item is an index by which the item is identified from among all other items.

Array index. A number or character string identifying a particular element of an array.

ASCII character set. The character set used by the VAX computer. ASCII stands for American Standard Code for Information Interchange.

Assignment command. A DCL command that assigns data to a symbol.

Authorized privilege. A privilege that a user is allowed to enable. The system manager determines the authorized privilege for each user. Most users are authorized for the NETMBX and TMPMBX privileges.

Automatic substitution. See Personal command substitution.

Batch job. A detached process created by VMS to run a procedure submitted to a batch queue. A batch job runs without an interactive user at a terminal.

Batch process. A process created by VMS to run programs in a batch job. See also Interactive process, Network process.

Binary operator. An operator that requires two operands (its arity is 2).

Bit. Short for “binary digit.” A single digit of binary information, representing a 0 or 1.

Boolean. A data type that represents the two logical values true and false.

Call. See Invoke.

Captive account. A VMS user account that “captures” users as they log in. A user is forced to execute the LOGIN.COM procedure and cannot interrupt it with <CTRL/y>. Once the login procedure is complete, the user is automatically logged out.

Carriage return carriage-control. A file attribute specifying that each record in the file is assumed to have a carriage return/line feed pair at the end. This attribute affects how records are displayed and printed.

Case statement. A flow-of-control construct that chooses one of several possible execution paths based on the value of a symbol. There is no case statement in DCL, but it can be simulated using the GOTO command and substitution.

Character set. The collection of printable and control characters used by a computer or other device. The VAX uses the ASCII character set.

Character string. A data type that represents a sequence of characters in some character set. In DCL, the character set is ASCII. A character string may contain zero, one, or more characters.

Child process. A subprocess owned by some other process, which is called the parent process.

Cleanup. See Procedure cleanup.

Combination time. A string representing the combination of an absolute time and a delta time. A combination time represents a particular point some time before or after another point in time. The full VMS format is:
dd-mon-yyyy hh:mm:ss.cdddd-hh:mm:ss.cc.

- Command.** See Assignment command, DCL command, Flow-of-control command, Personal command.
- Command language interpreter.** A system program that can analyze and perform commands entered by the user. The standard command language interpreter for VMS is the Digital Command Language (DCL).
- Command parameter.** An item of information included in a command, which specifies an object that the command should manipulate. Command parameters are separated from the command verb and from one another by spaces.
- Command procedure.** See DCL command procedure.
- Command qualifier.** A keyword included in a DCL command to specify an option or modify the standard behavior of the command. Qualifier keywords are preceded by a slash (/).
- Command syntax.** The form of a command. Command syntax includes the verb, parameter order, qualifiers, and the punctuation used to connect them.
- Command verb.** A word identifying a particular command to be performed. DCL includes verbs such as COPY and WRITE.
- Comment.** A phrase or sentence included in a program to make it more readable by human beings. Comments are completely ignored by the computer. In DCL, comments begin with an exclamation point (!) and extend to the end of the line.
- Comparison time.** An absolute time that has been reformatted so that two times can be compared as if they were character strings. The full VMS format is:
yyyy-mm-dd hh:mm:ss.cc.
- Condition.** The boolean expression in an IF statement, which determines whether the “then” statements or the “else” statements are executed.
- Context symbol.** A DCL symbol that keeps track of the state of an ongoing request such as that performed with the F\$PID lexical function. The F\$PID function obtains the process identifiers of the system processes, one on each call. The context symbol keeps track of the last process examined between each call to the function.
- Control character.** A character used to control the terminal or other device to which it is sent. For example, the linefeed character causes the cursor on a terminal to move down to the next line. Contrast with Printable character.
- Control sequence.** A sequence of characters sent to a terminal in order to perform a control or formatting action other than displaying a character. For example, most terminals accept a control sequence that clears the screen.
- Control string.** The character string that specifies how data is to be formatted by the F\$FA0 lexical function.
- Current directory.** See Working directory.
- Data.** The collection of information created and manipulated by a program.
- Data item.** An individual item of information, such as an integer or a character string.

Data type. The class of information to which a data item belongs. DCL supports integer and character string types.

Date. See Time.

DCL code. The DCL commands that make up a DCL procedure.

DCL command. A sequence of words and punctuation marks that instructs DCL to perform some operation. A command consists of a verb, parameters, and qualifiers.

DCL command procedure. A file containing DCL commands, which can be executed by DCL in place of commands entered at the terminal.

DCL environment. A collection of information that affects the way DCL operates. Environment items include the working directory and the state of procedure verification. This information can be inspected and altered by the DCL user.

DCL procedure. See DCL command procedure.

DCL prompt. A character string displayed at the terminal to signal that the user may enter a new command. The DCL prompt is the dollar sign (\$) unless the user changes it.

Default answer. The answer to a question, which is automatically provided when the user does not enter any other answer. It is a VMS convention to include the default answer in square brackets ([]) after the question.

Default directory. The official VMS term for Working directory.

Default value. A value used by a program when no value is explicitly specified.

Delta time. A string representing a certain amount of time, or the difference between two absolute times. The full VMS format is: *ddd-hh:mm:ss.cc*.

Detached process. A main process that is not owned by any other process. VMS creates a detached process for an interactive user, a batch job, or a remote network request.

Device. A hardware component, attached to a computer, which provides input or output capability. Examples are terminals, disk drives, tape drives, and communications controllers.

Device allocation. The act of reserving a device for exclusive use by a process. This is accomplished with the ALLOCATE command.

Device name. A character string that uniquely identifies a particular device from among all devices accessible to a system. The device name includes components identifying the system to which the device is connected, the type of device, its controller letter, and its device number.

Directory. A VMS file that acts as a catalog for other files. A directory contains a list of data files and other directories, which are considered to reside in that directory. Using directories, a hierarchical structure is imposed on all the files on a device. See also Login directory, Master file directory, Root directory, Working directory.

Distribution list. A text file containing a list of user names. Distribution lists are used by the MAIL utility to send messages to groups of users.

- End-of-file.** The state of an input file in which all records have been read. In the case of a disk file, the next read operation will return a “no more records” status.
- Equivalence string.** The value associated with a logical name.
- Error handler.** The portion of a DCL procedure that deals with errors detected during the execution of the procedure.
- Escape sequence.** See Control sequence.
- Evaluate.** To determine the final value of an expression. The expression’s operators and operands are combined in a predefined order to produce a new value.
- Execute.** A term often applied to programs, including DCL procedures. A program is executed when a computer interprets the statements (commands) in the program and carries out the actions specified by those statements.
- Expression.** A combination of operators and operands that specifies a new value to be computed. For example, $A + B$ produces a new integer by adding the values of the symbols A and B.
- Field.** An individual data item in a record or structure. The fields in a structure can be manipulated as separate items.
- File.** A collection of data arranged in the form of records. Each record contains a related portion of the data, and all the records taken together make up the entire file. See also Log file, Process-permanent file, Sequential file, Temporary file, Text file.
- File attributes.** A collection of information associated with a file, which identifies the file, describes its internal format, and specifies its file protection.
- File prologue.** The portion of an indexed file that contains descriptions of the record keys and other aspects of the file.
- File protection.** The scheme used by VMS to safeguard files against unauthorized access or modification.
- File sharing.** An RMS facility, which allows multiple processes to access and modify the same file at the same time.
- File spec.** Short for “file specification.”
- File specification.** A character string identifying a particular file or a group of files.
- Fixed-length record.** A record in a sequential or indexed file whose design stipulates that every record in the file be the same size. Such records are usually formatted according to one or more predetermined record structures.
- Flow-of-control command.** A DCL command that enables a procedure to alter its flow of execution. Such commands include **IF** and **GOTO**. Flow-of-control commands provide the power required to make DCL a general-purpose programming language.
- Generic match.** An indexed file key lookup based on a partial key rather than on the whole key. For example, the generic key "K" will match any record whose key begins with the letter K.

Global symbol. A symbol created at the global symbol level using the double equal sign (==) assignment statement. Global symbols can be accessed and created from any procedure level.

Glossary. A list of terms and their definitions.

Group number. The portion of a UIC that identifies a related group of users. See also Member number.

Home directory. See Login directory.

Image data. Lines in a command procedure that do not contain DCL commands but rather contain input data for a program (image). This input data is used in lieu of data read from the terminal. Any procedure lines not beginning with a dollar sign (\$) are considered image data.

Index. In DCL, an identifier used to select an individual character in a character string or to identify an item in an array. See also Array index, String index.

Infinite loop. A loop in the flow of control of a program that, once initiated, can never be terminated. Infinite loops are caused by loops with no termination checks or faulty checks.

Initialization. See Volume initialization.

Integer. A data type that represents whole numbers—negative, zero, or positive. In DCL, an integer is represented as a 32-bit (longword) quantity, allowing the integer to range from -2,147,483,648 to +2,147,483,647.

Interactive process. A VMS process created to perform work for an interactive user logged in at a terminal. See also Batch process, Network process.

Interrupt. A break in execution caused by a system event, which requires immediate attention when it occurs. If a program is executing when an interrupt takes place, program execution must be suspended so that the interrupt can be handled as soon as possible. The only interrupt handled by a DCL procedure is the one caused by the <CTRL/y> key.

Invoke. To begin the execution of a command procedure or other sequence of command.

Iteration. A single execution of a loop body; one cycle through a loop.

Iterative translation. The process by which a logical name is successively translated until a final equivalence string is produced.

Job. A detached process and all its subprocesses. The term *job* is also used to refer to a batch job.

Job controller. The VMS program that maintains the batch and print queues.

Job logical name table. A logical name table associated with a job. All the processes in the job can access the table.

Job queue. A batch or print queue maintained by the job controller. Jobs are submitted to a batch queue with the SUBMIT command. Files are submitted to a print queue with the PRINT command.

Key field. In an indexed file record, a field whose values are to be indexed for fast lookup. Every indexed file has a primary key field and may have additional alternate keys. Key fields are usually referred to simply as keys.

Keyword. A character string chosen from among a fixed set of valid strings, used to represent a particular function or option. Keywords are used in some lexical functions to define the exact action taken by the function.

Label. A name given to a command in a DCL procedure. The name allows the GOTO command to alter the sequential flow of execution by specifying the named command as the next one to execute.

Length. See String length.

Lexical function. A built-in DCL subroutine that can be used as an operand in an expression. Lexical functions compute all kinds of values, from the current time to the name of the local DECnet node.

List. A sequence of items separated by some delimiter character, often a comma (,).

Literal. A sequence of characters that represents a constant data value, such as 0, -42, or "my name". A literal specifies a constant value, while a variable stands for a changing value.

Loading. See Volume loading.

Log file. A file created by a batch job to contain the output of the job. Because there is no terminal on which to display results, the results are sent to the log file.

Logical name. A named entity that stands for all or part of a file specification or other value. Logical names are maintained in logical name tables.

Logical name table. A table of logical names and their values. Each process has its own logical name table. There are also logical name tables available to more than one process.

Logical name table directory. A system data structure that specifies a set of logical name tables. There is one directory for each process and one for the system as a whole.

Logical name translation. The process by which a logical name is replaced with its equivalence string.

Login directory. The directory established as the working directory when a user first logs in. The login directory is usually the root of all the user's directories.

Longword. The VAX term for a collection of 32 bits. A longword is the amount of memory occupied by a standard VAX integer.

Loop. A sequence of commands that is executed repeatedly until some termination condition occurs. The sequence of commands is called the loop body. See also Infinite loop.

Loop body. The sequence of commands that is repeatedly executed in a loop.

- Main procedure.** The command procedure in a DCL application where execution begins when the application is invoked. The main procedure is the first procedure to be executed; it may invoke other subprocedures in turn.
- Master file directory.** The top-level directory on a disk volume. All system and user directories are accessed through the master file directory.
- Member number.** The portion of a UIC that identifies a particular member within a user group. See also Group number.
- Message file.** A special kind of image file, which contains the codes and text for VMS messages.
- Metalinguistic symbol.** A symbolic name that stands for some information to be included in a command or other instruction to a computer. For example, in `DELETE file`, the word *file* is a metalinguistic symbol standing for any valid file specification.
- Mounting.** See Volume mounting.
- Network process.** A process created by VMS to perform a task for a remote network node. See also Batch process, Interactive process.
- Null device.** A nonphysical device that serves as a “black hole” for data. All output to the null device is discarded. An attempt to read from the null device results in an immediate end-of-file condition. The null device on VMS is named `NL:`.
- Null string.** A character string composed of no characters. It has a length of zero.
- Null value.** A special value stored in an alternate key field of an indexed file. The value specifies that the field is empty and should not be indexed.
- Open a file.** To identify and prepare a specific file for input/output operations. A file must be opened before it can be read or written.
- Open for append.** To open a sequential file so that new records can be added to the end of the file.
- Operand.** A literal, symbol, or other data value used in an expression.
- Operator.** A character or character sequence that specifies a particular kind of data manipulation in an expression (e.g., `+` for addition).
- Operator precedence.** A number that determines the order in which operators are applied to operands. An operator with a higher precedence is evaluated before one with a lower precedence.
- Optional argument.** A lexical function argument that may be included in the function call, but need not be. If an optional argument is omitted, the function provides a common default value.
- Optional parameter.** A command or procedure parameter that may be included in the command line but need not be. If an optional parameter is omitted, a common default is provided.

- Owner UIC.** The user identification code (UIC) of the person who owns a particular object such as a file. The owner UIC is used to determine whether a particular user has access to the object.
- P0 space.** The region of process memory in which VMS utilities and application programs are executed.
- P1 space.** The region of process memory in which the DCL program is executed.
- Parameter.** See Command parameter, Optional parameter, Procedure parameter, Product parameter, Required parameter.
- Parent process.** A process that owns one or more subprocesses, which are called the child processes.
- Parsing a file spec.** The process through which a partial file spec entered by the user is transformed into a complete file spec with all elements filled in. A file spec must be parsed, either explicitly by a program or implicitly by VMS, before it can be used to access files.
- Pattern.** A character string that acts as a template or example for some kind of matching operation. For example, the `SEARCH` command requires one or more patterns to tell it which records are to be matched.
- Personal command.** A symbol set to a particular combination of DCL command verb, parameters, and qualifiers. When used as the initial symbol in a command, DCL performs personal command substitution to replace the symbol with its value.
- Personal command substitution.** An implicit form of substitution in which DCL automatically replaces the initial symbol in a command with its value.
- PID.** See Process identifier.
- Precedence.** See Operator precedence.
- Primary key.** The main key in an indexed file record. Every indexed file has a primary key.
- Printable character.** A character that can be displayed on a terminal or printed on a printer. Each printable character has a glyph, which represents it visually. Contrast with Control character.
- Procedure.** See DCL command procedure, Main procedure.
- Procedure call unwind.** The mechanism by which a subprocedure unwinds all active procedure calls so that the main procedure is terminated. During the unwind, the cleanup code of all active procedures is executed.
- Procedure cleanup.** The procedure termination code executed to close open files, delete temporary files, or otherwise restore the DCL environment to its normal state.
- Procedure exit.** The termination of a procedure, which occurs when there are no more commands to execute or when an `EXIT` command is encountered.
- Procedure level.** A level at which DCL reads and executes commands from a procedure. The procedure invoked at prompt level runs at procedure level 1. A subprocedure invoked by that procedure runs at procedure level 2, and so on.

- Procedure parameter.** A value received by a procedure, used to control the procedure or provide data for it to operate upon. A DCL subprocedure or CALL subroutine can receive up to eight parameters.
- Process.** The VMS environment in which programs are run for users. See also Batch process, Detached process, Interactive process, Network process.
- Process identifier.** An integer that uniquely identifies a process on a single VAX or a VAXcluster. A process identifier is often called a PID.
- Process mode.** A keyword identifying the type of a process: interactive, batch, network, or subprocess.
- Process-permanent file.** A file opened by DCL, which remains open until it is explicitly closed. Two well-known examples are SYS\$INPUT and SYS\$OUTPUT.
- Process privilege.** A privilege that a particular process has enabled. A process can only enable privileges listed in the user's authorized privileges.
- Product parameter.** An item of information required by a software product for its normal operation. Product parameters are often specified with logical names.
- Prompt level.** The level at which DCL accepts commands from the terminal by issuing a prompt and waiting for user input.
- Protection mask.** The 16 indicators that determine which of the four access categories (system, owner, group, world) are allowed which types of access to a file (read, write, execute, delete).
- Qualifier.** See Command qualifier.
- Record.** A group of related data fields, which a program treats as a unit. A file is composed of a collection of records: individual records are read from and written to a file. This term is often used interchangeably with Structure. See also Fixed-length record, Variable-length record.
- Record locking.** The mechanism by which RMS implements file sharing. In simple terms, a record is locked when it is read and remains locked until it is updated or until another record is read.
- Record Management System (RMS).** The subsystem of VMS that handles data files, both sequential and indexed.
- Record structure.** The format of the records in a file; the layout of the fields in the record.
- Required parameter.** A command or procedure parameter that must be specified when the command is entered.
- Restartable batch job.** A batch job that can be restarted if the system fails during its execution. A restarted job can determine that it was in fact restarted and change its behavior accordingly.
- Resultant file spec.** The expanded file spec returned by the F\$PARSE lexical function.

Return point. The command in a procedure at which execution continues after a subroutine is invoked and returns. When a subroutine is invoked, DCL remembers the position of the next command in the procedure, the return point. When the subroutine completes, DCL continues execution at the return point.

Return value. The data item that is the result of a lexical function.

Rights identifier. A name identifying a particular user, category of user, or application. Rights identifiers are associated with users and are needed to determine whether a user has access to a system object, such as a file or application.

Root directory. A directory whose fundamental purpose is to act as the parent for a related set of subdirectories and the files contained in them.

Run. See Execute.

Scheduler. The VMS facility that periodically selects a process to execute on the VAX. The scheduler guarantees that all processes get a chance to run.

Search list. A logical name that has more than one equivalence string. The strings specify directories to be searched, in order, for specified files.

Search operation. A file operation in which a parsed file spec is used to locate one or more files. The search succeeds if at least one file matches the file spec; it fails otherwise.

Sequential execution. The default method of command procedure execution, in which DCL executes commands sequentially from beginning to end. Sequential execution can be altered with flow-of-control commands.

Sequential file. A file composed of a sequence of individual records. The records can only be written or read in sequential order.

Signal an error. The means by which a program or procedure notifies the user that an error has occurred. A signaled error often results in a message being displayed at the terminal.

Statement. A command or instruction in a programming language. In DCL programming, "statement" is often used in place of the more common "command."

Status code. The means by which VMS returns an indication of the success or failure of a subroutine, program, or entire application.

Stream ID. An integer that identifies a file search operation performed by the F\$SEARCH lexical function. Multiple stream IDs are needed if more than one search operation takes place simultaneously.

String. See Character string.

String index. The number assigned to each character in a string. The first character is index 0, the second 1, and so forth. The string index allows a character string to be treated as an array of characters.

String length. The number of characters in a character string.

- Structure.** A data aggregate that consists of multiple fields organized into one collective data item. The entire structure can be manipulated as one data item, and the fields can be accessed individually. The fields need not all be of the same data type.
- Subdirectory.** A file directory subordinate to another directory. A subdirectory is an ordinary directory, but the prefix "sub" clarifies its relationship to the parent directory.
- Subprocedure.** A command procedure invoked by another procedure as a subroutine.
- Subprocess.** A process created and owned by another VMS process. The SPAWN command can be used to create a subprocess from DCL.
- Subroutine.** A portion of a program that stands by itself and can be called by other parts of the program. In DCL, there are three kinds of subroutines: subprocedures invoked with the at-sign (@) command, CALL subroutines, and GOSUB subroutines.
- Subscript (array).** See Array index.
- Substitution.** The alteration of a DCL command by replacing a symbol or lexical function with its value. See also Ampersand substitution, Apostrophe substitution, Personal command substitution.
- Substring.** A portion of an existing character string. For example, "00 BA" is a substring of "FOO BAR".
- Symbol.** A named entity used as a variable in a DCL procedure. A symbol is created and assigned a value with an assignment command.
- Symbol level.** The command level at which a symbol is created. The levels are global level, DCL prompt level, procedure level 1, procedure level 2, and so on.
- Symbol name.** The sequence of characters that names a DCL symbol.
- Symbol table.** The data structure in which DCL stores symbols and their values. The overall symbol table is organized into levels, each with its own symbols.
- Symbol value.** The integer or character string value of a symbol.
- Symbolic array.** An array whose data items are identified by a character string index (as opposed to a numeric index).
- System group.** A user group with system privilege. System groups are usually the groups with numbers 0–10 (octal, since UICs are specified in octal).
- Temporary command level.** The command level created by DCL's default interrupt handler when <CTRL/y> is pressed during execution of a command procedure.
- Temporary file.** A file created for the purpose of storing temporary information for a relatively short time. A program that creates a temporary file is responsible for deleting the file before it exits.
- Terminal characteristics.** A set of values that determines the appearance and behavior of a terminal. These values can be displayed with the SHOW TERMINAL command and altered with the SET TERMINAL command.
- Text file.** A sequential file whose records contain arbitrary text. Most of the time such files have variable-length records with the carriage return carriage-control attribute.

Time. A general VMS term used to refer to a date, a time, or some combination of the two. See also Absolute time, Combination time, Comparison time, and Delta time.

Translation. See Iterative translation, Logical name translation.

Unary operator. An operator that requires one operand (its arity is 1).

Update a record. To perform a file operation that modifies the contents of an existing record.

User address. A string that identifies a user on a VAX system. The user address consists of a node name and a user name.

User identification code (UIC). A number that identifies the owner of a VMS object. The UIC is divided into two fields, the group number and the member number.

User name. The symbolic name assigned to a VMS user's account. User names are typically last names, initials, or nicknames.

Variable-length record. A record in a sequential or indexed file whose design allows each record in the file to be of a different size. The text files on VMS contain variable-length records.

Variable with fixed control (VFC). A type of record format in which each record has a variable length and includes a fixed-size control area at the beginning. This control area might contain a line number or some other control information common to all records. The DCL OPEN command creates VFC records.

Verification. A method of debugging a command procedure by tracing its commands as they are executed by DCL. Verification is enabled with the SET VERIFY command.

Virtual device. A facility provided by VMS software, which acts like a hardware device and is therefore used by a program as if it were a hardware device. Examples are mailboxes, virtual terminals, and VAXstation windows.

Volume. The magnetic medium used on a disk or tape drive to store permanent information.

Volume initialization. The act of formatting a disk or magnetic tape volume so that it can accept and store data.

Volume label. The character string that names a particular disk or tape volume.

Volume loading. The action taken when a human being places a data volume on a disk or tape drive. Once the volume is loaded, it can be mounted to make it available to the system.

Volume mounting. The process by which VMS recognizes a data volume loaded on a disk or tape drive. Data transfers cannot be performed until the volume is mounted.

Whitespace. A sequence of spaces or tabs in a command or character string.

Wildcard spec. A file spec containing one or more of the wildcard matching characters (asterisk, percent sign, or ellipses).

Working directory. The directory in which a user is currently working, as established with the SET DEFAULT command. Files are located in the working directory if no other device and directory is specified.

Index

(special characters are indexed at the end)

abbreviations

- caution about using in command procedures, 20
- DCL verbs
 - allowed abbreviation of, 13
 - rules for, 11
- keywords, building so that they may be abbreviated, 201
- personal commands as, 71
- terminal input, using F\$LOCATE to allow, 130

absolute time

- See also* combination time; delta time; time.
- glossary entry, 352
- represents specific points in time, 203

access

- See also* captive accounts; protection; security.
- algorithm, used for UIC- and ACL-based protection checking, 239
- blocking access by route of entry, 232
- category
 - based on relation of owner to user, 226
 - glossary entry, 352
- control entry (ACE)
 - glossary entry, 352

- identifier, format and description of components, 232
- setting working directory protection using an ACE, 234
- use in file protection, 230
- control list (ACL)
 - ACE assigned creator when not owner of file, 238
 - compared with user identification codes (UIC), 225
 - default ACL for new file, steps determining, 238
 - default, for directories, 237
 - default, for files, 237
 - default identifier entry for directories, 233
 - file protection use of ACL, 225
 - format of identifier entries, description of components, 232
 - glossary entry, 352
 - kinds of rights identifiers, 231
 - matching process when a file has an ACL, 231
 - modifying ACLs with editor or SET, 234
 - NOPROPAGATE used to control default ACL, 238
 - potential outcomes of matching process, 231
 - SET ACL command qualifiers, 235

- access** (*continued*)
 - control list (ACL) (*continued*)
 - use in file protection, 230
 - warning, order of entries is critical, 233
 - DCL, controlling with captive accounts, 122
 - flags for each access category, 227
 - modes
 - defining logical names in different modes, 168
 - for VAX process, glossary entry, 352
 - logical names, VMS use of each mode, 168
 - privileges assigned to processes, 167
 - privilege scheme description, 17
 - /READ access when opening a file, 184
 - requirements for directory operations (table), 227
 - requirements for file operations (table), 228
 - rights identifier
 - glossary entry, 362
 - maintained with AUTHORIZE, 231
- accessing**
 - files in personal directories before those in public directories, using search lists, 173
- ACE**
 - See* access, control entry.
- ACL**
 - See* access, control list.
- addition (+)**
 - integer operator, arity, result type, and value (table), 32
- address, user**
 - glossary entry, 364
 - identifies location and name of user, 171
- ALLOCATE**
 - command to request exclusive use of device, 247
- allocating**
 - devices, 247
- altering**
 - See* modifying.
- alternate key**
 - in indexed file, 210
 - glossary entry, 352
- ampersand substitution**
 - See also* apostrophe substitution; substitution.
 - description, 73
 - explicit substitution of text in DCL
 - command, glossary entry, 352
 - used in concert with apostrophe substitution, 73
- ANALYZE/RMS_FILE**
 - file utility, one line description, 159
- and (.AND.)**
 - boolean operator, arity, result type, and value (table), 34
 - integer operator, arity, result type, and value (table), 32
- annotation**
 - See also* documentation.
 - separator, use of \$ alone on line in DCL
 - command procedures, 19
 - use of comments to clarify complex procedures, 19
- answer, default**
 - See* default answer.
- apostrophe substitution**
 - See also* ampersand substitution; substitution.
 - creating DCL arrays, techniques and problems, 285
 - explicit substitution of symbol or lexical function values, 66
 - explicit substitution of text in DCL
 - command, glossary entry, 352
 - interpretation when input to INQUIRE, 124
 - trailing, when required in apostrophe substitution, 67
- APPEND**
 - file utility, one line description, 159
 - protection mask setting, 229
- appending**
 - See also* creating; modifying.
 - files, OPEN/APPEND allows records to be added to file, 189
 - open for append, glossary entry, 359
- applications**
 - appropriate for DCL, 3
 - controlling access with captive accounts, 122
 - passing commands to applications from DCL, 134
 - sample, description, and listing, *See* MODL.
 - structure, overview of application design, 111

- archiving**
 - files, possible application for DCL, 5
- arguments**
 - See also* parameters.
 - allowing for prompts or batch, 131
 - input for an operation, glossary entry, 352
 - lexical function
 - characteristics, 37
 - in format of lexical functions, 36
 - optional, glossary entry, 359
- arithmetic**
 - techniques built up from rudimentary operations, 196
- arity**
 - defined as number of operands required by an operator, 31
 - glossary entry, 352
- arrays**
 - See also* lists.
 - glossary entry, 352
 - handling
 - procedure parameters as an eight element array, 287
 - substitution of array element names, 286
 - index, glossary entry, 353
 - integer, arrays with integer indexes, 284
 - simulation
 - in DCL, 284
 - double substitution used for simulating arrays, 73
 - symbolic
 - array, glossary entry, 363
 - creating, 288
 - problems connected with symbolic arrays, 289
 - warning, arrays may require symbol table expansion, 290
- ASCII (American Standard Code for Information Interchange)**
 - character set
 - appendix with table, 293
 - glossary entry, 353
 - used by VAX architecture, 24
 - output, formatting, 78
- ASK**
 - features of complex input routine, 132
 - listing in subroutine library, 298
 - personal command created using READ, 125
- ASSIGN**
 - logical name definition, parameters reversed from DEFINE, 165
- assignment**
 - See also* symbols; values.
 - bit-fields in character strings, 44
 - command
 - description, 25
 - expression use by, 26
 - glossary entry, 353
 - symbols are assigned a value with, 22
 - description of use, 25
 - substring, 42
 - summary of assignment commands (table), 47
- attributes**
 - complete list returned by DIRECTORY /FULL, 158
 - file
 - attributes, glossary entry, 356
 - descriptive information (table), 157
- AUTHORIZE**
 - See also* access; protection; security.
 - access control rights identifiers defined with AUTHORIZE, 231
 - command, accessing with DCL, 4
- authorized privileges**
 - description, 17
 - glossary entry, 353
- background execution**
 - SPAWN /NOWAIT allows simultaneous execution of processes, 263
- BACKUP**
 - file utility, one line description, 159
 - message file handling, 102
 - protection mask setting, 229
- /BACKUP**
 - file selection qualifier, one line description, 160
- backups**
 - example of use for DCL, 4
- base**
 - numeric, use of octal, decimal, and hexadecimal, 23
- batch**
 - See also* interactive.
 - breaking batch procedures into steps for possible restart, 278
 - controlling restart entry point, 276
 - environment described, 270
 - glossary entry, 361
 - jobs
 - application for DCL, 3
 - concepts and facilities, 270

batch (*continued*)jobs (*continued*)

- controller responsible for job queue maintenance, 270
- duplicate jobs, preventing occurrence of duplicate jobs, 282
- glossary entry, 353
- invoking with SUBMIT command, 271
- periodic batch jobs resubmit themselves on schedule, 279
- queues for running batch jobs, 270
- restartable batch jobs, restarting after STOP /REQUEUE, 276
- status report can be tailored and mailed to user, 274

LOGIN.COM procedure invoked, 271

process

- description, 254
- glossary entry, 353
- versus interactive, 271

BATCH system identifier

description, 231

BATCH\$RESTART

controlling restart entry point, 277

/BEFORE

- example of use of DCL command qualifiers, 13
- file selection qualifier, one line description, 160

beginning DCL command procedures

- using @ command, 18
- detailed description, 58

binary operators

- See also* unary operators.
- notation, description in relation to hexadecimal, 291
- operators
 - description and examples, 32
 - glossary entry, 353

bit

- glossary entry, 353
- numbering of bits in character string, 44

bit bucket

See null device.

bit-field assignment

- altering sequence of bits in value of symbol, 44
- control characters can be created in DCL command, 46

bitwise

- and (.AND.), arity, result type, and value (table), 34
- inclusive or (.OR.), arity, result type, and value (table), 34
- not (.NOT.), arity, result type, and value (table), 34

boolean

- description and use, 25
- FALSE, even integers and certain character strings are used to represent FALSE, 25
- field type, definition symbol and characteristics, 207
- fields in record structures, created with bit-field assignment, 208
- glossary entry, 353
- operators, table of operator arity, result type, and value, 34
- responses, limiting to yes or no, 130
- TRUE, odd integers and certain character strings are used to represent TRUE, 25

Buffett, Jimmy

quotation, xvi, 411

building

- software systems from components, use for DCL, 6

/BY_OWNER

file selection qualifier, one line description, 160

BYPASS

privilege that affects protection-checking scheme, 228

calendar

possible use for DCL, 4

CALL

- compared with
 - @ command, 62
 - GOSUB command, 62
- description and use, 62
- new to VMS Version 5, 62
- summarized in table of subroutine facilities, 64

calling

- See also* subroutines.
- DCL applications, methodology for, 112
- command procedures, using @ command, 18
- command procedures, using @ command, detailed description, 58
- subroutines, 119

- canceling**
 - a procedure from temporary command level, 105
- captive account**
 - See also* access; protection; security.
 - controlling access to applications with captive accounts, 122
 - glossary entry, 353
- card reader**
 - specified as device name, 144
- carriage return carriage-control**
 - glossary entry, 353
 - standard, but not guaranteed, record format, 188
- case**
 - characters must be same case before comparison, 127
 - DCL converts all symbols to uppercase, 22
 - statement
 - glossary entry, 353
 - how to emulate case statement in DCL, 202
- catalog**
 - directory as a catalog of files, 145
- cautions**
 - See* warnings.
- “changes” attribute**
 - See also* files, indexed.
 - required if alternate key fields are to be updated, 220
- changing**
 - See* modifying.
- characteristics**
 - terminal, description, 246
 - terminal, glossary entry, 363
- characters**
 - ASCII character set, appendix, 294
 - control
 - character, glossary entry, 354
 - component of DCL character set, 24
 - creating control characters in DCL command, 46
 - not representable in string literals, 24
 - glossary entry, 353
 - multinational character set, appendix, 293
 - printable
 - component of DCL character set, 24
 - glossary entry, 360
 - set of
 - ASCII character set used by VAX architecture, 24
 - defined as complete collection of characters and code numbers, 24
 - special, sending special characters to terminal, 76
 - strings
 - See* strings, character.
- checking**
 - See also* cleanup; debugging; errors; status, code.
 - existence of file with F\$SEARCH, 152
 - file specification validity with F\$PARSE, 153
 - status code
 - methods for, 92
 - with ON command, 95
 - terminal input, character strings, 129
 - type of symbol values with F\$TYPE, 39
 - validity of integer in input string, 128
- child**
 - See also* parent; process.
 - directory, parsing requires special treatment, 150
 - process, glossary entry, 353
- choosing**
 - DCL, reasons to choose DCL as programming language, 2
- cleanup**
 - See also* checking; debugging; errors; status.
 - closing open files as part of cleanup code, 186
 - code
 - example of use in main procedure, 115
 - use in subprocedures, example structure, 117
 - keyword checking in cleanup code, 201
 - procedure
 - example of use of EXIT, 62
 - glossary entry, 360
 - terminating procedures after errors, techniques for, 97
- CLOSE**
 - necessary for closing open files, 184
 - open files must be explicitly closed, 186
- closing**
 - See also* opening.
 - files, as part of cleanup code, 186
- code**
 - See* DCL, code; status, code.
- .COM**
 - type given file containing DCL command procedure, 18

combination time

See also absolute time; delta time; time.
format using absolute and delta times,
204

VMS format, glossary entry, 353

command

See also procedures, command.

DCL command, glossary entry, 355

description of format and use for DCL
commands, 11

execution, controlling in subprocesses,
262

file location of error messages for status
code interpretation, 103

flow-of-control, glossary entry, 356

interpreter, guided by information
concerning DCL environment,
82

language interpreter, glossary entry, 354
level

temporary, canceling a procedure
from, 105

temporary following interrupt, 105

modifying, using the substitution facili-
ties for, 66

parameter, glossary entry, 354

personal

application invocation use of, 112

command substitution, glossary
entry, 360

glossary entry, 360

procedures

See procedures, command.

protection mask, commands that can set
the protection mask, 229

qualifier, glossary entry, 354

sequence, stored in text file as command
procedure, 18

syntax, glossary entry, 354

temporary command level, glossary entry,
363

verbs

description and use in DCL, 11

glossary entry, 354

comments

See also documentation.

description and use in DCL command
procedure, 19

glossary entry, 354

move to end of procedure to avoid slow
down, 296

comparison

See also boolean.

time

format to represent times to be
compared, 203

VMS format, glossary entry, 354

times using DCL commands, 203

computer-supported cooperative work

DCL as a tool for, 5

concatenation (+)

string operator, arity, result type, and
value (table), 33

concealing

root directory logical name values, 170

condition

glossary entry, 354

IF command test, 52

conditional execution

IF command as control structure for, 52

/CONFIRM

example of use of DCL command
qualifiers, 12

constant

See also symbols.

literal, glossary entry, 358

context symbol

glossary entry, 354

use with F\$PID lexical function, 257

continuation lines

See also documentation.

use of hyphen to allow multiple lines in
DCL command, 13

CONTINUE

returning to program after interrupt, 105

control

characters

creating in DCL command with
bit-field assignment, 46

glossary entry, 354

not representable in string literals, 24
flow, *See* flow-of-control; control,
structures.

keys

<CTRL/y>, program interrupt
handling, 104

sequences

glossary entry, 354

terminals, sending special characters
to terminal, 76

uses for, 77

string, to specify format for F\$FA0,
glossary entry, 354

structures

See also flow-of-control.

control (*continued*)

- structures (*continued*)
 - concepts and commands, 49
 - conditional execution, IF, 52
 - iteration, 56
 - sequential execution, 49
 - unconditional goto, GOTO, 50

CONTROL access

- description, 227

controller

- job, supports batch processing facility, 270
- job, glossary entry, 357

conventions

- typographic, 7

conversion

- data, using F\$STRING and F\$INTEGER
 - lexical functions, 39
- noninteger strings converted to integers
 - become zero, 36
- rules for matching operators and operand types, 35
- time formats converted using F\$CETIME, 205

CONVERT

- file utility, one line description, 159

COPY

- file utility, one line description, 159
- protection mask setting, 229

CREATE

- file utility, one line description, 159
- protection mask setting, 229
- use in creating indexed file using FDL file, 216
- use to build normal text file, 189

CREATE/DIRECTORY

- file utility, one line description, 159

/CREATED

- file selection qualifier, one line description, 160

creating

- See also* appending; modifying.
- arrays
 - with integer indexes, 284
 - with symbolic indexes, 288
- boolean fields in record structures, created
 - with bit-field assignment, 208
- control characters in DCL command with bit-field assignment, 46
- date fields in record structures, created
 - with substring assignment, 208
- files
 - impact of search lists on, 173
 - using CREATE, 189
 - using OPEN, 187
 - using WRITE, 188
- global symbols, prevented by NOGLOBAL keyword, 84
- indexed files, 216
- integer fields in record structures, created
 - with bit-field assignment, 208
- lists, 198
- procedure status codes, 93
- record structures by storing values in character string, 208
- strings
 - with multiple characters using bit-field assignment, 45
 - with one character using bit-field assignment, 45
- subprocesses, 260
- symbols
 - global, 29
 - procedure level, 27
 - prompt level, 27
 - using assignment command for, 22, 26
- text fields in record structures, created
 - with substring assignment, 208

CSI (control sequence initiator)

- eight-bit terminal use of hexadecimal 9B as, 76

<CTRL/c>

- procedure interrupt handling, 104

<CTRL/y>

- program interrupt handling, 104
- use SPAWN to execute commands without terminating interrupted process, 262

<CTRL/z>

- end-of-file when read from terminal, 126

current directory

- See also* directories.
- defined as default file directory, 88

customer

- See also* status, code.
- facility number, required by procedure status codes, 93

data

See also symbols.

entry, interactive, obtaining, 123

glossary entry, 354

image

data, input to applications called
from DCL, 134

input data for a program, glossary
entry, 357

internal and external representation of
data, 23

item, glossary entry, 354

manipulation techniques in DCL, 196

structuring, nonexistent in DCL, 2

types

associated with value not symbol, 26

boolean, representation and use, 25

character, representation and use, 23

description, 22

glossary entry, 355

integer, representation and use, 23

volumes, UIC protection available, 225

[.DATA]

example of subdirectory under working
directory, 146

date

See also time.

field type, definition symbol and charac-
teristics, 207

fields in record structures, created with
substring assignment, 208

manipulating using DCL commands, 202

DCL (Digital Command Language)

See also command.

application domain, 3

caution about viruses, 6

code

glossary entry, 355

name for commands in DCL com-
mand procedure, 18

commands

format description, 11

glossary entry, 355

interpreter, description of program
that carries out instructions,
11

parameters, description and use, 12

procedure, glossary entry, 355

qualifiers, description and use, 12

syntax, description and use, 13

debugging

See debugging.

defined, 1

environment, glossary entry, 355

preventing escape to, 122

as a programming language, 2

prompt

character, dollar sign (\$) as prompt,
11

glossary entry, 355

strong points, 2

uses in

complex file manipulation, 5

environmental extension, 3

software development, 5

system management, 4

weak points, 2

DEALLOCATE

command to release an allocated device,
249

deallocating

devices, 249

DEASSIGN

logical name deletion command, 176

debugging

See also checking; cleanup; errors; status;
code.

capturing

output with /LOG, 142

procedure output with /OUTPUT, 141

concepts and techniques, 136

DCL advantages and disadvantages

compared with conventional
languages, 136

enabling verification

from inside procedure, 137

from outside procedure, 137

!!! flag to help locate temporary
debugging commands, 138

permanent debug lines

activated with flag, 140

cost execution time, 141

SHOW, displaying values of symbols, 140

temporary debug lines marked with !!!,
140

tracing procedures using SET VERIFY, 83

verification, glossary entry, 364

decimal base (base 10)

representation and use, 23

decisions

See also control, structures.

flow-of-control commands discussed, 49

IF command discussed, 52

DECnet

- node, file specification component, full description, 143
- path specification for reaching nodes, 144
- second log session created to record debug output, 142

default

- answer
 - for READ command, 126
 - glossary entry, 355
- directory
 - component of DCL environment, characteristics, 88
 - do not change in a procedure, 88
 - glossary entry, 355
 - importance to VMS user, 15
- error handler, *See* errors.
- interrupt handler described, 105
- protection ACE, use in setting protection of new files in a directory, 234
- question responses should not contain dangerous material, 126
- value
 - glossary entry, 355
 - lexical functions with optional arguments provide a default value, 37
- working directory, glossary entry, 365

DEFINE

- defining logical names with, 165
- logical name
 - creation and value assignment, 15
 - definition with, 162
- redirecting
 - program input using, 134
 - screen output to a file, 80
- temporary definition of a symbol using /USER_MODE, 80

defining

- See also* undefining.
- logical names
 - in different access modes (table), 168
 - with DEFINE command, 162
- symbols, with assignment command, 26

DELETE

- example of DCL command parameters and qualifiers, 12
- file utility, one line description, 159

DELETE access

- description, 227

deleting

- duplicates from lists with DCL commands, 198
- files
 - temporary, 191
- indexed file records, avoid use of generic matching, 221
- logical names using DEASSIGN command, 176
- records in an indexed file, 220
- symbols using DELETE/SYMBOL command, 47
- temporary files, procedure for, 60

delta time

- See also* absolute time; combination time; time.
- format for showing difference between two times, 204
- glossary entry, 355

detached process

- See also* process.
- description, 254
- glossary entry, 355

devices

- ALLOCATE command to request exclusive use, 247
- allocating specific devices with /GENERIC qualifier, 248
- characteristics and operations, 241
- DEALLOCATE, command to release an allocated device, 249
- device class, integer useful for verification, 243
- existence checking with EXISTS keyword to F\$GETDVI, 242
- file specification component
 - full description, 144
 - overview, 14
- glossary entry, 355
- information available with F\$GETDVI, 242
- list of potential devices, 241
 - logical name use with, 162
- messages about allocation using /LOG, 248
- MOUNT qualifiers (table), 251
- mount, steps performed in mounting a volume, 251
- names
 - characteristics, 241
 - format has changed with advent of new technology, 242
- null device, glossary entry, 359

- devices** (*continued*)
 - specifying with logical names, advantages, 144
 - system, listing with SHOW DEVICE, 245
 - terminal server is single device that appears as many, 242
 - UIC protection available, 225
 - virtual
 - description and examples, 241
 - glossary entry, 364
 - volume initialization, description of parameters and use, 249
- DIALUP system identifier**
 - description, 231
- difference**
 - integer subtraction operator, arity, result type, and value (table), 32
 - time, how to calculate and represent differences between times, 204
- DIFFERENCE**
 - file utility, one line description, 159
 - using \$SEVERITY to analyze its status code, 96
- Digital Command Language (DCL)**
 - See DCL.
- directories**
 - See also files.
 - access protection categories, 227
 - accessing public and private directories with search lists, 173
 - as catalog of files, 145
 - child, parsing requires special treatment, 150
 - concepts
 - introduced, 14
 - and facilities, 143
 - default
 - do not change in a procedure, 88
 - glossary entry, 355
 - importance to VMS user, 15
 - protection, how to determine default protection, 237
 - determining parent of arbitrary directory difficult, 150
 - facility code use for naming compatibility, 112
 - file specification
 - component, full description, 145
 - component, overview, 14
 - manipulation involving rooted directory difficult, 150
 - glossary entry, 355
 - home
 - glossary entry, 357
 - importance to VMS user, 15
 - <> is alternate symbol for demarcating directory name, 146
 - logical name
 - descriptions of, 180
 - table directory, glossary entry, 358
 - use with, 162
 - login
 - glossary entry, 358
 - importance to VMS user, 15
 - master file directory, glossary entry, 359
 - operations, table of required access categories, 227
 - parent, parsing requires special treatment, 150
 - protection, setting default protection using an ACE, 234
 - root
 - defining with logical names, 170
 - glossary entry, 362
 - searching for files, 151
 - subdirectory, glossary entry, 363
 - system directory organization, overview, 177
 - UIC protection available, 225
 - wildcards for parts of directory name, 151
 - working directory, glossary entry, 365
- DIRECTORY**
 - command useful in search, 151
 - file utility, one line description, 159
- DIRECTORY/FULL**
 - file attributes display by, 158
- DIRECTORY/PROTECTION**
 - displaying values in protection mask, 229
- disabling**
 - automatic checking of status codes, 96
 - predefined personal commands inside of procedures, 72
- discarding output**
 - device NL: used for, 145
 - from a SEARCH command, 195
- disk**
 - specified as device name, 144
- DISPLAY**
 - suggested personal command for writing to screen, 76

- displaying**
 - See also* output; terminal.
 - commands and data during program execution, 137
 - default process protection mask, 238
 - logical name values, 166
 - output during procedure execution, 75
 - protection mask, 229
 - sequential files
 - record at a time is slow but allows more control, 193
 - TYPE/PAGE displays text a screenful at a time, 192
 - suppressing error message display on terminal, 80
 - text, using TYPE command, 134
 - values of symbols for debugging purposes, 140
- distributed applications**
 - using DCL with distributed applications, 5
- distribution list**
 - managed by sample application MODL, 304
 - glossary entry, 355
- division**
 - integer operator, arity, result type, and value (table), 32
- documentation**
 - See also* comments.
 - separator, use of \$ alone on line in DCL command procedures, 19
 - use of comments to clarify complex procedures, 19
 - VMS documentation important for DCL programmer, list of, 8
- duplicate jobs**
 - See also* jobs.
 - preventing occurrence of duplicate jobs, 282
- editing**
 - terminal input, READ input string editing with F\$EDIT, 127
- ELSE**
 - use with IF for listing alternative commands, 54
- end-of-file**
 - <CTRL/z> signal from terminal, 126
 - glossary entry, 356
 - handler can be provided for READ, 185
 - indexed files, signaled by key match failure, 215
- /END_OF_FILE**
 - qualifier specifies end-of-file-handler for READ, 185
- ENDSUBROUTINE**
 - use with CALL command, 63
- environment**
 - batch jobs, 270
 - DCL
 - concepts and facilities, 82
 - environment, glossary entry, 355
 - main procedure use in establishing DCL environment, 113
 - process
 - commands to establish (table), 259
 - some information available only for current process, 259
 - steps in establishing DCL environment for a procedure, 115
 - subprocesses, list of inherited and not-inherited environment properties, 260
- environmental extension**
 - application for DCL, 3
- .EQ.**
 - integer operator, arity, result type, and value (table), 32
- .EQS.**
 - example of use with .OR., description of precedence, 58
 - string operator, arity, result type, and value (table), 33
- equals (.EQ., .EQS.)**
 - integer operator, arity, result type, and value (table), 32
 - string operator, arity, result type, and value (table), 33
- equivalence string**
 - glossary entry, 356
 - value of logical name, 162
- errors**
 - See also* checking; cleanup; debugging; interrupts.
 - captive account handling of, 122
 - cleanup before termination of procedure, 97
 - creating procedure status codes, 93
 - default error handler
 - characteristics, 94
 - modifying, 95
 - problems, 94
 - disabling automatic checking with SET NOON, 96
 - handler, glossary entry, 356

- errors** (*continued*)
- handling
 - concepts and facilities, 90
 - custom, example of use in main procedure, 114
 - integrating with interrupt handling, 106
 - with cleanup code, example of use in subprocedures, 117
 - without cleanup code, example of use in subprocedures, 116
 - ignoring error codes using SET NOON, 100
 - nested procedures require unwinding to handle errors, 98
 - SIGNAL**
 - listing in subroutine library, 301
 - message and error handling subroutine, characteristics and use, 120
 - signal on error, glossary entry, 362
 - suppression of error message display on terminal, 80
 - /ERROR**
 - OPEN qualifier specifies handler for open errors, 184
 - READ qualifier specifies handler for read errors, 185, 214
 - ESC (escape character)**
 - seven-bit terminal use with [, 76
 - escape sequence**
 - glossary entry, 356
 - terminals, sending characters to older terminals, 76
 - evaluate**
 - See also* evaluation.
 - glossary entry, 356
 - evaluation**
 - See also* substitution.
 - environments for automatic substitution of expressions, 70
 - expressions, how processed by DCL, 31
 - precedence rules for expression evaluation, 33
 - modifying with parentheses, 34
 - table, 35
 - symbols, using apostrophe substitution in, 66
 - /EXCLUDE**
 - file selection qualifier, one line description, 160
 - .EXE**
 - file type for executable file produced by linker, 146
 - execute, execution**
 - See also* control, structure; flow-of-control; invoke; run.
 - glossary entry, 356
 - invoking procedures, 58
 - overview, 18
 - parallel processes, using SPAWN, 263
 - sequential, description, 49
 - sequential, glossary entry, 362
 - EXECUTE access**
 - description, 227
 - executive**
 - access mode in which a process may run, 167
 - existence**
 - symbols, checking using F\$TYPE, 39
 - exit**
 - See also* EXIT.
 - procedure, description, 61
 - procedure, glossary entry, 360
 - subroutine invoked with CALL, 62
 - EXIT**
 - command to force termination of procedure, 61
 - convention to use explicitly for ease of understanding, 63
 - GOSUB requires use of EXIT to avoid dropping into subroutine, 65
 - status code parameter for reporting procedure outcome, 93
 - expiration date**
 - setting using combination time, 204
 - /EXPIRED**
 - file selection qualifier, one line description, 160
 - expressions**
 - arbitrary, substituting the value of, 69
 - assignment command use of, 26
 - evaluation precedence rules, 33
 - modifying with parentheses, 34
 - table, 35
 - explicit substitution normally unnecessary with, 69
 - glossary entry, 356
 - using, 31
 - where legal, 69
 - WRITE command use of expressions, 76
 - extracting**
 - fields from a structure, 209
 - list items, 199
 - substrings, using F\$EXTRACT, 41

- F\$**
used to begin name of lexical function, 36
- F\$CVSI**
extracts fields from a structure, 209
- F\$CVTIME**
converts between time formats, 203
- F\$CVUI**
reason not used in this book, 209
- F\$EDIT**
edits character strings, 127
trims trailing spaces from a line, 193
- F\$ELEMENT**
extracts items from a list, 199
file specs, do not use to extract components, 200
- F\$ENVIRONMENT**
item description table, 256
obtains process environment information, 256
process message settings, 85
process verification settings, 139
- F\$EXTRACT**
extracts fields from a structure, 209
extracts substring from a string, 41
- F\$FAO**
directive description table, 78
formats strings with text and data, 78
- F\$FILE_ATTRIBUTES**
item description table, 157
obtains file information, 157
obtains protection mask values, 230
- F\$GETDVI**
item description table, 243
obtains device information, 242
- F\$GETJPI**
item description table, 255
obtains process information, 254
- F\$GETSYI**
obtains system information, 37
- F\$INTEGER**
converts string to integer, 39
avoids operators with mixed operand types, 36
substitutes value of arbitrary expression in command, 69
- F\$LENGTH**
returns length of character string, 40
validates length of input string, 129
- F\$LOCATE**
searches string for pattern, 40
validates boolean input, 130
- F\$LOGICAL**
replaced by F\$TRNLNM in VMS Version 5, 175
- F\$MESSAGE**
looks up message for status code, 101
- F\$MODE**
use F\$GETJPI instead to obtain process mode, 256
- F\$PARSE**
checks validity of file specification, 153
parses and expands file specs, 148
preferred function to extract file spec component, 200
- F\$PID**
obtains process IDs of system processes, 257
privileges determine which other processes will be located, 258
used to display CPU time of all users, 288
- F\$PROCESS**
use F\$GETJPI instead to obtain process name, 256
- F\$SEARCH**
searches for files that match a given spec, 152
- F\$SETPRV**
enables or disables privileges, 86
- F\$STRING**
converts integer to string, 39
avoids operators with mixed operand types, 36
substitutes value of arbitrary expression in command, 69
- F\$TIME**
returns current absolute time, 203
- F\$TRNLNM**
item description table, 176
replaces F\$LOGICAL in VMS Version 5, 175
translates logical name, 174
- F\$TYPE**
determines whether symbol exists, 39
returns type of symbol value, 39
validates integer in input string, 128
- F\$USER**
use F\$GETJPI instead to obtain UIC, 256
- F\$VERIFY**
enables and disables procedure verification, 137
work-arounds for inconsistent behavior, 139

- facility**
 - code, naming compatibility use of, 112
 - number, required by procedure status codes, 93
- FALSE boolean value**
 - even integers and certain strings used to represent, 25
 - literal must be established with assignment command, 25
- FDL (File Definition Language)**
 - editor, creating indexed files using FDL editor, 216
 - use in creating indexed files, 216
- field**
 - See also* records.
 - extracting fields from a structure, 209
 - glossary entry, 356
 - key, glossary entry, 358
 - structured records are formatted into fields, 205
 - types, records, definition symbols, and characteristics, 207
- file**
 - See also* directories.
 - access
 - categories, based on relation of owner to user, 226
 - protection categories, 227
 - /READ access requested when opening a file, 184
 - appending, OPEN/APPEND allows WRITE to add records to text file, 189
 - attributes
 - complete list displayed with DIRECTORY /FULL, 158
 - file descriptive information (table), 157
 - glossary entry, 356
 - blocks, free paging, testing with IF command, 53
 - carriage return carriage-control attribute set in normal text file, 188
 - changing file type with F\$PARSE, 151
 - checking
 - existence of file with F\$SEARCH, 152
 - validity of file spec with F\$PARSE, 153
 - CLOSE necessary to close open files, 184
 - close open files in two places in a procedure, 186
 - command
 - procedure, defined, 1
 - procedure given type COM, 18
 - concepts and facilities, 143
 - counting files of a given type using F\$SEARCH, 154
 - CREATE not WRITE builds normal text file, 189
 - creating
 - impact of search lists on, 173
 - using CREATE, 189
 - using OPEN, 187
 - using WRITE, 188
 - default protection, steps to determine default protection, 237
 - deleting temporary files, 191
 - directory, file specification component, full description, 145
 - displaying sequential files
 - record at a time is slow but allows more control, 193
 - TYPE/PAGE displays text a screenful at a time, 192
 - expiration date, setting using combination time, 204
 - F\$SEARCH, existence checking with, 152
 - facility code use for naming compatibility, 112
 - glossary entry, 356
 - indexed
 - alternate key, glossary entry, 352
 - attributes of primary and alternate keys, 211
 - creating alternate key fields using "changes" attribute for later updating, 220
 - creating indexed files using FDL, 216
 - file operations, 210
 - file prologue contains information describing keys, 211
 - file prologue, glossary entry, 356
 - generic match using /MATCH qualifier, 215
 - key fields indexed for fast lookup, 210
 - key lookup by exact match, 214
 - key lookup, /MATCH qualifier, 215
 - limitation on number of alternate keys, 210
 - opening an existing indexed file for reading, 213
 - opening for writing with /READ /WRITE qualifier, 219
 - operations, concepts and operations, 210
 - reading duplicate records from a file, 214

- file** (*continued*)
 - indexed (*continued*)
 - reading randomly by key, 213
 - reading sequentially, 213
 - updating, 220
 - updating a file using WRITE command, 220
 - warning to avoid generic matching when deleting records, 221
 - writing, 219
 - information, file attributes (table), 157
 - locked records, ways to unlock records, 223
 - log, glossary entry, 358
 - manipulation
 - complex, using DCL for complex file manipulation, 5
 - DCL usefulness for, xiii
 - master file directory
 - glossary entry, 359
 - root of entire directory tree, 145
 - message, glossary entry, 359
 - multiple
 - copies of same file distinguished by version number, 147
 - searches requires multiple contexts, 155
 - opening
 - a file, glossary entry, 359
 - existing files, 184
 - for append, glossary entry, 359
 - operations
 - for manipulating files, summary, 158
 - table of required access categories, 228
 - output of WRITE may not be normal text file, 188
 - overview, 14
 - owner identification UIC described, 16
 - parsing file spec
 - glossary entry, 360
 - without existence check, 149
 - permanent files, differences from temporary files, 190
 - process-permanent
 - file for video output, 75
 - file, glossary entry, 361
 - standard files and uses, 265
 - prologue, glossary entry, 356
 - protection
 - concepts and facilities, 225
 - displaying values in protection mask, 229
 - glossary entry, 356
 - mask, access flags for each access category, 227
 - obtaining values of mask in a procedure, 230
 - setting flags in protection mask, 229
 - reading
 - existing file, 184
 - sequential, 183
 - record format
 - free-format text versus record structures, 183
 - redirecting output from screen to a file, 79
 - resultant file spec, glossary entry, 361
 - searching file contents, 194
 - sequential
 - concepts and operations, 183
 - file, glossary entry, 362
 - file operations, 183
 - files with structured records, 206
 - sharing
 - files over multiple processes, 221
 - glossary entry, 356
 - specifications
 - annotated list of components of DCL file specification, 14
 - complete parsing algorithm, 165
 - full description of each component, 143
 - glossary entry, 356
 - logical names in, 163
 - parsing and completing missing components, 148
 - search list for parsing, 180
 - warning, do not use F\$ELEMENT to extract components, 200
 - stream ID useful in identifying context of each search, 155
 - structured records, introduction and description, 205
 - superceding in personal over public directories using search lists, 173
 - SYS\$COMMAND, process-permanent file
 - for input from procedure or keyboard, 123
 - SYS\$INPUT, process-permanent file
 - for input from procedure or keyboard, 123
 - temporary files
 - deleting temporary files, 191
 - description and differences from permanent files, 190

file (*continued*)
 temporary files (*continued*)
 glossary entry, 363
 text, glossary entry, 363
 transfer over network at night is good use
 for DCL, 5
 type, changing, 150
 UIC protection available, 225
 utilities (table), 158
 variable-length versus fixed-length
 records, 184
 VMS utilities, table of names and
 purpose, 159
 warning, protection mask does not
 include all information about
 file protection, 230
 writing
 sequential, 183
 to, 187

File Definition Language (FDL)
See FDL.

fixed-length records
See also records.
 glossary entry, 356
 uncharacteristic of text files, 184

flag
See also checking; cleanup; debugging;
 errors.
 global debug flag, allows permanent
 debug lines, 140

floating-point numbers
 nonexistent in DCL, 2, 197

flow-of-control
See also control, structures.
 commands, glossary entry, 356
 concepts and commands, 49

FOR loops
See also GOTO.
 DCL is missing FOR loops, 3

.FOR
 file type for FORTRAN source file, 146

formats
 DCL command format described, 11
 disks, standard is Files-11 Structure Level
 2, 249
 tapes, standard is Level 3 ANSI, 249

formatting
 input prompt lines, using READ and
 F\$FAO, 125
 output to terminal using F\$FAO directives
 (table), 78

formfeed
 bit-field assignment to create, 45

French

used in quotations, xvi, 411

functions, lexical

See lexical functions.

.GE.

integer operator, arity, result type, and
 value (table), 32

generic match

glossary entry, 356

reading records in an indexed file, 215

.GES.

string operator, arity, result type, and
 value (table), 33

global

level, symbol creation and properties, 29

sections, UIC protection available, 225

symbol

 assigning the value, 29

 creation prevented by NOGLOBAL
 keyword, 84

 deleting, 47

 distinguishing from prompt-level
 symbols, 29

 examples of use, 30

 existence limited to login period, 31

 facility code use for naming compat-
 ibility, 112

 flag use to activate permanent debug
 lines, 140

 glossary entry, 357

 level, creation and properties, 29

 name for symbols created at global
 level, 29

 obtaining the value, 29

 substring assignment on, 44

glossary

glossary entry, 357

glyphs

printable characters in DCL character
 strings, 24

goals

writing book, xiv

GOSUB

See also calling.

command, compared with CALL com-
 mand, 62

description of DCL command, 64

EXIT, importance of EXIT to avoid
 dropping into subroutine, 65

limited to 16 GOSUB levels, 65

summarized in table of subroutine
 facilities, 64

gotchas

See warnings. GOTO

See also control, structures.

avoid using, reasons discussed, 53

flow-of-control command discussed, 50

how to emulate a case statement with
GOTO, 202

subroutine library use of, 119

use in conjunction with IF command, 52

graphics

DCL has no graphics capabilities, 3

greater than (.GT., .GTS.)

integer operator, arity, result type, and
value (table), 32

string operator, arity, result type, and
value (table), 33

greater than or equal (.GE., .GES.)

integer operator, arity, result type, and
value (table), 32

string operator, arity, result type, and
value (table), 33

group

access category, who can access, 226

logical name

table described, 164

table, detailed description, 179

number

component of UIC, 16

glossary entry, 357

system group, glossary entry, 363

GRPPRV

privilege that affects protection-checking
scheme, 228

.GT.

integer operator, arity, result type, and
value (table), 32

.GTS.

string operator, arity, result type, and
value (table), 33

handler

See also error handling; interrupt
handling.

error handler, glossary entry, 356

hexadecimal

base 16, used in DCL, 23

notation, appendix, 291

hiding, symbols, 28

appendix describing, 291

hints

See also warnings.

allowed abbreviation of DCL verb, 13

.COM file type may be omitted after @, 18

components of DCL file specification

may be omitted, 14

directory specifications, 146

how to use this book, 6

logical name can include nonadjacent file
spec components, 163

master file directory name may be

omitted, 145

rules of abbreviation in DCL command

verb, 11

time functions allow omission of certain

fields, 205

times can be specified with fields omitted,
205

use for DCL to extend user environment,
33

using implicit substitution to write

personal commands, 71

versions, access new and old files, 147

wildcards for parts of file spec compo-
nents, 151

home directory

See also directories.

glossary entry, 357

importance to VMS user, 15

identifiers

ACE, format and description of compo-
nents, 232

process identifier, glossary entry, 361

rights identifier, glossary entry, 362

stream ID, glossary entry, 362

types of identifiers assigned when user
logs in, 231

UIC, description of components, 16

IF

See also control, structures.

flow-of-control command discussed, 52

lacking compound form prior to VMS

5.0, work-around discussed, 54

simple versus compound forms, 52

IF_DEBUG

conditional debugging with, 141

image data

See also data.

input

data for a program, glossary entry,
357

to applications called from DCL, 134

index

See also arrays.
arrays can be simulated in DCL, 284
book, 367
identifier for selecting item from string or
array, glossary entry, 357
string index, glossary entry, 362
substring assignment format, 42

/INDEX

qualifier for READ access to indexed file,
213

indexed files

See files, indexed.

infinite loops

See also control, structures.
example using GOTO command, 51
glossary entry, 357
preventing in file searching, 154

inheritance

See also process.
symbols and logical names by subprocesses, 264

initialization

code for setting DCL environment in a
procedure, 88
how to block interrupts during initializa-
tion, 108
volumes
initialization, glossary entry, 364
parameters and use, 249

input

ASK, listing in subroutine library, 298
checking validity, full description, 128
program, redirecting using DEFINE, 134
spawned processes accept command
string or procedure file, 261
terminal, features of complex ASK routine
discussed, 132
user
converted to uppercase by INQUIRE
unless quoted, 124
obtaining for an interactive program,
123
obtaining with INQUIRE, 124
READ, preferred input command, 125

INQUIRE

description and warnings about using,
124

integer

checking for valid integer in input string,
128
DECnet node name specified by integer,
144
even integers used as boolean FALSE, 25
field type, definition symbol and charac-
teristics, 207
fields in record structures, created with
bit-field assignment, 208
glossary entry, 357
indexes, arrays in DCL, 284
noninteger strings converted to integers
become zero, 36
operators, table of arity, result type, and
value, 32
representation and use, 23
restricted in size to 32 bits in DCL, 23

interactive

See also batch.
processes
description, 254
description, 10
glossary entry, 357
versus batch, 271
programs, obtaining user input, 123
session, description, 10

INTERACTIVE system identifier

description, 231

interpreter

command language interpreter, glossary
entry, 354
DCL command interpreter, description
of program that carries out
instructions, 11

interrupting

processes, SPAWN used to execute
commands without terminating
interrupted process, 262

interrupts

See also errors.
captive account ignoring of <CTRL/y>,
122
custom interrupt handler, how to estab-
lish, 106
default
interrupt handler described, 105
interrupt handler, problems dis-
cussed, 106
error handling should also handle
interrupts, 106
glossary entry, 357

interrupts *(continued)*

- handling
 - <CTRL/γ>, concepts and facilities, 104
 - custom, example of use in main procedure, 114
 - integrating with error handling, 106
 - with cleanup code, example of use in subprocedures, 117
 - without cleanup code, example of use in subprocedures, 116
- how to block during initialization, 108
- other uses not necessarily terminating program, 109
- subroutine interrupt handler, potential problems, 105
- temporary command level, characteristics, 105
- warning about using SET NOCONTROL, 109

invoke, invoking

- See also* control, structures; execute; flow-of-control; run.
- batch jobs, 271
- DCL applications, directory and personal command conventions for, 112
- command procedures, 18, 58
- glossary entry, 357
- subroutines with
 - @, 59
 - CALL, 62
 - GOSUB, 62
- use of system logical name and personal command to invoke an application, 112

ISAM

- See* files, indexed.

item

- data item, glossary entry, 354

iteration

- See also* control, structures.
- execution of commands in loop, 56
- glossary entry, 357

iterative translation

- blocked by * wildcard in logical names, 167
- glossary entry, 357
- logical names look up described, 163

Jennings, Will

- quotation, xvi

jobs

- batch
 - concepts and facilities, 270
 - glossary entry, 353
 - restartable, glossary entry, 361
 - restarting, 276
 - sending mail about status, 274
- controller
 - glossary entry, 357
 - responsible for job queue maintenance, 270
- description of uses, 266
- detached process or batch, glossary entry, 357
- duplicate, preventing occurrence of duplicate jobs, 282
- invoking batch job using SUBMIT command, 271
- logical name table
 - described, 164
 - description of uses, 266
 - detailed description, 179
 - glossary entry, 357
- queues
 - for running batch jobs, 270
 - glossary entry, 358
- status, sending mail about batch job status, 274

kernel

- See also* access; protection; security.
- access mode in which a process may run, 167

/KEY

- qualifier for READ access to indexed file, 213

keyboard

- logical names for keyboard input, 123

keys

- See also* files, indexed.
- alternate key in indexed file, glossary entry, 352
- attributes of primary and alternate keys, 211
- field, glossary entry, 358
- indexed file
 - fields to be indexed for fast lookup, 210
 - matching, 215
- primary, glossary entry, 360

keywords

- abbreviation, how to recognize keywords that may be abbreviated, 201
- arguments, character strings used in many lexical functions, 38
- description and suggestions for use, 201
- glossary entry, 358
- LOOKUP_KEYWORD, listing in subroutine library, 300

labels

- destination for GOTO command, 50
- duplicate, second label replaces first, 51
- glossary entry, 358
- volumes
 - can be labeled during initialization, 249
 - label, glossary entry, 364

languages

- command language interpreter, glossary entry, 354
- debugging, DCL advantages and disadvantages compared with conventional languages, 136

.LE.

- integer operator, arity, result type, and value (table), 32

length

- character string length has upper limit of about 900 characters, 24
- file specification name component, maximum length, 146
- string
 - glossary entry, 358
 - length, glossary entry, 362
- substring assignment format, 42
- terminal input strings, checking, 129

.LES.

- string operator, arity, result type, and value (table), 33

less than (.LT., .LTS.)

- integer operator, arity, result type, and value (table), 32
- string operator, arity, result type, and value (table), 33

less than or equal (.LE., .LES.)

- integer operator, arity, result type, and value (table), 32
- string operator, arity, result type, and value (table), 33

letters

- See* glyphs; strings, character; substrings.

level

- global, symbol creation and properties, 29
- procedure
 - creation and properties of symbols at procedure level, 27
 - glossary entry, 360
 - levels as sources of DCL commands, 59
- procedures, nesting depth limit for calls is 32, 59
- prompt
 - creation and properties of symbols at prompt level, 27
 - level as source of DCL commands, 59
 - level, glossary entry, 361
- subroutine at same procedure level using GOSUB, 64
- symbol
 - determination and use, 26
 - level, glossary entry, 363
 - level, indicates defining context, 21
 - temporary command level following interrupt, characteristics, 105
 - glossary entry, 363

lexical functions

- See also* individual names.
- argument characteristics, 37
- description and use, 36
- explicit substitution using apostrophe, 67
- glossary entry, 358

LIBCALL

- personal command
 - definition, 296
 - to call library of subroutines, 119

library, subroutine

- example of use, 119
- library, complete listings, 296
- use of large procedure as, 118

limitations

- GOSUB, limited to 16 GOSUB levels, 65
- procedure calls may only reach depth up to 32, 59
- procedures may only accept up to eight parameters, 60

.LIS

- file type for listing file produced by Fortran compiler, 146

lists

- See also* arrays.
- creating and manipulating lists with DCL commands, 198
- distribution list, glossary entry, 355
- extracting list items with F\$ELEMENT, 199
- file types, constructing list from a directory, 198
- glossary entry, 358
- Manager for Organized Distribution Lists, sample application, *See* MODL.
- removing duplicates from lists with DCL commands, 198
- search list, glossary entry, 362
- splitting apart, 199

literals

- See also* symbols.
- character string literal represented by use of double quotes, 24
- external representations of data within programs, 23
- glossary entry, 358
- LNMDCL_LOGICAL
 - search list used by SHOW LOGICAL command, description, 180
- LNMDIRECTORIES
 - search list of the two logical name table directories, description, 180
- LNMSFILE_DEV
 - search list for file specification parsing, description, 180
- LNMSGROUP_TABLE
 - group logical table description, 179
- LNMSJOB_TABLE
 - job logical table description, 179
- LNMSPROCESS_DIRECTORY
 - logical name table directory, description, 180
- LNMSPROCESS_TABLE
 - process logical table description, 179
- LNMSYSTEM_DIRECTORY
 - logical name table directory, description, 180
- LNMSYSTEM_TABLE
 - system logical table description, 179
- loading**
 - glossary entry, 358
 - volume loading, glossary entry, 364
- local symbols**
 - See* procedure level.
- LOCAL system identifier
 - description, 231

locking records

- See also* records.
- glossary entry, 361
- sharing files over multiple processes, 222
- /LOG
 - capturing terminal session output with, 142
 - example of use of DCL command qualifiers, 12

log file

- See also* debugging; errors.
- glossary entry, 358

logical data

- See* boolean.

logical names

- See also* names.
- access modes, use by VMS of each mode, 168
- characters allowed in logical name, 162
- concepts and facilities, 161
- defining
 - and using, 162
 - logical names in different access modes, 168
 - root directories with, 170
- deleting with DEASSIGN command, 176
- device specification with, 144
- differences between logical names and symbols, 161
- directories of logical name tables, 180
- displaying values using SHOW LOGICAL, 166
- facility code use for naming compatibility, 112
- for keyboard input, 123
- glossary entry, 358
- inheritance by subprocesses, 264
- iterative translation
 - blocked by * wildcard, 167
 - described, 163
- names of logical name tables, 179
- nonadjacent file spec components, 163
- overview, 15
- privileges required to access each table, 164
- purpose in replacement of device and directory names, 162
- search lists are logical names with more than one value, 172
- standard process and job logical names (table), 169
- system logical names (table), 178
- tables
 - description of four types, 164

logical names (continued)

tables (continued)

- detailed descriptions, 179
- directory, glossary entry, 358
- glossary entry, 358
- job, description of use, 266
- job, glossary entry, 357
- relationship to processes, 161
- UIC protection available, 225
- translating using F\$TRNLNM, 174
- translation, glossary entry, 358
- user address, identifies location and name of user, 171
- using for product parameters, 173
- wildcard * use with logical names, 167

login

- directory
 - glossary entry, 358
 - importance to VMS user, 15
- procedure, description and use of
 - LOGIN.COM, 20

LOGIN.COM

- description and use of login procedure, 20

LOGOUT command

- replacing with a DCL procedure, 3

longword

- glossary entry, 358
- hexadecimal notation, 291
- restriction of size of integers to 32 bits, 23

LOOKUP_KEYWORD

- library routine for manipulating keywords, 201
- listing in subroutine library, 300

loops

- See also* control, structures; GOTO.
- body
 - component of loop, characteristics, 56
 - glossary entry, 359
- description 56
- glossary entry, 358
- infinite
 - example using GOTO command, 51
 - glossary entry, 357
 - preventing in file searches, 154
- terminating
 - multiple termination tests, 58
 - techniques for, 57

lowercase

- See also* strings.
- input to INQUIRE must be inside double quotes to remain lowercase, 124

.LT.

- integer operator, arity, result type, and value (table), 32

.LTS.

- string operator, arity, result type, and value (table), 33

mail

- batch job status report can be tailored and mailed to user, 274
- Manager for Organized Distribution Lists, sample application, 304
- user address, glossary entry, 364

MAIL

- executing commands in mail utility from DCL, 134

mailbox

- virtual device example, 241

main procedure

- example of establishing DCL environment, 113
- glossary entry, 359

MAKEROOT.COM

- procedure for creating new system roots, 4

management

- system management applications for DCL, 4

Manager for Organized Distribution Lists

- See* MODL.

manipulating

- files, DCL usefulness for, xiii
- data in DCL, techniques and tricks, 196

.MAP

- file type for image map produced by linker, 146

masks, protection

- access flags for each access category, 227
- default
 - for directories, 237
 - for files, 237
- glossary entry, 361

master file directory

- glossary entry, 359

/MATCH

- qualifier for READ access to indexed file, 215
- qualifier for SEARCH patterns (table), 194

- matching**
 - access control list, algorithm, 231
 - generic
 - glossary entry, 356
 - match using /MATCH qualifier, 215
 - indexed file keys, 215
 - rules used by DCL to match operators and operand types, 35
- maximum**
 - how to compute with DCL arithmetic commands, 196
- member number**
 - component of UIC, 16
 - glossary entry, 359
- memory**
 - system, associated with VMS processes, 253
- MERGE**
 - file utility, one line description, 159
- messages**
 - conversion of status codes to messages with F\$MESSAGE, 101
 - customizing VMS messages, 85
 - enabling components and returning to entry settings, 85
 - environment, establishing, example of use in main procedure, 114
 - file
 - glossary entry, 359
 - containing error messages for status codes (table), 102
 - four components of a VMS message, 85
 - need to explicitly set file to search for status codes, 101
 - setting, saving, example of use in main procedure, 113
- metalinguistic symbol**
 - description and typography, 8
 - glossary entry, 359
- minimum**
 - how to compute with DCL arithmetic commands, 196
- /MODIFIED**
 - file selection qualifier, one line description, 160
- modifying**
 - See also* appending; creating.
 - access control list, 234
 - character strings, using bit-field assignment for, 44
 - commands, using the substitution facilities, 66
 - default error handler, 95
 - file type, 150
 - symbol values
 - using assignment command for, 22, 26
- MODL (Manager for Organized Distribution Lists)**
 - command descriptions, 306
 - file layout
 - description, 313
 - table of record structures, 314
 - listings
 - command procedures, 315
 - file definition, 351
 - how to use, 305
 - introduction, 304
 - sample application, appendix, 304
 - selection criteria, how to specify, 307
- modulus**
 - how to compute with DCL arithmetic commands, 197
- MOUNT**
 - command to mount a volume, 250
 - qualifiers (table), 251
- mounting volumes**
 - See* volume mounting.
- multinational character set**
 - appendix, 293
- multiple commands**
 - stored in text file as command procedure, 18
- multiplication (*)**
 - integer operator, arity, result type, and value (table), 32
- names**
 - component of DCL file specification, overview, 14
 - device name, glossary entry, 355
 - file specification component
 - characters allowed, 146
 - full description, 146
 - maximum length, 146
 - overview, 14
 - global symbols, conventions for, 29
 - job logical name table, glossary entry, 358
 - logical
 - inheritance of logical names by subprocesses, 264
 - name, overview, 15
 - name table directory, glossary entry, 358
 - name table, glossary entry, 358
 - names for keyboard input, 123

names (*continued*)

- process, setting, 259
- symbol
 - description of use, 21
 - name, glossary entry, 363
 - rules for composition, 22
- terminal name, item of information
 - associated with VMS process, 11
- UNIQUE_NAME, listing in subroutine library, 302
- unique names must be created for temporary files, 190
- user name
 - glossary entry, 364
 - item of information associated with VMS process, 10
 - VMS process, same as user name for first interactive process, 10

naming

- conventions for compatibility with other VMS applications, 111

.NE.

- integer operator, arity, result type, and value (table), 32

negation (-)

- integer operator, arity, result type, and value (table), 32

.NES.

- string operator, arity, result type, and value (table), 33

network

- copying files over a network using DCL, 5
- process
 - description, 254
 - glossary entry, 359

NETWORK system identifier

- description, 231

NL:

- null device, used for discarding output, 80

node

- file specification component
 - full description, 143
 - overview, 14

NOGLOBAL

- keyword to hide global symbols in a procedure, 84

NOLOCAL

- keyword to hide predefined local symbols in a procedure, 84

/NOLOG

- DEFINE qualifier to turn off message concerning previous existence, 165

NOON is not a time of day

- See SET NOON.

not (.NOT.)

- boolean operator, arity, result type, and value (table), 34
- integer operator, arity, result type, and value (table), 32

not equal (.NE., .NES.)

- integer operator, arity, result type, and value (table), 32
- string operator, arity, result type, and value (table), 33

null

- device
 - glossary entry, 359
 - mechanism for discarding SEARCH output, 195
 - NL: used to discard output, 145
 - used for discarding output, 80
- string
 - character string with no characters, 24
 - glossary entry, 359
 - value glossary entry, 359

number

- See integers.

.OBJ

- file type for object file produced by FORTRAN compiler, 146

obtaining

- protection mask values, 230

octal base

- base 8, used in DCL, 23

ON CONTROL_Y

- establish interrupt handler with, 106

ON ERROR/SEVERE_ERROR/WARNING

- establish error handler with, 95

OPEN

- danger of opening file twice, 186
- files must be opened before reading, 184

OPEN/READ

- indexed files, preparation for reading, 213

OPEN/READ/WRITE

- indexed files, preparation for writing, 219

OPEN/SHARE

- sharing file over multiple processes, 221

open for append

- used to add records to end of sequential file, 189
- glossary entry, 359

opening files

- glossary entry, 359
- indexed files, 213
- sequential files, 184

operands

- associated with operator to create new values, 31
- glossary entry, 359

operations

- file manipulation, summary of operations and qualifiers, 158
- search operation, glossary entry, 362

operators

- See also* individual operators.
- arity, number of required operands, 32
- binary
 - description and examples, 32
 - glossary entry, 353
- boolean, arity, result type, and value (table), 34
- character string, arity, result type, and value (table), 33
- evaluation precedence rules, 33
 - modifying with parentheses, 34
 - table, 35
- glossary entry, 359
- integer, table of arity, result type, and value, 32
- precedence, glossary entry, 359
- rules used by DCL to match operators and operand types, 35
- unary
 - description and examples, 32
 - glossary entry, 364
- using in expressions to create new values, 31

optional

- arguments
 - glossary entry, 359
 - lexical functions may accept optional arguments, 37
- parameter, glossary entry, 359

or (.OR.)

- boolean operator, arity, result type, and value (table), 34
- example of use with .EQS., description of precedence, 58
- integer operator, arity, result type, and value (table), 32
- returning status codes with, 95

order of evaluation

- precedence rules for expression, 33
- modifying with parentheses, 34
- table, 35

output

- See also* displaying.
- debugging output using SHOW, 140
- discarding, from a SEARCH command, 195
- displaying during procedure execution, concepts and commands, 75
- formatting terminal output, 78
- procedure, captured with /OUTPUT, 141
- program, redirecting, 79
- /OUTPUT
 - capturing procedure output with, 141
 - utilities qualifier for redirecting output to files, 79

owner

- access category, who can access, 226
- identification through UICs, 16
- UIC
 - associated with files and directories, 226
 - glossary entry, 360

P0 space

- glossary entry, 360
- virtual memory in which program image is run, 253

P1 space

- glossary entry, 360
- virtual memory for processes controlled by DCL command interpreter, 253

P1–P8

- reserved symbols for parameters of a procedure, 60

padding

- blank spaces inserted during substring replacement, 43

paging file blocks

- free, testing with the IF command, 53

parallel execution

- among processes, 262
- of processes using SPAWN /NOWAIT, 263

parameters

- See also* arguments.
- allowing for prompts or batch, 131
- command parameter, glossary entry, 354
- DCL command parameters, description and use, 12
- glossary entry, 360
- handling as an eight element array, 287
- limited to eight for a procedure, 60
- optional, glossary entry 359
- P1–P8 parameters not permitted with GOSUB, 359
- passing product parameters using logical names, 173
- procedure
 - concepts and characteristics, 59
 - parameter, glossary entry, 361
 - product parameter, glossary entry, 361
 - required, glossary entry, 361
 - subroutines use of, 63

parent

- See also* child; process.
- directory, parsing requires special treatment, 150
- process, glossary entry, 360

parsing

- controlling default specs with F\$PARSE, 148
- file specifications
 - complete algorithm, 165
 - completing missing components, 148
 - glossary entry, 360
- protection masks, warning against, 230
- subdirectories requires special treatment, 150
- SYNTAX_ONLY allows skip of existence check, 149

path

- DECnet specification for reaching nodes, 144

pattern

- glossary entry, 360
- matching
 - character string, using F\$LOCATE, 40
 - file searching, 194
 - wildcard use for searching directories, 151

percentage

- how to use DCL commands to calculate percentages, 197

periodic batch jobs

- resubmit themselves on schedule, 279

permission

- to access files, *See* access.

personal commands

- application invocation use of, 112
 - calling subroutines with, 119
 - evaluating with implicit substitution, 70
 - glossary entry, 360
 - predefined
 - blocking example of use in main procedure, 113
 - dangerous to procedure execution, 72
 - protecting procedures
 - from their effects, 83
 - using SET SYMBOL, 84
 - substitution, glossary entry, 360
- PID (process identifier)**
- glossary entry, 360
 - used to specify a process, 257

pool

- system memory set aside for data structures containing control information, 253

precedence

- glossary entry, 360
- operator, glossary entry, 359
- order in which operators are applied, 33
- table of operator precedence, 35

primary key

- description of indexed file, 210
- glossary entry, 360

print jobs

- application for DCL, 3

printable character

- character with a corresponding glyph, 24
- glossary entry, 360

privileges

- access modes assigned to processes, 167
- authorized
 - description and usefulness, 17
 - glossary entry, 353
- description of VMS privilege scheme, 17
- effect on protection-checking scheme, 228
- environment, establishing, example of use in main procedure, 114
- explicit enabling within procedures, 86
- logical name tables, privileges required to access, 164
- process
 - description and usefulness, 17
 - glossary entry, 361
 - setting, 259

- privileges** (*continued*)
 - rights identifier for access, glossary entry, 362
 - SETPRV, privilege that allows a user to enable privileges, 17
 - setting, saving, example of use in main procedure, 113
 - SPAWN requires TMPMBX or PRMMBX privilege, 260
 - system group, glossary entry, 363
 - to access files, *See* access.
- procedures, command**
 - See also* command.
 - call unwind, glossary entry, 360
 - calling with @, 18
 - detailed description, 58
 - cleanup, glossary entry, 360
 - complex, using comments to clarify, 19
 - DCL command procedures, glossary entry, 355
 - debugging using verification, 136
 - default error handler, problems, 94
 - disabling predefined personal commands inside of procedures, 72
 - error handling, 90
 - status code checking methods, 92
 - example of DCL command procedure, 18
 - exit, glossary entry, 360
 - file containing DCL program, 1
 - glossary entry, 360
 - GOSUB allows subroutine at same procedure level, 64
 - hiding predefined local or global symbols, 84
 - input, obtaining user input for an interactive program, 123
 - invoking with @, 18
 - detailed description, 58
 - level
 - as sources of DCL commands, 59
 - creation and properties of symbols at procedure level, 27
 - glossary entry, 360
 - symbols, substring assignment, 42
 - limitation in number of parameters is eight, 60
 - login procedure, description and use, 20
 - main
 - example of use in establishing DCL environment, 113
 - glossary entry, 359
 - mechanism to capture DCL command sequence, 17
 - message components must be enabled, 85
 - obtaining values of protection masks, 230
 - organization requirements, 118
 - parameters
 - glossary entry, 361
 - handling as arrays, 287
 - passing values, allow for prompts or batch, 131
 - personal commands, uses and dangers, 72
 - privileges may need to be enabled by procedure, 86
 - protecting from effects of predefined personal commands, 83
 - redirecting a program's input from SYS\$COMMAND, 133
 - setting DCL environment at beginning and end, 88
 - status code
 - returned by EXIT command, 93
 - use for reporting outcomes, 91
 - status must be checked after every command, 92
 - steps in establishing DCL environment, 115
 - terminating
 - automatically, 58
 - effect on procedure-level symbols, 28
 - with EXIT, 58
 - tracing, 20
 - unwinding nested procedure calls after errors, techniques, 98
 - verification as debugging tool for DCL, 136
 - verifying, 20, 83
- process**
 - See also* child; parent.
 - access modes, full description, 167
 - batch
 - glossary entry, 353
 - versus interactive, 271
 - process described, 254
 - child, glossary entry, 353
 - components associated with VMS processes, 253
 - concepts and facilities, 253
 - creating subprocesses, 260
 - current, obtaining information, 254
 - default protection
 - changing with SET PROTECTION, 238
 - displaying with SHOW PROTECTION, 238

process (*continued*)

- detached process
 - described, 254
 - glossary entry, 355
- environment, cannot obtain information
 - for other than current process, 259
- four sets of privileges associated, 17
- glossary entry, 361
- how to list all processes on the system, 258
- identifier
 - glossary entry, 361
 - PID is used to specify a process, 257
- information associated with VMS
 - process, 10
- interactive
 - description, 10
 - glossary entry, 357
 - versus batch, 271
 - process described, 254
- interrupting, SPAWN to execute commands
 - without terminating interrupted process, 262
- items of
 - information associated with a VMS process, 10
 - DCL information available using F\$ENVIRONMENT, 256
 - process information available using F\$GETJPI, 255
- kinds of processes described, 254
- logical name
 - directory for, 180
 - table, 169
 - table described, 164
 - table, detailed description, 179
 - tables accessible to, 164, 179
 - tables relationship to, 161
- mode
 - glossary entry, 361
 - obtaining to check if process is interactive, 255
- multiple, SPAWN /NOWAIT allows simultaneous execution of processes, 263
- name, setting, 259
- network
 - glossary entry, 359
 - process described, 254
- owner identification UIC described, 16
- parallel execution, 262
- parent, glossary entry, 360

privileges

- glossary entry, 361
- setting, 259
- process-permanent files, *See* process-permanent, files
- scheduler chooses process to run, 253
- setting process environment, commands available (table), 259
- SPAWN requires TMPMBX or PRMMBX privilege, 260
- spawned, characteristics, 261
- subprocess
 - described, 254
 - glossary entry, 363
 - inheritance of symbols and logical names, 264

process-permanent**files**

- batch process (table), 267
- glossary entry, 361
- interactive process (table), 266
- standard files and uses, 265
- SYS\$COMMAND, 123, 265
- SYS\$ERROR, 265
- SYS\$INPUT, 123, 265
- SYS\$OUTPUT, 75, 265
- logical names, importance for command procedures, 169

product

- integer multiplication operator, arity, result type, and value (table), 32
- parameters
 - glossary entry, 361
 - passing parameters using logical names, 173

program

- input, redirecting from SYS\$COMMAND, 133
- interrupting, SPAWN to execute commands
 - without terminating interrupted process, 262
- redirecting output, 79

prologue

- file prologue
 - glossary entry, 356
 - information describing an indexed file, 210

prompt

- character, DCL, dollar sign (\$) as, 11
- DCL prompt, glossary entry, 355
- level
 - as source of DCL commands, 59
 - creation and properties of symbols at prompt level, 27
 - glossary entry, 361

- prompt** (*continued*)
 - level (*continued*)
 - symbols, deleting, 47
 - symbols, distinguishing from global symbols, 29
 - missing parameters can be requested, 131
 - READ, controlling prompt display, 125
- protection**
 - See also* access; captive accounts; security.
 - access
 - algorithm for UIC- and ACL-based protection checking, 239
 - categories based on owner relation to user, 226
 - category, glossary entry, 352
 - control entries, format and description of components, 232
 - blocking access by route of entry, 232
 - default protection ACE, applies protection to new files in directory, 234
 - dialup, access control list, use to deny dialup access, 232
 - files, concepts and facilities, 225
 - masks
 - access flags for each access category, 227
 - default, for directories, 237
 - default, for files, 237
 - format for specifying protection mask, 228
 - glossary entry, 361
 - obtaining values of mask in a procedure, 230
 - setting during creation or with SET FILE, 229
 - mechanism involving
 - access control list, 225
 - UIC code, 225
 - networks, access control list, use to deny network access, 232
 - objects that can be protected, 225
 - privileges alter protection-checking scheme, 228
 - process default
 - changing with SET PROTECTION, 238
 - displaying with SHOW PROTECTION, 238
 - remote, access control list, use to deny remote access, 232
 - rights identifiers defined for access control with AUTHORIZE, 231
 - warning, default process protection may be overridden by ACE, 238
- PURGE
 - file utility, one line description, 159
- purging**
 - temporary files, procedure for, 60
- qualifiers**
 - command qualifier, glossary entry, 354
 - DCL commands, description and use, 12
- queries, user**
 - ASK, listing in subroutine library, 298
 - features of complex ASK routine discussed, 132
 - obtaining user input for an interactive program, 123
- queues**
 - batch job, characteristics, 270
 - job, glossary entry, 358
 - UIC protection available, 225
- quitting**
 - procedures, 61
- quotations**
 - Jimmy Buffett, xvi, 411
 - technical folklore, 9
- quotient**
 - integer division operator, arity, result type, and value (table), 32
- ratios**
 - how to use DCL commands to calculate percentages, 197
- READ
 - description, 125
 - obtains records from open file and stores in symbol, 185
 - preferred input command, reasons discussed, 125
- READ/DELETE
 - deleting records in an indexed file, 220
- READ access
 - description, 227
- READALL
 - privilege that affects protection-checking scheme, 228

- reading**
 - duplicate records from indexed files, 214
 - files
 - existing, 184
 - sequential, 183
 - indexed files, 213
- Record Management System (RMS)**
 - See* RMS.
- records**
 - See also* fields.
 - components of sequential files, 183
 - deleting, in an indexed file, 220
 - file component, overview, 14
 - fixed-length, glossary entry, 356
 - glossary entry, 361
 - indexed files may contain fixed- or variable-length records, 210
 - locked
 - overriding with READ/NOLock, 223
 - ways to unlock locked records, 223
 - locking
 - for use with shared files, 222
 - glossary entry, 361
 - sample application records, description of fields, 314
 - size limit can be extended with /SYMBOL qualifier, 219
 - structures
 - field types, definition symbols and characteristics, 207
 - file format alternative to free-format text, 183
 - glossary entry, 361
 - introduction, 205
 - /SYMBOL, qualifier for WRITE to allow larger record size, 219
 - update a record, glossary entry, 364
 - updating locked records, how to wait for them to be unlocked, 223
 - variable
 - with fixed control format, glossary entry, 364
 - record, glossary entry, 364
 - versus fixed-length records, 184
- recursion**
 - DCL permits, 59
- redirecting**
 - program input
 - forcing input from SYS\$COMMAND, 133
 - using DEFINE, 134
 - program output, 79
- reduction (-)**
 - string operator, arity, result type, and value (table), 33
- regression test**
 - DCL good choice in software development, 6
- remainder**
 - how to compute with DCL arithmetic commands, 197
- REMOTE system identifier**
 - description, 231
- removing**
 - See* deleting.
- RENAME**
 - file utility, one line description, 159
- replacement**
 - See* substitution.
- replacing**
 - symbol values using assignment command, 26
- representation**
 - internal and external representation of data, 23
- required parameter**
 - glossary entry, 361
- responses**
 - obtaining user input for an interactive program, 123
- \$RESTART**
 - batch jobs, controlling restart entry point, 277
- restartable batch jobs**
 - glossary entry, 361
 - \$RESTART and BATCH\$RESTART control restart entry, 277
- restarting**
 - batch jobs, 276
- resubmitting**
 - batch jobs, 279
- resultant file spec**
 - file spec returned by parsing operation, 148
 - glossary entry, 361
- return**
 - point
 - glossary entry, 362
 - place where DCL continues after returning from subroutine, 59
 - value
 - glossary entry, 362
 - used in place of lexical function in expressions, 36

RETURN
 use with GOSUB command, 65

rights identifiers
 maintained using AUTHORIZE, 231
 glossary entry, 362
 mnemonic identifiers for UICs, 16

RMS (Record Management System)
 facility for reading and writing files, 183
 glossary entry, 361

root
 directory
 defining with logical names, 170
 glossary entry, 362
 master file directory, root of entire
 directory tree, 145

rounding
 how to compute with DCL arithmetic
 commands, 197

run, running
See also control, structures; execute;
 flow-of-control; invoke.
 batch jobs, 270
 command procedures 18, 58
 DCL applications, directory and personal
 command conventions for, 112
 glossary entry, 362

sample application
 appendix with listing and description of
 use, *See* MODL.

scheduler
 chooses process to run, 253
 glossary entry, 362

screens
 checking for ANSI control sequence
 acceptance before screen
 formatting, 246

script
 command procedure, defined, 1

search
 lists
 glossary entry, 362
 logical names with more than one
 value, 172
 operation, glossary entry, 362

SEARCH
 file utility, one line description, 159
 qualifiers and method of matching, 194
 using to search files, 193

searching
 directories
 F\$SEARCH useful in checking for file
 existence, 152
 stream ID useful in identifying
 context of each search, 155
 using wildcards to facilitate, 151

files
 contents can be searched, 193
 loop to count files, precaution against
 infinite loop, 154
 multiple files, suggested use, 155
 string, using F\$LOCATE, 40
 symbol value
 complete DCL algorithm for, 30
 partial DCL algorithm for, 28

security
See also access; captive accounts;
 protection.
 access modes give privileges to processes,
 167
 captive accounts use to ensure, 122
 file protection, 225
 concealing root directory logical name
 values, 170
 description of VMS privilege scheme, 17
 disable verification to conceal contents of
 procedure, 138

selection
 file selection qualifiers (table), 160

separator lines
 use of \$ alone on line in DCL command
 procedure, 19

sequential
 execution
 form of DCL flow of control, 49
 glossary entry, 362

files
 concepts and operations, 183
 glossary entry, 362

SET
 command to set process environment,
 259

SET ACL
 command qualifiers (table), 235

SET DEFAULT
 command to change working directory,
 15
 extending with a DCL procedure, 3
 file utility, one line description, 159

SET DIRECTORY
 file utility, one line description, 159

SET FILE
 file utility, one line description, 159
 protection mask setting, 229

SET HOST
 second log session for recording debug output, 142

SET MESSAGE
 explicitly set file to check for status code interpretation, 103

SET NOCONTROL
 blocking interrupts during initialization with, 108

SET NOON
 used to disable status checking, 96

SET NOVERIFY
 disabling procedure verification, 137

SET ON
 used to enable status checking, 96

SET SYMBOL
 keywords to hide symbol definitions, 84

SET VERIFY
 enabling procedure verification, 137
 tracing procedures for debugging, 83

SETPRV
 privilege that allows users to enable privileges, 17

setting
 default directory protection using access control entries, 234
 process
 environment, 259
 name, 259
 privileges, 259
 protection mask, 229
 status codes with DN and .DR. logical operator, 95

severity
See status code severity.

SEVERITY
 reserved symbol for recording success of command, 92

shadowing
 symbols, 28

shareable images
 call from DCL procedure not supported, 3

sharing
 files over multiple processes, 221

shortcuts
See hints.

SHOW DEFAULT
 file utility, one line description, 159

SHOW DEVICE
 listing system devices, 245

SHOW LOGICAL
 logical names, displaying values, 166

SHOW SYMBOL
 use following interrupt, 105

SIGNAL
 listing in subroutine library, 301
 message and error handling subroutine, characteristics and use, 120

signal on error
 glossary entry, 362

simultaneous
 execution of process using SPAWN /NOWAIT, 263
 searches, description of, 155

/SINCE
 file selection qualifier, one line description, 160

software development
 DCL as appropriate management tool, 5

SORT
 file utility, one line description, 159

source code management
 DCL as appropriate language for, 5

SPAWN
 qualifiers (table), 261
 temporary command level, use after interrupt, 105
 used to execute commands without terminating interrupted process, 262

SPAWN /NOWAIT
 allows simultaneous execution of processes, 263

spawning a subprocess
 description, 260
 examples, 262

specification
 file, *See* files.
 parsing a file spec, glossary entry, 360
 resultant file spec, glossary entry, 361
 wildcard spec, glossary entry, 364

starting DCL command procedures
 using @ command, 18
 detailed description, 58

statements
 glossary entry, 362
 used to mean commands in most languages, 25

status

See also checking; cleanup; debugging; errors.

batch job, sending mail about status, 274
code

- checking, methods for, 92
- checking with ON command, 95
- conversion to text message with F\$MESSAGE, 101
- defining, example of use in main procedure, 113
- disabling automatic checking with SET NOON, 96
- explicit checking, examples, 96
- facility code use for naming compatibility, 112
- format of fields described (table), 91
- glossary entry, 362
- procedure, creating, 93
- returned by program to indicate success or failure, 90
- searching message files for status codes, 101
- severities described (table), 91
- SIGNAL, message and error handling subroutine, characteristics and use, 120
- SIGNAL, listing in subroutine library, 301

command outcome must be checked within procedures, 92

\$STATUS

reserved symbol for recording status of command, 92

STOP

canceling a procedure from temporary command level, 105

STOP/REQUEUE

batch restart option, 276

stopping

- procedure execution, 61
- programs, <CTRL/y>, program interrupt handling, 104

stream ID

- glossary entry, 362
- multiple file search context identifier for F\$SEARCH, 155

string

assignment of substrings, 42
character

- altering bits in, 44
- control characters not representable in, 24

- glossary entry, 353
- length as number of individual characters, 24

- length has upper limit of about 900 characters, 24

- literal represented by use of double quotes, 24

- manipulation of lexical functions, 40

- null string has no characters, 24

- operators, arity, result type, and value (table), 33

- representation and use, 24

- terminal input validating, 129

checking for valid integer in input string, 128

concatenation operator (+), arity, result type, and value (table), 33

creating

- more than one character with bit-field assignment, 45

- one-character strings using bit-field assignment, 45

- strings with multiple characters using bit-field assignment, 45

equivalence string, glossary entry, 356

formatting output using F\$FAO directives, 78

glossary entry, 362

hints

- about manipulating character strings, 42

- about substring assignment, 44

indexes

- creating arrays with symbolic indexes, 288

- glossary entry, 362

length, glossary entry, 362

- lexical functions to manipulate strings, 40

- noninteger strings converted to integers become zero, 36

- null, glossary entry, 359

- numbering of bits in character strings, 44

- operators, arity, result type, and value (table), 33

- reduction operator (-), arity, result type, and value (table), 33

- representation and use, 24

- string** (*continued*)
 - searching, using F\$LOCATE, 40
 - substitution of symbol values using
 - apostrophe, 67
 - terminal input, checking length, 129
 - using loops to remove spaces in a string, 57
- structures**
 - application, overview of complex design, 111
 - extracting fields from, 209
 - glossary entry, 363
 - record
 - creating structures by storing values
 - in character string, 208
 - displaying record structure may
 - cause strange results, 208
 - structure, glossary entry, 361
 - structured records, introduction, 205
 - symbols defining position, size, and type of field, 206
 - type symbols for fields, 207
- subdirectories**
 - glossary entry, 363
 - parsing requires special treatment, 150
 - tree structure allowed in directory
 - specifications, 145
- SUBMIT**
 - command
 - for queueing batch job, 271
 - qualifiers (tables), 272–273
- submitting**
 - batch jobs, 271
- subprocedures**
 - custom interrupt handling of, 108
 - defined, 59
 - glossary entry, 363
 - interrupt handler behavior, potential
 - problems 105
 - with cleanup code, example of use in
 - subprocedures, 117
 - without cleanup code, example of use in
 - subprocedures, 116
- subprocess**
 - creating with SPAWN command, 260
 - glossary entry, 363
 - inheritance of symbols and logical names, 264
 - input at spawn time from command string
 - or procedure file, 261
 - list of inherited and not-inherited environment properties, 260
 - parent process execution controlled by
 - /WAIT, 262
 - process, description, 254
 - termination, possible conditions, 265
- SUBROUTINE**
 - use with CALL command, 63
- SUBROUTINE-LIBRARY.COM**
 - complete listing, 297
- subroutines**
 - See also* calling.
 - characteristics and commands that
 - invoke, 63
 - defined, 59
 - glossary entry, 363
 - GOSUB allows subroutine at same procedure level, 64
 - invoking with
 - @, 58
 - CALL, 62
 - GOSUB, 62
 - library
 - complete listings, 296
 - example of use, 119
 - use of large procedure as, 118
 - SUBROUTINE command use with CALL, 63
 - summary table of facilities, 64
 - with cleanup code, example of use in
 - subprocedures, 117
 - without cleanup code, example of use in
 - subprocedures, 116
- subscript**
 - See also* arrays.
 - array, glossary entry, 363
 - arrays can be simulated in DCL, 284
- substitution**
 - ampersand
 - explicit substitution of text in DCL
 - command, glossary entry, 352
 - used in concert with apostrophe
 - substitution, 73
 - apostrophe
 - creating DCL arrays, techniques and
 - problems, 285
 - description, 66
 - explicit substitution of text in DCL
 - command, glossary entry, 352
 - bit-field assignment using integer values, 45
 - concepts and operators, 66
 - explicit, using apostrophe substitution
 - for, 66
 - glossary entry, 363

substitution (*continued*)

- implicit
 - in evaluation of personal commands, 70
 - requirements for, 70
- INQUIRE input apostrophe substitution, 124
- logical name values substituted, 161
- personal command substitution, glossary entry, 360
- principle underlying, 66
- substring assignment to replace parts of strings, 43
- textual, contrasted with assembler macros and C preprocessor, 66
- wildcards
 - file operations use of, 158
 - logical names allow use of * wildcard, 167
 - pattern matching templates for parts of file spec components, 151
 - spec, glossary entry, 364
 - using with F\$SEARCH, 153
 - value of arbitrary expression, 69

substrings

- assignment, 42
- extracting substring with F\$EXTRACT, 41
- glossary entry, 363

subtraction (-)

- integer operator, arity, result type, and value (table), 32

suggestions

See hints.

sum

- integer addition operator, arity, result type, and value (table), 32

superceding

- files in personal over public directories, using search lists for, 173

supervisor

- access mode in which a process may run, 167

suspending program execution

- program interrupt handling, 104

/SYMBOL

- qualifier for WRITE to allow larger record size, 219

symbolic arrays

- creation and use, 288
- glossary entry, 363

symbols

- See also* assignment; literals; values.
- case, DCL converts all symbols to uppercase, 22
 - checking
 - the type or existence with F\$TYPE, 39
 - validity, VERIFY_SYMBOL, listing in subroutine library, 303
 - creating
 - global, 29
 - procedure level, 27
 - prompt level, 27
 - using assignment command for, 22, 26
 - data type, associated with value not symbol, 26
 - description and items of information associated, 21
 - differences between logical names and symbols, 161
 - evaluating, using apostrophe substitution in, 66
 - explicit substitution using apostrophe, 67
 - facility code use for naming compatibility, 112
 - global
 - assigning the value, 29
 - creation prevented by NOGLOBAL keyword, 84
 - deleting, 47
 - distinguishing from prompt-level symbols, 29
 - examples of use, 30
 - existence limited to login period, 31
 - facility code use for naming compatibility, 112
 - glossary entry, 357
 - level, creation and properties, 29
 - name for symbols created at global level, 29
 - obtaining the value, 29
 - substring assignment on, 44
 - glossary entry, 363
 - inheritance of symbols by subprocesses, 264
 - level, glossary entry, 363
 - local, *See* procedure level.
 - metalinguistic
 - description and typography, 8
 - glossary entry, 359
 - name
 - glossary entry, 363
 - rules for composition, 22

- symbols** (*continued*)
 - procedure-level
 - creation and properties of symbols at procedure level, 27
 - effect of terminating a procedure on, 28
 - substring assignment, 42
 - prompt-level
 - creation and properties of symbols at prompt level, 27
 - deleting, 47
 - distinguishing from global symbols, 29
 - keywords to hide symbol definitions, 84
 - reserved, P1–P8 procedure-level symbols for procedure parameters, 60
 - shadowing, 28
 - SHOW SYMBOL, use following interrupt, 105
 - table
 - arrays may require symbol table expansion, 290
 - glossary entry, 363
 - values
 - assignment using assignment command, 22
 - complete search algorithm for, 30
 - glossary entry, 363
 - modifying using assignment command, 22, 26
 - partial search algorithm for, 28
 - replacing using assignment command, 26
 - variables, symbols are used as variables in DCL, 21
- syntax**
 - command syntax, glossary entry, 354
 - format and rules governing format of DCL commands, 13
- SYNTAX_ONLY**
 - parsing without existence check, 149
- SYS\$BATCH**
 - name of system batch job queue, 270
- SYS\$COMMAND**
 - forcing program input to use, 133
 - process logical name, brief description (table), 169
 - process-permanent file for input from procedure or keyboard, 123
- READ** use of, 125
- sources for detached and subprocesses of
 - batch procedure, 267
 - interactive procedure, 266
- SYS\$DISK**
 - working directory device name, 145
 - process logical name, brief description (table), 169
- SYS\$ERROR**
 - destinations for detached and subprocesses of
 - batch procedure, 267
 - interactive procedure, 266
 - process logical name, brief description (table), 169
 - redirection of screen output using, 80
- SYS\$INPUT**
 - process logical name, brief description (table), 169
 - process-permanent file for input from procedure or keyboard, 123
 - sources for detached and subprocesses of
 - batch procedure, 267
 - interactive procedure, 266
 - work around for input from procedure, 133
- SYS\$LOGIN**
 - job logical name, brief description (table), 169
- SYS\$MESSAGE**
 - directory, status code messages stored in, 101
- SYS\$OUTPUT**
 - destinations for detached and subprocesses of
 - batch procedure, 267
 - interactive procedure, 266
 - process logical name, brief description (table), 169
 - process-permanent file for video output, 75
 - redirecting with DEFINE command, 80
- SYS\$SCRATCH**
 - job logical name, brief description (table), 169
 - temporary file logical name, 190
- SYS\$PRV**
 - privilege that affects protection-checking scheme, 228
- system**
 - access category, who can access, 226
 - devices, listing with SHOW DEVICE, 245
 - directory organization, overview, 177
 - displaying CPU time of all users, use of an array, 288
 - F\$GETSYI lexical function to get system information, 37

- system** (*continued*)
 - groups
 - access privileges for certain UICs, 16
 - glossary entry, 363
 - identifiers, category of rights identifiers, 231
 - logical names
 - application naming convention use of, 112
 - directory for, 180
 - table, 178
 - table described, 164
 - table, detailed description, 179
 - management, applications for DCL, 4
 - roots, creating new system roots using DCL, 4
- tables**
 - job logical name table, glossary entry, 357
 - logical name table
 - directory, glossary entry, 358
 - glossary entry, 358
 - described, 161
 - symbol table, glossary entry, 363
- tape**
 - used as device, 144
- techniques**
 - See* hints.
- telephone directory**
 - possible use for DCL, 4
- temporary**
 - See also* files.
 - command level, glossary entry, 363
 - file, glossary entry, 363
- terminal**
 - characteristics can be determined with F\$GETDVI, 246
 - characteristics, glossary entry, 363
 - checking for acceptance of ANSI control sequences, 246
 - controlling the behavior, 246
 - input
 - obtaining with INQUIRE, 124
 - READ, preferred input command, 125
 - name, item of information associated with VMS process, 11
 - output, formatting, 78
 - sending special characters to control a terminal, 76
 - session output, capturing with LOG, 142
 - specified as a device name, 144
 - visual display methods during procedure execution, 75
- terminating**
 - loops
 - multiple termination tests, 58
 - techniques for, 57
 - procedures
 - automatically, 58
 - effect on procedure-level symbols, 28
 - using EXIT, 58
 - subprocesses, possible conditions, 265
- test case library**
 - possible use for DCL in software development, 6
- text**
 - See also* image data.
 - DCL command procedure stored in text file, 18
 - displaying
 - contents of file, 192
 - large amounts with TYPE command, 134
 - text on video screen using WRITE command, 75
 - field type, definition symbol and characteristics, 207
 - fields in record structures, created with substring assignment, 208
 - file, glossary entry, 363
 - formatting output using F\$FAO directives, 78
 - free-format text, file format alternative to record structure, 183
 - OPEN/APPEND allows WRITE to add records to text file, 189
- THEN**
 - use with IF command, 52
- time**
 - See also* date.
 - absolute time
 - format to represent specific points in time, 203
 - VMS format, glossary entry, 352
 - combination time
 - format using absolute and delta times, 204
 - VMS format, glossary entry, 353
 - comparison
 - of two times, 203
 - time, format to represent times to be compared, 203
 - time, VMS format, glossary entry, 354

time (*continued*)

- converting absolute time to comparison time, 203
- delta time
 - format for showing difference between two times, 204
 - VMS format, glossary entry, 355
- difference between times, how to calculate, 204
- displaying CPU time of all users, use of an array, 288
- F\$CVTIME, converts between time formats, 203
- glossary entry, 364
- manipulating using DCL commands, 202
- omission of certain time fields is permitted, 205
- READ prompt string can display current time, 125
- TODAY, absolute time in DCL, 203
- TOMORROW, absolute time in DCL, 203
- truncating current time to minutes field, 208
- YESTERDAY, absolute time in DCL, 203
- TODAY
 - absolute time in DCL, 203
- TOMORROW
 - absolute time in DCL, 203
- tracing**
 - See also* debugging; errors.
 - command procedures, using SET VERIFY, 20
 - procedures using SET VERIFY for debugging, 83
- translating, logical names**
 - using F\$TRNLNM, 174
 - value look up in, 163
- /TRANSLATION_ATTRIBUTES
 - concealing root directory logical name values with, 170
- translation**
 - iterative, glossary entry, 357
 - logical name translation. glossary entry, 358
- trash**
 - device NL: used to discard output, 145
- tricks**
 - See* hints.
- TRUE boolean value**
 - odd integers and certain strings used to represent, 25
 - literal must be established with assignment command, 25

truncations

- allowed abbreviation of DCL verbs, 13
- rules of abbreviation in DCL command verbs, 11

types

- changing file type, 150
- data
 - associated with value not symbol, 26
 - boolean, description and use, 25
 - character string representation and use, 23
 - integer representation and use, 23
 - type, glossary entry, 355
 - types in DCL, 22
- file specification component
 - full description, 146
 - overview, 14
- noninteger strings converted to integers become zero, 36
- rules used by DCL to match operators and operand types, 35
- standard file types for different file contents, 146
- structures use codes for type of fields, 207
- symbols, checking using F\$TYPE, 39

TYPE

- file utility, one line description, 159
- /PAGE qualifier displays text screenful at a time, 192
- text display using TYPE command, 134

typographic conventions, 7

UIC (user identification code)

- See* user identification code.
- glossary entry, 364

unary operators

- See also* binary operators.
- description and examples, 32
- glossary entry, 364

unconditional goto

- See also* control, structures.
- GOTO command as control structure, 50

UNDEFINE

- suggested personal command for de-assigning logical names, 176

undefining

- See also* defining.
- logical names using personal command, 176

UNIQUE_NAME

- listing in subroutine library, 302
- subroutine for creating a name for temporary file, 190

unwinding

- See also* calling; cleanup; debugging; errors.
- nested procedure calls after errors, techniques for, 98
- procedure call unwind, glossary entry, 360

updating

- alternate key fields requires use of “changes” attribute at creation, 220
- indexed files, updating a file using WRITE/UPDATE command, 220
- locked records, how to wait for them to be unlocked, 223
- records, glossary entry, 364

user

- access mode in which a process may run, 167
- address
 - glossary entry, 364
 - identifies location and name of user, 171
- identification code (UIC)
 - avoid changing UIC within a procedure, 87
 - characteristics, 87
 - compared with access control list (ACL), 225
 - description of purpose, 16
 - file protection use of UIC, 225
 - glossary entry, 364
- name
 - glossary entry, 364
 - item of information associated with VMS process, 10
 - process name for first interactive process same as user name, 10

/USER_MODE

- qualifier to define logical name for duration of image, 168
- temporary redirection of input, 134
- temporary redirection of output, 80

utilities

- file manipulation (table), 159
- passing input to programs from procedure, 134

Utley, Michael

- quotation, xvi

validating

- See also* checking; errors; verification.
- checking validity of input, 128
- file specification with F\$PARSE, 153
- terminal input
 - booleans, 130
 - character strings, 129
 - integers, 128

values

- See also* assignment; symbols.
- default
 - glossary entry, 355
 - in lexical functions with optional arguments, 37
- global symbols
 - assigning, 29
 - obtaining, 29
- logical names
 - displaying with SHOW LOGICAL, 166
 - equivalence string as, 162
- null, glossary entry, 359
- prompting for procedure parameters in batch, 131
- return value, glossary entry, 362
- returned from lexical functions, use in expression, 36
- symbol
 - characteristics of, 22
 - complete search algorithm for, 30
 - modifying using assignment command, 22, 26
 - partial search algorithm for, 28
 - value, glossary entry, 363

variable

- character matching, wildcard spec, glossary entry, 364
- with fixed control (VFC), record format, glossary entry, 364

variable-length record

- allowed in indexed files, 210
- characteristic of text files, 184
- glossary entry, 364

variables

- See* symbols.

VAXcluster

- device names are segmented, 242

verbs

See also command.
command verb, glossary entry, 354
description and use of DCL command verbs, 11

verification

See also debugging; errors; validating.
command procedures, using SET VERIFY for tracing, 20

debugging

method, glossary entry, 364
tool for DCL, 136

enabling

before entering procedure, 137
from inside procedure, 137

image

displaying image data lines, 137
verification, setting from inside procedure, 139

procedure

displaying command lines, 136
verification, setting from inside procedure, 139

VERIFY_SYMBOL

listing in subroutine library, 303

verifying procedures

See verification.

versions

file

accessing highest and lowest versions, format, 147
range of version numbers is 1 to 32767, 147
specification component, description, 147
specification component, overview, 14
0 gives highest version, 147
-1 gives next highest version, 147
-32768 gives lowest version, 148

VMS

See VMS versions.

VFC (variable with fixed control)

record format, glossary entry, 364

video screen output methods during procedure execution, 75

virtual devices

description and examples, 241
glossary entry, 364

viruses

possible problem for DCL, caution, 6

VMS

designing compatible command procedure applications, 111
documentation set listed, 8
note about differences in versions, 7
process
 items of information associated, 10
review of basic concepts, 10
versions
 CALL introduced in Version 5, 62
 compound IF introduced in Version 5, 52, 54
 lack of compound IF command prior to Version 5, work-arounds, 54
 lexical function to obtain, 38
 replacement of F\$LOGICAL by F\$TRNLNM in Version 4, 175

VMSINSTAL

choosing DCL for software installation, xiii
file manipulation, example of DCL use, 5
lack of second "L" in, annoyance about, xiii

volume

initialization, to format disk or tape,
 glossary entry, 364
label, glossary entry, 364
loading, glossary entry, 364
magnetic medium on which information is stored, 241
medium to store permanent information,
 glossary entry, 364
mounting
 to allow system to recognize a device, 250
 glossary entry, 364
owner UIC described, 16
steps performed in mounting, 251

warning messages

SIGNAL subroutine treats as informational, 121

warnings

See also hints.
angle brackets (<>) must be accepted as directory delimiters, 146
arrays may require symbol table expansion, 290
avoid
 ampersand substitution, list of problems, 73

- warnings** (*continued*)
 - avoid (*continued*)
 - changing UIC within a procedure, 87
 - SET NOCONTROL, 109
 - SET NOVERIFY from inside a procedure, 138
 - using abbreviations in command procedures, 20
 - using operators with mixed operand types, 36
 - characters must be same case before comparison, 127
 - comments should be moved to end of procedure to avoid slowdown, 296
 - common mistakes using apostrophe substitution, 69
 - debug commands cost execution time even when inactive, 141
 - default
 - answers should not contain dangerous material, 126
 - directory, do not change in a procedure, 88
 - error handler does not always detect syntax errors, 95
 - interrupt handler would be confusing to naive user, 106
 - DEFINE and ASSIGN have reverse order of parameters, 165
 - deleting records from indexed file, avoid generic matching, 221
 - determining parent of arbitrary directory difficult, 150
 - device names cannot be trusted to show type of device, 246
 - displaying record structure on screen may cause strange results, 208
 - F\$ELEMENT should not be used to extract component from file spec, 200
 - file
 - names with hyphens cannot be used as symbol names, 289
 - spec manipulation involving rooted directory difficult, 150
 - global symbols, creation prevented by NOGLOBAL keyword, 84
 - GOSUB requires EXIT to avoid dropping into subroutine, 65
 - GOTO commands shunned, reasons discussed, 53
 - indexed file, open for writing with /READ /WRITE qualifiers, 219
 - INQUIRE is idiosyncratic and should be used sparingly, 124
 - “job” does not necessarily mean batch job, 266
 - labels, avoid using duplicate labels, 51
 - limit boolean input to yes or no, 130
 - P1–P8 reserved for names of procedure parameters, 60
 - passing parameters using logical names, 173
 - personal commands must be prevented from influencing a procedure, 83
 - predefined personal commands may damage procedures, 72
 - protection
 - default process protection may be overridden by ACE, 238
 - mask does not include all information about file protection, 230
 - record size should always be extended with /SYMBOL qualifier, 219
 - SPAWN requires TMPMBX or PRMMBX privilege, 260
 - specifying local node on file spec has network overhead, 144
 - status codes are not returned by all DCL commands, 92
 - subprocesses should inherit symbols and logical names with caution, 264
 - SYS\$SCRATCH is not automatically deleted, 192
 - updating indexed file alternate key fields requires use of “changes” attribute during creation, 220
 - WRITE command does not create normal text file, 188
- WHILE loops**
 - See also GOTO.
 - DCL is missing WHILE loops, 3
- whitespace**
 - F\$EDIT can be used to remove, 128
 - glossary entry, 364
 - removed from input to INQUIRE unless double quotes, 124
- wildcards**
 - See also substitution.
 - file operations use of, 158
 - logical names allow use of * wildcard, 167
 - pattern matching templates for parts of file spec components, 151
 - spec, glossary entry, 364
 - using with F\$SEARCH, 153

windowing

DCL has no windowing capabilities, 3

windows

virtual device example, 241

working directory

See also directories.

defined as default file directory, 88

glossary entry, 365

world

See also access; protection; security.

access category, who can access, 226

worms

possible problem for DCL, caution, 6

WRITE

displaying text with, 75

files, creating a new file, 188

limitations on record size and expression values, 188

output of WRITE is not normal text file, 189

WRITE/UPDATE

indexed files, using write command to update file, 220

WRITE access

description, 227

writing

files

sequential, 183

indexed, 219

XDA (eXample DCL Application)

introduced as fictitious application for purposes of illustration, 7

YESTERDAY

absolute time in DCL, 203

zero

local DECnet node is named zero, 144

noninteger strings converted to integers become zero, 36

version number giving highest version, 147

0::

DECnet local node is named zero, 144

[000000]

master file directory, root of entire directory tree, 145

1B

hexadecimal value of ESC character, control sequence initiator for seven-bit terminals, 76

9B

hexadecimal value of CSI character, control sequence initiator for eight-bit terminals, 76

! (exclamation point)

comment, glossary entry, 354

DCL command procedures, use for comments in, 19

use in F\$FA0 directives, 78

!!! (triple exclamation point)

flag to help locate temporary debugging commands, 138

" (double quote)

DCL command parameter use, 13

DCL command qualifier syntax use, 13 rules for use in parameters, 61

"" (paired double quotes)

default answer trigger for READ, 127

INQUIRE, removes quotes but leaves

whitespace and lowercase, 124 matched double quotes used to denote string literal, 24

used inside double quotes to represent one double quote, 24

\$ (dollar sign)

alone on line ignored in DCL command procedures, 19

DCL command procedure, use to indicate command, 18

DCL prompt character, use by DCL command interpreter, 11

device name segment divider, 242

% (percent)

wildcard for one character in a file spec component, 151

& (ampersand)

explicit substitution of text in DCL command, glossary entry, 352

used in concert with apostrophe substitution, 73

' (apostrophe)

explicit substitution of

text in DCL command, glossary entry, 352

symbol or lexical function values, 66 interpretation when input to INQUIRE, 124

- ' (**apostrophe**) (*continued*)
 - trailing, when required in apostrophe substitution, 67
- '' (**double apostrophe**)
 - use inside string literals to cause substitution, 67
- () (**parentheses**)
 - DCL command qualifier syntax use, 13
 - lexical functions format requirement, 36
 - modifying operator precedence with, 34
- * (**asterisk**)
 - integer operator, arity, result type, and value (table), 32
 - wildcard
 - for characters in file spec components, 151
 - use in logical names blocks iterative translation, 167
- + (**plus**)
 - integer operator, arity, result type, and value (table), 32
 - string operator, arity, result type, and value (table), 33
- , (**comma**)
 - DCL command
 - parameter syntax use, 13
 - qualifier syntax use, 13
 - use as placeholder for optional arguments, 38
- (**minus**)
 - integer operator, arity, result type, and value (table), 32
 - line continuation symbol in DCL, 13
 - string operator, arity, result type, and value (table), 33
- . (**period**)
 - separator for subdirectories in file specification, 146
 - type component of DCL file specification syntax use, 14
- ... (**ellipsis**)
 - wildcard for directory subtree, 151
- / (**slash**)
 - integer operator, arity, result type, and value (table), 32
 - use in DCL command qualifiers, 12
- : (**colon**)
 - appended to prompt by INQUIRE, 124
 - device component of DCL file specification syntax use, 14
 - label for GOTO command destination, used in syntax, 50
- :: (**double colon**)
 - distinguishing DECnet node name, 143
 - node component, DCL file specification syntax, 14
- := (**colon equal**)
 - substring assignment character for procedure level, 42
- ::= (**colon double equal**)
 - substring assignment character for global symbols, 44
- ;
(**semicolon**)
 - version component of DCL file specification syntax use, 14
- <> (**angle brackets**)
 - alternate characters for delimiting directory name, 146
- = (**equal sign**)
 - assignment command described and discussed, 25
 - assignment command for creating symbols, 22
 - assignment statement meaning different from equality comparison, 26
 - bit-field assignment operator for current procedure level, 45
 - DCL command qualifier syntax use, 13
 - use in setting value to DCL command qualifiers, 12
- == (**double equals**)
 - assignment command for symbols at global level, 29
 - bit-field assignment operator for global level, 45
- @ (**at-sign**)
 - command, compared with CALL command, 62
 - procedure invocation by @ creates procedure level symbols, 27
 - summarized in table of subroutine facilities, 64
 - use to invoke a DCL command procedure, 18
 - detailed description, 58
- [] (**square brackets**)
 - bit-field assignment statement syntax, 45
 - directory component of DCL file specification syntax, 14
 - substring assignment statement syntax, 42
- [] (**empty brackets**)
 - stand for working directory, 145
- [-]
 - parent directory of default working directory, 146

Why don't you wander and follow *la vie dansante*?

— Jimmy Buffett
Will Jennings
Michael Utley

From the Author's Preface:

"I have three goals in writing this book. The first is to help people make an intelligent choice between DCL and more conventional programming languages. . . .

"My second goal is to offer a programming language to computer users who are not conversant with conventional languages. Secretaries, administrative assistants, word processors, database administrators, system managers: all these people are more challenged and productive if they can create new programs. . . .

"My final goal is to make the DCL programming process enjoyable for everyone. Programming is satisfying when you understand the features of the language, know how to apply them, and have the ability to build upon previous work to match new situations."

Paul C. Anagnostopoulos is an independent consultant specializing in the VMS operating system. He was formerly a member of the VAX/VMS Development Group at Digital Equipment Corporation, where he also worked with Customer Services and the Artificial Intelligence Group. He is a graduate of Brown University and has done graduate work at MIT.

digital

Digital Press

12 Crosby Drive
Bedford, Massachusetts 01730

ORDER NUMBER EY-C168E-DP
ISBN 1-55558-023-8
PH ISBN 0-13-940256-X

Cover design by Sandra Calef