VAX COBOL User Manual

Order Number: AA–H632E–TE

This manual explains how to develop VAX COBOL programs; it also describes the features of the language and how to use VMS features from VAX COBOL.

Revision/Update Information:	This revised manual supersedes *VAX COBOL User Manual* (Order Number: AA–H632D–TE).
Operating System and Version:	VMS Version 5.0 or higher
Software Version:	VAX COBOL Version 4.3

digital equipment corporation
maynard, massachusetts

First Printing, June 1980
Revised, October 1984
Updated, April 1985
Updated, February 1986
Updated, February 1987
Revised, January 1988
Updated, December 1988
Revised, December 1989

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

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

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

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK5185

Contents

Preface . . . xxvii

Summary of Technical Changes . . . xxix

Part I Developing VAX COBOL Programs

Chapter 1 Overview of VAX COBOL

Chapter 2 Developing VAX COBOL Programs at DCL Command Level

2.1 Choosing a Reference Format . . . 2–1
2.1.1 Terminal Reference Format . . . 2–1
2.1.2 ANSI Reference Format . . . 2–2
2.1.3 Converting Between Reference Formats . . . 2–2

2.2 DCL Commands for Program Development . . . 2–2

2.3 Creating a VAX COBOL Program . . . 2–5

2.4 Using the COPY Statement in Your Source Program . . . 2–5

2.5 Compiling a VAX COBOL Program . . . 2–6
2.5.1 The COBOL Command . . . 2–6
2.5.2 COBOL Command Qualifiers . . . 2–7
2.5.3 Compiling Programs with Conditional Compilation Lines . . . 2–16
2.5.4 Compiler Error Messages . . . 2–16
2.5.5 Compiler Listings . . . 2–17
2.5.5.1 Source Program Listing . . . 2–18
2.5.5.2 Storage Map Portion of Compiler Listing . . . 2–20
2.5.5.3 Compilation Summary . . . 2–24
2.5.5.4 Compiler Listing Including the /MACHINE_CODE Qualifier . . . 2–25
2.5.5.5 Compiler Listing for a Contained Program . . . 2–28

2.6 Linking a VAX COBOL Program . . . 2–29
2.6.1 The LINK Command . . . 2–30
2.6.2 LINK Command Qualifiers . . . 2–31
2.6.3 Positional Qualifiers . . . 2–33
2.6.4 Using an Object Module Library . . . 2–34

2.6.5 Object Libraries 2–35
2.6.5.1 Using System-Supplied Object Module Libraries 2–35
2.6.5.2 Defining the Search Order for Libraries 2–36
2.6.5.3 Default User Object Module Libraries 2–36
2.6.5.4 System Libraries 2–37
2.6.6 Shareable Images 2–37
2.6.6.1 Creating a Shareable Image 2–37
2.6.6.2 Using Transfer Vectors 2–40
2.6.7 Linker Error Messages 2–41

2.7 **Running a VAX COBOL Program** 2–42
2.7.1 COBOL Run-Time Errors 2–43

2.8 **Program Switches** 2–43
2.8.1 Setting Switches Internally 2–44
2.8.2 Setting Switches for a Process 2–44
2.8.3 Order of Evaluation 2–44
2.8.4 Checking and Controlling Switch Settings 2–44
2.8.5 Example Using Program Switches 2–45

Chapter 3 Using the VMS Debugger

3.1 **VMS Debugger Concepts** 3–1

3.2 **Features of the Debugger** 3–2

3.3 **Getting Started with the Debugger** 3–2
3.3.1 Compiling and Linking to Prepare for Debugging 3–3
3.3.1.1 Establishing the Debugging Configuration 3–3
3.3.2 Starting and Ending a Debugging Session 3–3
3.3.3 Issuing Debugger Commands 3–4

3.4 **Notes on VAX COBOL Support** 3–9

3.5 **Sample Debugging Session** 3–10

Part II Using VAX COBOL Features on VMS

Chapter 4 Numeric Data Handling

4.1 **How the Compiler Stores Numeric Data** 4–1

4.2 **Sign Conventions** 4–1

4.3 **Invalid Values in Numeric Items** 4–2

4.4 **Evaluating Numeric Items** 4–2
4.4.1 Numeric Relation Tests 4–2
4.4.2 Numeric Sign Tests 4–3
4.4.3 Numeric Class Tests 4–4
4.4.4 Success/Failure Tests 4–4

4.5 **Using the MOVE Statement** 4–5
4.5.1 Elementary Numeric Moves 4–5

4.5.2 Elementary Numeric Edited Moves 4–7
4.5.3 Common Move Errors 4–8

4.6 **Using the Arithmetic Statements** 4–8
4.6.1 Intermediate Results 4–9
4.6.2 Specifying a Truncation Qualifier 4–9
4.6.3 Using the ROUNDED Phrase 4–9
4.6.3.1 ROUNDED with REMAINDER 4–10
4.6.4 Using the SIZE ERROR Phrase 4–10
4.6.5 Using the GIVING Phrase 4–11
4.6.6 Multiple Operands in ADD and SUBTRACT Statements 4–11
4.6.7 Common Errors in Arithmetic Statements 4–11

4.7 **Arithmetic Expression Processing** 4–12

Chapter 5 Nonnumeric Data Handling

5.1 **Data Organization** 5–1
5.1.1 Group Items 5–2
5.1.2 Elementary Items 5–2

5.2 **Special Characters** 5–2

5.3 **Testing Nonnumeric Items** 5–3
5.3.1 Relation Tests of Nonnumeric Items 5–3
5.3.1.1 Classes of Data 5–3
5.3.1.2 Comparison Operations 5–4
5.3.2 Class Tests for Nonnumeric Items 5–4

5.4 **Data Movement** 5–5

5.5 **Using the MOVE Statement** 5–6
5.5.1 Group Moves 5–7
5.5.2 Elementary Moves 5–7
5.5.2.1 Edited Moves 5–8
5.5.2.2 Justified Moves 5–9
5.5.3 Multiple Receiving Items 5–9
5.5.4 Subscripted Moves 5–10
5.5.5 Common Nonnumeric Item MOVE Statement Errors 5–10
5.5.6 Using the MOVE CORRESPONDING Statement for Nonnumeric Items 5–10
5.5.7 Using Reference Modification 5–11

Chapter 6 Table Handling

6.1 **Introduction** 6–1

6.2 **Defining Tables** 6–1
6.2.1 Defining Fixed-Length, One-Dimensional Tables 6–1
6.2.2 Defining Fixed-Length, Multidimensional Tables 6–4
6.2.3 Defining Variable-Length Tables 6–6
6.2.4 Storage Allocation for Tables 6–7
6.2.4.1 Using the SYNCHRONIZED Clause 6–8

6.3 **Initializing Values of Table Elements** 6–11

6.4 Accessing Table Elements 6–14
6.4.1 Subscripting 6–14
6.4.2 Subscripting with Literals 6–14
6.4.3 Subscripting with Data Names 6–16
6.4.4 Subscripting with Indexes 6–16
6.4.5 Relative Indexing 6–17
6.4.6 Index Data Items 6–17
6.4.7 Assigning Index Values Using the SET Statement 6–17
6.4.7.1 Assigning an Integer Index Value with a SET Statement . . . 6–18
6.4.7.2 Incrementing an Index Value with the SET Statement 6–18
6.4.8 Identifying Table Elements Using the SEARCH Statement 6–18
6.4.8.1 Implementing a Sequential Search 6–18
6.4.8.2 Implementing a Binary Search 6–19

Chapter 7 Using the STRING, UNSTRING, and INSPECT Statements

7.1 Concatenating Data Using the STRING Statement 7–1
7.1.1 Multiple Sending Items 7–1
7.1.2 Using the DELIMITED BY Phrase 7–2
7.1.3 Using the POINTER Phrase 7–4
7.1.4 Using the OVERFLOW Phrase 7–4
7.1.5 Common STRING Statement Errors 7–5

7.2 Separating Data Using the UNSTRING Statement 7–6
7.2.1 Multiple Receiving Items 7–6
7.2.2 Controlling Moved Data Using the DELIMITED BY Phrase 7–8
7.2.2.1 Multiple Delimiters 7–12
7.2.3 Using the COUNT Phrase 7–12
7.2.4 Saving UNSTRING Delimiters Using the DELIMITER Phrase 7–13
7.2.5 Controlling UNSTRING Scanning Using the POINTER Phrase 7–14
7.2.6 Counting UNSTRING Receiving Items Using the TALLYING Phrase . . . 7–15
7.2.7 Exiting an UNSTRING Statement Using the OVERFLOW Phrase 7–16
7.2.8 Common UNSTRING Statement Errors 7–16

7.3 Examining and Replacing Characters Using the INSPECT Statement 7–17
7.3.1 Using the TALLYING and REPLACING Options of the INSPECT Statement 7–17
7.3.2 Restricting Data Inspection Using the BEFORE/AFTER Phrase 7–18
7.3.3 Implicit Redefinition 7–19
7.3.4 Examining the INSPECT Operation 7–21
7.3.4.1 Setting the Scanner 7–22
7.3.4.2 Active/Inactive Arguments 7–22
7.3.4.3 Finding an Argument Match 7–23
7.3.5 The TALLYING Phrase 7–24
7.3.5.1 The Tally Counter 7–24
7.3.5.2 The Tally Argument 7–24
7.3.5.3 The Tally Argument List 7–25
7.3.5.4 Interference in Tally Argument Lists 7–26
7.3.6 Using the REPLACING Phrase 7–29
7.3.6.1 The Search Argument 7–30
7.3.6.2 The Replacement Value 7–30
7.3.6.3 The Replacement Argument 7–31
7.3.6.4 The Replacement Argument List 7–31
7.3.6.5 Interference in Replacement Argument Lists 7–32
7.3.7 Using the CONVERTING Option 7–33
7.3.8 Common INSPECT Statement Errors 7–33

Chapter 8 The Basics of Handling VAX COBOL Files and Records

8.1 **VAX Record Management Services** 8–1

8.2 **File Attributes** 8–2

8.3 **Record Attributes** 8–2
8.3.1 Record Format 8–3
8.3.1.1 Fixed-Length Records 8–4
8.3.1.2 Variable-Length Records 8–5
8.3.2 Print-Controlled Files 8–7

8.4 **File Design Considerations** 8–7

8.5 **File Handling** 8–8
8.5.1 Identifying a File from Your VAX COBOL Program 8–8
8.5.1.1 Using the VALUE OF ID Clause for Device Independence 8–9
8.5.1.2 Using Logical Names 8–10
8.5.2 Choosing File Organization and Record Access Mode 8–11
8.5.2.1 File Organizations 8–11
8.5.2.2 Record Access Modes 8–12

8.6 **Opening and Closing Files** 8–14

8.7 **File Compatibility** 8–15
8.7.1 Data Type Differences 8–15
8.7.2 Data Record Formatting Differences 8–16

8.8 **Backing Up Your Files** 8–16

8.9 **Low-Volume I/O (ACCEPT and DISPLAY)** 8–16
8.9.1 Mnemonic Names (SPECIAL-NAMES Paragraph) 8–16
8.9.2 Logical Name Devices 8–17
8.9.3 ACCEPT Statement 8–18
8.9.4 DISPLAY Statement 8–18

8.10 **Printing with VAX VFP** 8–18

Chapter 9 Processing Sequential Files

9.1 **Sequential File Organization** 9–1

9.2 **Design Considerations** 9–2

9.3 **Statements for Sequential File Processing** 9–2

9.4 **Defining a Sequential File** 9–3

9.5 **Creating a Sequential File** 9–4

9.6 **Reading a Sequential File** 9–6

9.7 **Updating Records in a Sequential File** 9–7

9.8 **Extending a Sequential File** 9–8

Chapter 10 Processing Relative Files

10.1 Relative File Organization 10–1

10.2 Design Considerations 10–2

10.3 Statements for Relative File Processing 10–3

10.4 Defining a Relative File 10–4

10.5 Creating a Relative File 10–5
10.5.1 Sequential Access Mode Creation 10–5
10.5.2 Random Access Mode Creation 10–6

10.6 Reading a Relative File 10–7
10.6.1 Sequential Reading 10–7
10.6.2 Random Reading 10–8
10.6.3 Dynamic Reading 10–9

10.7 Updating a Relative File 10–10
10.7.1 Rewriting Relative Records 10–10
10.7.1.1 Sequential Access Mode Rewriting 10–11
10.7.1.2 Random Access Mode Rewriting 10–12
10.7.2 Deleting Relative Records 10–13
10.7.2.1 Sequential Access Mode Deletion 10–13
10.7.2.2 Random Access Mode Deletion 10–14

Chapter 11 Processing Indexed Files

11.1 Indexed File Organization 11–1

11.2 Design Considerations 11–2

11.3 Statements for Indexed File Processing 11–3

11.4 Defining an Indexed File 11–3

11.5 Creating and Populating an Indexed File 11–4

11.6 Reading an Indexed File 11–6
11.6.1 Sequential Reading 11–6
11.6.2 Random Reading 11–7
11.6.3 Dynamic Reading 11–9

11.7 Updating an Indexed File 11–10
11.7.1 Sequential Updating 11–10
11.7.2 Random Updating 11–12
11.7.3 Dynamic Updating 11–14

Chapter 12 Input/Output Exception Conditions Handling

12.1 Planning for the At End Condition 12–1

12.2 Planning for the Invalid Key Condition 12–2

12.3 Using File Status Values 12–3
12.3.1 VAX COBOL File Status Values 12–3
12.3.2 RMS File Status Values 12–5

12.4 Using Declarative Procedures to Handle Exception Conditions 12–7

Chapter 13 Sharing Files and Protecting Records

13.1 File-Sharing and Record-Locking Concepts 13–1

13.2 Ensuring Successful File Sharing 13–3
13.2.1 Providing Disk Residency 13–3
13.2.2 Using VMS File Protection 13–3
13.2.3 Determining the Intended Access Mode to a File 13–4
13.2.4 Indicating the Access Allowed to Other Streams 13–5
13.2.5 Describing Types of Access Streams 13–5
13.2.6 Summarizing Related File-Sharing Criteria 13–6
13.2.7 Checking File Operations 13–9
13.2.8 Specifying the OPEN EXTEND in a File-Sharing Environment 13–11
13.2.8.1 OPEN EXTEND with a Shared Sequential File 13–11
13.2.8.2 OPEN EXTEND with a Shared Relative File 13–11
13.2.8.3 OPEN EXTEND with a Shared Indexed File 13–12

13.3 Using Record Locking 13–12
13.3.1 Specifying Automatic Record Locking 13–13
13.3.2 Specifying Manual Record Locking 13–14
13.3.3 Locking Error Conditions 13–16
13.3.3.1 Hard Record Locks 13–16
13.3.3.2 Soft Record Locks 13–17
13.3.4 Releasing Locks on Deleted Records 13–17
13.3.5 Bypassing a Record Lock 13–18

Chapter 14 Using the COBOL SORT and MERGE Statements

14.1 ASCENDING and DESCENDING KEY Phrases 14–1
14.1.1 Sorting Concepts 14–2

14.2 USING and GIVING Phrases 14–2

14.3 INPUT PROCEDURE and OUTPUT PROCEDURE Phrases 14–3

14.4 WITH DUPLICATES IN ORDER Phrase 14–5

14.5 COLLATING SEQUENCE IS Alphabet-Name Phrase 14–6

14.6 File Organization 14–6

14.7 Multiple Sorts 14–6

14.8 Sorting Variable-Length Records 14–8

14.9 Preventing I/O Aborts 14–8

14.10 The MERGE Statement 14–9

14.11 Sample Programs Using the SORT and MERGE Statements 14–9

Chapter 15 Database Programming with VAX COBOL

15.1 The Self-Paced Demonstration Package 15–1

15.2 VAX COBOL Data Manipulation Language (DML) 15–2

15.3 Creating a VAX COBOL DML Program 15–4

15.4 Compiling a VAX COBOL DML Program 15–4
15.4.1 Copying Database Records in a VAX COBOL Program 15–5
15.4.2 Using the /MAP Compiler Qualifier 15–5

15.5 Linking a VAX COBOL DML Program 15–5

15.6 Running a VAX COBOL DML Program 15–5

15.7 A Database 15–6

15.8 Schema 15–6

15.9 Storage Schema 15–6

15.10 Subschema 15–6

15.11 Stream 15–7

15.12 Using CDD/Plus 15–8

15.13 Database Records 15–8

15.14 Database Data Item 15–8

15.15 Database Key 15–8

15.16 Record Types 15–9

15.17 Set Types 15–9

15.18 Sets 15–12
15.18.1 Simple Set Relationships 15–14
15.18.1.1 System-Owned Sets 15–15
15.18.1.2 Simple Sets 15–15
15.18.1.3 Forked Sets 15–16
15.18.2 Multiset Relationships 15–18
15.18.2.1 Many-to-Many Relationships Between Two Types of Records 15–18
15.18.2.2 Many-to-Many Relationships Between Records of the Same Type 15–20
15.18.2.3 One-to-Many Relationships Between Records of the Same Type 15–25

15.19 Areas 15–29

15.20 Realms 15–29

15.21 Run Unit 15–29

15.22 Currency Indicators 15–30
15.22.1 Current of Realm 15–30
15.22.2 Current of Set Type 15–31
15.22.3 Current of Record Type 15–32
15.22.4 Current of Run Unit 15–32

15.23 Currency Indicators in a VAX COBOL DML Program 15–33
15.23.1 Using the RETAINING Clause 15–34
15.23.2 Using Keeplists 15–36
15.23.3 Transactions and Quiet Points 15–38

15.24 VAX COBOL DML Programming—Tips and Techniques 15–39
15.24.1 The Ready Modes 15–39
15.24.1.1 Record Locking 15–40
15.24.2 COMMIT and ROLLBACK 15–41
15.24.3 The Owner and Member Test Condition 15–43
15.24.4 Using IF EMPTY Instead of IF OWNER 15–44
15.24.5 Modifying Members of Sorted Sets 15–44
15.24.6 CONNECT and DISCONNECT 15–46
15.24.7 RECONNECT 15–47
15.24.8 ERASE ALL 15–48
15.24.9 ERASE Record-Name 15–49
15.24.10 Freeing Currency Indicators 15–50
15.24.10.1 Establishing a Known Currency Condition 15–50
15.24.10.2 Releasing Record Locks 15–52
15.24.11 FIND and FETCH Statements 15–52
15.24.12 FIND ALL Option 15–53
15.24.13 FIND NEXT and FETCH NEXT Loops 15–53
15.24.14 Qualifying FIND and FETCH 15–55

15.25 Handling Database Exception Conditions 15–56
15.25.1 AT END Phrase 15–56
15.25.2 ON ERROR Phrase 15–56
15.25.3 USE Statement 15–56
15.25.4 How to Translate DB-CONDITION Values to Exception Messages 15–58

15.26 Debugging and Testing VAX COBOL DML Programs 15–58

15.27 DBQ Commands and DML Statements 15–59

15.28 Sample Debugging and Testing Session 15–60

15.29 Reading a VAX COBOL Subschema Map Listing 15–68
15.29.1 PARTSS1 Subschema Map Listing 15–69
15.29.2 PARTSS3 Subschema Map Listing 15–73

15.30 Examples 15–75
15.30.1 Populating a Database 15–75
15.30.2 Backing Up a Database 15–83
15.30.3 Accessing and Displaying Database Information 15–89
15.30.4 PARTBOM Sample Run 15–91
15.30.5 Creating Relationships Between Records of the Same Type 15–92

15.30.6 STOOL Program Parts Breakdown Report—Sample Run 15–96
15.30.7 Creating New Record Relationships 15–96
15.30.7.1 PERSONNEL-UPDATE Sample Run—Listing Before Promotion 15–102
15.30.7.2 PERSONNEL-UPDATE Sample Run—Listing After Promotion 15–102

Chapter 16 Producing Printed Reports with VAX COBOL

16.1 Designing the Report 16–1

16.2 Components of a Report 16–2

16.3 Accumulating and Reporting Totals 16–5

16.4 The Logical Page and the Physical Page 16–7

16.5 Programming the Conventional VAX COBOL Report 16–7
16.5.1 Defining the Logical Page in a Conventional Report............. 16–7
16.5.2 Controlling the Spacing in a Conventional Report 16–8
16.5.3 Advancing to the Next Logical Page in a Conventional Report....... 16–8
16.5.3.1 Programming for the Page-Overflow Condition in a Conventional Report.......................... 16–9
16.5.3.2 Using a Line Counter 16–10
16.5.4 Printing the Conventional Report 16–10
16.5.5 A Conventional File Report Example 16–10

16.6 Programming the Linage-File VAX COBOL Report 16–14
16.6.1 Defining the Logical Page in Linage-File Report 16–14
16.6.2 Controlling the Spacing in a Linage-File Report................ 16–16
16.6.3 Using the LINAGE-COUNTER 16–16
16.6.4 Advancing to the Next Logical Page in a Linage-File Report 16–16
16.6.5 Programming for the Page-Overflow Condition 16–16
16.6.6 Printing a Linage-File Report 16–20
16.6.7 A Linage-File Report Example 16–21

16.7 Modes for Printing Reports 16–23
16.7.1 Directly Allocating a Printer 16–23
16.7.2 Spooling to a Mass Storage Device 16–24

16.8 Programming the Report Writer Report 16–24
16.8.1 Using the REPORT Clause in the File Section 16–25
16.8.2 Defining the Report Section and the Report File 16–25
16.8.3 Defining a Report Writer Logical Page with the PAGE Clause 16–25
16.8.4 Describing Report Group Description Entries 16–26
16.8.5 Vertical Spacing for the Logical Page 16–29
16.8.6 Horizontal Spacing for the Logical Page 16–30
16.8.7 Assigning a Value in a Print Line 16–31
16.8.8 Defining the Source for a Print Field 16–31
16.8.9 Specifying Multiple Reports 16–32
16.8.10 Generating and Controlling Report Headings and Footings 16–33
16.8.11 Defining and Incrementing Totals 16–34
16.8.11.1 Subtotaling 16–34
16.8.11.2 Crossfooting 16–35
16.8.11.3 Rolling-Forward 16–35
16.8.11.4 RESET Option................................ 16–36
16.8.11.5 UPON Option 16–36

16.8.12 Restricting Print Items 16–37
16.8.13 Processing a Report Writer Report 16–38
16.8.13.1 Initiating the Report 16–38
16.8.13.2 Generating a Report Writer Report 16–39
16.8.13.3 Automatic Operations of the GENERATE Statement 16–39
16.8.13.4 Ending Report Writer Processing 16–40
16.8.13.5 Applying the USE BEFORE REPORTING Statement 16–41
16.8.13.6 Suppressing a Report Group 16–42
16.8.14 Selecting a Report Writer Report Type 16–42
16.8.14.1 Detail Reporting 16–42
16.8.14.2 Summary Reporting 16–43

16.9 Report Writer Examples 16–43
16.9.1 Input Data 16–43
16.9.2 REPORT1—Detail Report Program 16–44
16.9.3 REPORT2—Detail Report Program 16–49
16.9.4 REPORT3—Detail Report Program 16–57
16.9.5 REPORT4—Detail Report Program 16–63
16.9.6 REPORT5—Summary Report Program 16–71

16.10 Solving Report Problems 16–80
16.10.1 Printing More Than One Logical Line on a Single Physical Line 16–80
16.10.2 Group Indicating 16–85
16.10.3 Fitting Reports on the Page 16–86
16.10.4 Printing Totals Before Detail Lines 16–86
16.10.5 Underlining Items in Your Reports 16–87
16.10.6 Bolding Items in Your Reports 16–87

Chapter 17 Forms for Video Terminals

17.1 Clearing a Screen Area 17–3

17.2 Horizontal and Vertical Positioning of the Cursor 17–5

17.3 Assigning Character Attributes to Your Format Entries 17–8

17.4 Using the CONVERSION Clause to Display Data 17–10

17.5 Handling Data with ACCEPT Options 17–13
17.5.1 Using CONVERSION with ACCEPT Data 17–13
17.5.2 Using ON EXCEPTION When Accepting Data with CONVERSION . . . 17–14
17.5.3 Protecting Your Screen 17–15
17.5.4 Using NO ECHO with ACCEPT Data 17–17
17.5.5 Assigning Default Values to Data Fields 17–18

17.6 Using Keys on Your Terminal to Define Special Program Functions 17–21

17.7 Using the EDITING Phrase 17–31

Chapter 18 Interprogram Communication

18.1 Multiple COBOL Program Run-Unit Concepts 18–1
18.1.1 Definition of a Multiple COBOL Program Run Unit 18–1
18.1.2 Examples of COBOL Run Units 18–1
18.1.3 Calling Procedures 18–3

18.2 COBOL Program Attributes . . . 18–4
18.2.1 The INITIAL Clause . . . 18–4
18.2.2 The EXTERNAL Clause . . . 18–4

18.3 Transferring Execution Control . . . 18–5
18.3.1 The CALL Statement . . . 18–5
18.3.2 The EXIT PROGRAM Statement . . . 18–5
18.3.3 Nesting CALL Statements . . . 18–6

18.4 Accessing Another Program's Data Division . . . 18–8
18.4.1 The USING Phrase . . . 18–8
18.4.1.1 The Linkage Section . . . 18–10

18.5 Communicating with Contained COBOL Programs . . . 18–12
18.5.1 The COMMON Clause . . . 18–13
18.5.2 Defining and Using the GLOBAL Clause . . . 18–14
18.5.2.1 Sharing Data . . . 18–14
18.5.2.2 Sharing Files . . . 18–15
18.5.2.3 Sharing Database Resources . . . 18–15
18.5.2.4 Sharing Other Resources . . . 18–15
18.5.3 Sharing USE Procedures . . . 18–16

18.6 Including Non-COBOL Programs in the Run Unit . . . 18–19

18.7 Using VAX COBOL in the Common Language Environment . . . 18–23

18.8 The VAX Procedure Calling and Condition Handling Standard . . . 18–24
18.8.1 Register and Stack Usage . . . 18–24
18.8.2 Return of the Function Value . . . 18–25
18.8.3 The Argument List . . . 18–25

18.9 VMS Run-Time Library Routines . . . 18–25

18.10 VMS System Services Routines . . . 18–26

18.11 Calling Routines . . . 18–27
18.11.1 Determining the Type of Call . . . 18–27
18.11.2 Defining the Argument . . . 18–28
18.11.3 Calling the External Routine . . . 18–29
18.11.4 Calling System Routines . . . 18–29
18.11.4.1 System Routine Arguments . . . 18–29
18.11.4.2 Calling a System Routine in a Function Call . . . 18–33
18.11.4.3 Calling a System Routine in a Procedure Call . . . 18–34
18.11.5 Checking the Condition Value . . . 18–34
18.11.5.1 Library Return Status and Condition Value Symbols . . . 18–36
18.11.6 Locating the Result . . . 18–36

18.12 Calling Shareable Images . . . 18–36

18.13 Examples . . . 18–37

18.14 Additional Information . . . 18–40

Part III VAX COBOL Programming Options and Performance Considerations

Chapter 19 Using the REFORMAT Utility

19.1 ANSI-to-Terminal Format Conversion 19–1
19.1.1 ANSI-to-Terminal REFORMAT Command String 19–2

19.2 Terminal-to-ANSI Format Conversion 19–2
19.2.1 Terminal-to-ANSI REFORMAT Command String 19–3

19.3 REFORMAT Error Messages 19–4

Chapter 20 Optimizing Your VAX COBOL Program

20.1 Numeric Data Representation 20–1
20.1.1 Scaling and Mixing Data-Types 20–2
20.1.2 Using Significant Digits 20–2

20.2 Choices in Procedure Division Statements 20–2
20.2.1 Using ADD, SUBTRACT, MULTIPLY, and DIVIDE Instead of COMPUTE 20–2
20.2.2 Using GO TO DEPENDING ON Instead of IF, GO TO 20–3
20.2.3 Using Indexing Instead of Subscripting 20–3
20.2.4 Using PERFORM n TIMES Instead of PERFORM VARYING 20–4
20.2.5 Using SEARCH ALL Instead of SEARCH 20–4

20.3 Using VAX COBOL for I/O Operations 20–5
20.3.1 Using the APPLY Clause 20–6
20.3.1.1 Using the PREALLOCATION Phrase of the APPLY Clause 20–6
20.3.1.2 Using the EXTENSION Phrase of the APPLY Clause 20–7
20.3.1.3 Using the DEFERRED-WRITE Phrase of the APPLY Clause 20–7
20.3.1.4 Using the FILL-SIZE ON Phrase of the APPLY Clause 20–7
20.3.1.5 Using the WINDOW Phrase of the APPLY Clause 20–7
20.3.2 Using Multiple Buffers 20–7
20.3.3 Sharing Record Areas 20–8

20.4 Optimizing File Design 20–10
20.4.1 Sequential Files 20–10
20.4.2 Relative Files 20–10
20.4.2.1 Maximum Record Number (MRN) 20–11
20.4.2.2 Cell Size 20–11
20.4.2.3 Bucket Size 20–11
20.4.2.4 File Size 20–13
20.4.3 Indexed Files 20–14
20.4.3.1 Optimizing Indexed File I/O 20–15
20.4.3.2 Calculating Key Index Levels 20–19
20.4.3.3 Caching Index Roots 20–19

Appendix A Compiler Implementation Limitations

Appendix B Error Messages

B.1 Run-Time Errors B–1
B.1.1 Sample Run-Time Error B–1

B.2 Program Run Errors B–5
B.2.1 Faulty Data B–5
B.2.2 Program Logic Errors B–6

B.3 Run-Time Input/Output Errors B–7

B.4 Compiler Messages B–12

Appendix C Using the COBOL–81 SUBSET Flagger

C.1 Using VAX COBOL to Produce Compatible COBOL–81 Source Programs C–1

C.2 Using the /STANDARD=PDP11 Qualifier C–1

C.3 VAX COBOL Flagging Procedures C–2

C.4 Source Level Differences and Incompatibilities C–2
C.4.1 General Language Concepts C–2
C.4.2 Unsupported Language Elements by Division C–3

C.5 Alignment of COMP Data Items C–9

Appendix D Additional Information on COBOL Command Qualifiers

D.1 Using the /FLAGGER Qualifier D–1
D.1.1 What Is the /FLAGGER Qualifier D–1
D.1.2 /FLAGGER Options D–2
D.1.3 FIPS Levels D–2

D.2 /INSTRUCTION_SET Qualifier D–3
D.2.1 Overview of VAX Architectural Subsetting D–4
D.2.2 How Subsetting May Affect VAX COBOL Programs D–4
D.2.3 Determining the Instruction Set D–4
D.2.4 Selecting an Option D–4

D.3 Differences Using /STANDARD=85 and /STANDARD=V3 . . . . D–6
D.3.1 Overview D–6
D.3.2 DIVIDE Statement D–6
D.3.3 STRING Statement D–7
D.3.4 UNSTRING Statement D–7
D.3.5 INSPECT Statement D–8
D.3.6 PERFORM ... VARYING ... AFTER Statement D–8
D.3.7 PIC P Digits D–9
D.3.8 Size of Variable Length Tables D–10
D.3.9 EXIT PROGRAM Statement D–10

D.3.11 OPEN I-O and EXTEND Modes D–12

Appendix E Optional Programming Productivity Tools

E.1 VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA) E–1
E.1.1 Preparing an SCA Library E–2
E.1.2 Starting and Terminating an LSE or an SCA Session E–3
E.1.3 Compiling from Within LSE E–3
E.1.4 Notes on VAX COBOL Support E–4
E.1.4.1 Programming Language Placeholders and Tokens E–4
E.1.4.2 Placeholder and Design Comment Processing E–5
E.1.5 LSE and SCA Examples E–8
E.1.5.1 Data Definition E–9
E.1.5.2 IF Statement E–10

E.2 VAX CDD/Plus E–12
E.2.1 Overview of Data Dictionaries E–13
E.2.2 CDD/Plus Features E–13
E.2.3 CDD/Plus Concepts E–13
E.2.3.1 CDD/Plus Dictionary Formats E–13
E.2.3.2 Dictionary Path Names E–14
E.2.3.3 Dependency Recording E–14
E.2.3.4 Compiled Module Entities E–15
E.2.3.5 Entities E–15
E.2.3.6 Relationships E–16
E.2.3.7 Pieces Tracking E–16
E.2.3.8 Distributed Dictionary Access E–16
E.2.3.9 Data Security and Integrity E–17
E.2.3.10 CDD/Plus Call Interface E–17
E.2.4 Creating Data Definitions E–17
E.2.5 Accessing Data Definitions E–18
E.2.5.1 Using the COPY FROM DICTIONARY Statement E–18
E.2.5.2 Using the RECORD DEPENDENCY Statement E–18
E.2.6 Using the /DEPENDENCY_DATA Qualifier E–19
E.2.7 Viewing CDD/Plus Relationships E–20
E.2.8 VAX COBOL Support for CDD/Plus Data Types E–21

E.3 VAX COBOL GENERATOR E–22
E.3.1 VAX COBOL GENERATOR Features E–23

E.4 VAX Data Base Management System (VAX DBMS) E–24

E.5 VAX DEC/Test Manager E–24

E.6 VAX DEC/Code Management System (CMS) E–25

Index

Examples

2–1 Main Program and Subprograms 2–38
2–2 Command Procedure to Link a Program as a Shareable Image 2–39
2–3 Transfer Vectors 2–41
2–4 Using Program Switches 2–45
3–1 Source Code Used in the Sample Debug Session 3–10
4–1 Success/Failure Test 4–5
5–1 Item Concatenation Using Two MOVE Statements 5–6
5–2 Sample Record Description Using the MOVE CORRESPONDING Statement 5–11
6–1 One-Dimensional Table 6–2
6–2 Multiple Data Items in a One-Dimensional Table 6–2
6–3 Defining a Table with an Index and an Ascending Search Key 6–3
6–4 Defining a Two-Dimensional Table 6–4
6–5 Defining a Three-Dimensional Table 6–5
6–6 Defining a Variable-Length Table 6–7
6–7 Sample Record Description Defining a Table 6–8
6–8 Record Description Containing a COMP SYNC Item 6–9
6–9 Adding an Item Without Changing the Table Size 6–9
6–10 How Adding 3 Bytes Adds 4 Bytes to the Element Length 6–10
6–11 Initializing Tables with the VALUE Clause 6–11
6–12 Initializing a Table with the OCCURS Clause 6–12
6–13 Initializing Mixed Usage Items 6–13
6–14 Initializing Alphanumeric Items 6–13
6–15 Using a Literal Subscript to Access a Table 6–15
6–16 Subscripting a Multidimensional Table 6–15
6–17 Subscripting with Index Name Items 6–17
6–18 Sample Table 6–21
6–19 A Serial Search 6–22
6–20 Using SEARCH and Varying an Index Other Than the First Index 6–22
6–21 Using SEARCH and Varying an Index Data Item 6–23
6–22 Using SEARCH and Varying an Index Not Associated with the Target Table 6–23
6–23 Doing a Serial Search Without Using the VARYING Phrase 6–24
6–24 A Multiple-Key Binary Search 6–24
7–1 Using the STRING Statement and Literals 7–2
7–2 Sample Overflow Condition 7–5
8–1 Sample Record Description 8–3
8–2 Determining Fixed-Length Record Size 8–4
8–3 Determining Fixed-Length Record Size for Files with Multiple Record Descriptions 8–4
8–4 Creating Variable-Length Records with the DEPENDING ON Phrase 8–5
8–5 Creating Variable-Length Records with the RECORD VARYING Phrase 8–6
8–6 Creating Variable-Length Records and Using the OCCURS Clause with the DEPENDING ON Phrase 8–6
8–7 Defining Fixed-Length Records with Multiple Record Descriptions 8–7
8–8 Defining a Disk File 8–8
8–9 Defining a Magnetic Tape File 8–9
8–10 How to Override or Supplement a File Specification at Run Time 8–10

8–11 Sequential File SELECT Statements 8–13
8–12 Relative File SELECT Statements 8–13
8–13 Indexed File SELECT Statements 8–14
9–1 Defining a Sequential File 9–4
9–2 Creating a Sequential File 9–5
9–3 Reading a Sequential File 9–7
9–4 Rewriting a Sequential File 9–8
9–5 Extending a Sequential File 9–9
10–1 Defining a Relative File 10–4
10–2 Creating a Relative File in Sequential Access Mode 10–5
10–3 Creating a Relative File in Random Access Mode 10–6
10–4 Reading a Relative File Sequentially 10–8
10–5 Reading a Relative File Randomly 10–9
10–6 Reading a Relative File Dynamically 10–10
10–7 Rewriting Relative Records in Sequential Access Mode 10–11
10–8 Rewriting Relative Records in Random Access Mode 10–13
10–9 Deleting Relative Records in Sequential Access Mode 10–14
10–10 Deleting Relative Records in Random Access Mode 10–15
11–1 Defining an Indexed File 11–4
11–2 Creating and Populating an Indexed File 11–5
11–3 Reading an Indexed File Sequentially 11–7
11–4 Reading an Indexed File Randomly 11–8
11–5 Reading an Indexed File Dynamically 11–9
11–6 Updating an Indexed File Sequentially 11–11
11–7 Updating an Indexed File Randomly 11–12
12–1 Handling the At End Condition 12–2
12–2 Handling the Invalid Key Condition 12–3
12–3 Defining a File Status for a File 12–4
12–4 Using the File Status Value in an Exception Handling Routine 12–4
12–5 Referencing RMS-STS, RMS-STV, RMS-CURRENT-STS,
and RMS-CURRENT-STV Values 12–6
12–6 The Declarative Skeleton 12–8
12–7 A Declarative Procedure Skeleton 12–8
12–8 Five Types of Declarative Procedures 12–9
13–1 Two Access Streams to a Single File 13–6
13–2 Program Segment for RMS-STS File-Sharing Exceptions 13–10
13–3 Automatic Record Locking 13–13
13–4 Sample Program Using Manual Record Locking 13–15
14–1 INPUT and OUTPUT PROCEDURE Phrases 14–3
14–2 USING Phrase Replaces INPUT PROCEDURE Phrase 14–5
14–3 Overriding the COLLATING SEQUENCE IS Phrase 14–6
14–4 Using Two Sort Files 14–7
14–5 Using the AFTER STANDARD ERROR PROCEDURE 14–8
14–6 Using the MERGE Statement 14–9
14–7 Sorting a File with the USING and GIVING Phrases 14–10
14–8 Using the USING and OUTPUT PROCEDURE Phrases 14–11
14–9 Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases 14–13
14–10 Using the COLLATING SEQUENCE IS Phrase 14–15
14–11 Creating a New Sort Key 14–16

14–12 Merging Files 14–18
15–1 Printing a Listing of a Simple Set 15–16
15–2 Using Forked Sets 15–17
15–3 Currency Indicators 15–33
15–4 Using the RETAINING Clause 15–35
15–5 ROLLBACK Statement 15–43
15–6 Owner and Member Test Condition 15–44
15–7 RECONNECT Statement 15–48
15–8 FREE CURRENT Statement 15–51
15–9 FETCH NEXT Loop 15–54
15–10 Using a FETCH NEXT Loop to Walk Through a Set Type 15–54
15–11 A Single USE Statement 15–57
15–12 Multiple USE Statements 15–58
15–13 Sample VAX COBOL DML Program Statements 15–60
15–14 Sample DML Program Statements 15–65
15–15 Populating a Database 15–76
15–16 Backing Up a Database 15–83
15–17 Accessing and Displaying Database Information 15–89
15–18 Sample Run of the PARTBOM Program 15–91
15–19 Creating Relationships Between Records of the Same Type 15–92
15–20 Creating New Record Relationships 15–96
15–21 Sample Run of PERSONNEL-UPDATE Before Promotion 15–102
15–22 Sample Run of PERSONNEL-UPDATE After Promotion 15–103
16–1 Checking for the Page-Overflow Condition 16–9
16–2 Page Advancing and Line Skipping 16–12
16–3 Checking for Page-Overflow on a 28-Line Logical Page 16–18
16–4 Programming a 20-Line Logical Page Defined by the LINAGE Clause 16–22
16–5 Sample Program 1 16–45
16–6 Sample Program 2 16–49
16–7 Sample Program 3 16–57
16–8 Sample Program 4 16–64
16–9 Sample Program 5 16–71
16–10 Printing Labels Four-Up 16–81
16–11 Printing Labels Four-Up in Sort Order 16–84
17–1 Erasing a Screen 17–3
17–2 Cursor Positioning 17–6
17–3 Using PLUS for Cursor Positioning 17–7
17–4 Using Character Attributes 17–9
17–5 Using the CONVERSION Clause 17–11
17–6 Using the ON EXCEPTION Clause 17–14
17–7 Using the SIZE Phrase 17–16
17–8 Using NO ECHO 17–18
17–9 Using the DEFAULT Phrase 17–19
17–10 Using the CONTROL KEY IN Phrase 17–28
17–11 EDITING Phrase Sample Code 17–32
18–1 Execution Sequence 18–7
18–2 Sequence of Messages Displayed 18–8
18–3 Calling a BASIC Program from VAX COBOL 18–20
18–4 BASIC Program APP and Output Data 18–21

18–5 Calling a FORTRAN Program from VAX COBOL 18–22
18–6 FORTRAN Subroutine SQROOT 18–23
18–7 Random Number Generator 18–37
18–8 Sample Run of RUNTIME 18–37
18–9 Using SYS$SETDIR 18–38
18–10 Using $ASCTIM 18–38
18–11 Sample Run of CALLTIME 18–38
B–1 Using RMS Special Registers to Detect Errors B–7
B–2 Using RMS-CURRENT Special Registers to Detect Errors B–10
D–1 INSPECT Statement Using Format 3 . . . . . . D–8
D–2 INSPECT Statement Using Formats 1 and 2 . . . . . . D–8

Figures

2–1 DCL Commands for Developing Programs 2–3
2–2 VAX COBOL Source Program Listing 2–19
2–3 Storage Map Portion of VAX COBOL Compiler Listing 2–23
2–4 Compilation Summary of a VAX COBOL Source Program Listing 2–25
2–5 VAX COBOL Listing Specifying /MACHINE_CODE Qualifier 2–27
2–6 VAX COBOL Listing of Contained Program 2–29
6–1 Organization of the One-Dimensional Table in Example 6–1 6–2
6–2 Organization of Multiple Data Items in a One-Dimensional Table 6–3
6–3 Organization of a Table with an Index and an Ascending Search Key 6–4
6–4 Organization of a Two-Dimensional Table 6–5
6–5 Organization of a Three-Dimensional Table 6–6
6–6 Memory Map for Example 6–7 6–8
6–7 Memory Map for Example 6–8 6–9
6–8 Memory Map for Example 6–9 6–10
6–9 Memory Map for Example 6–10 6–11
6–10 Memory Map for Example 6–11 6–12
6–11 Memory Map for Example 6–13 6–13
6–12 Memory Map for Example 6–14 6–14
7–1 Results of the STRING Operation 7–2
7–2 Matching Delimiter Characters to Characters in a Field 7–19
7–3 Sample INSPECT Statement 7–21
7–4 Typical REPLACING Phrase 7–22
7–5 The Replacement Argument 7–31
9–1 Sequential File Organization 9–1
9–2 A Multiple-Volume Sequential File 9–2
10–1 Relative File Organization 10–2
11–1 Indexed File Organization 11–2
13–1 Multiple Access to a File 13–1
13–2 Relationship of Record Locking to File Sharing 13–2
13–3 File-Sharing Options 13–8
13–4 Why a Record-Already-Exists Error Occurs 13–12
13–5 Valid and Invalid Combinations for Manual Record Locking 13–15
15–1 Database and Application Program Relationship 15–6
15–2 Bachman Diagram 15–9
15–3 Partial Bachman Diagram of the PARTSS1 Subschema 15–10

15–4 Bachman Diagram of the PARTSS3 Subschema 15–10
15–5 Sample Occurrence Diagram 1 15–11
15–6 Sample Occurrence Diagram 2 15–12
15–7 One Occurrence of Set PART_SUPPLY 15–13
15–8 Set Relationship 15–13
15–9 Set Relationships 15–14
15–10 Occurrence Diagram of a Relationship Between Two Set Types 15–14
15–11 Bachman Diagram of a System-Owned Set Type 15–15
15–12 Bachman Diagram of a Simple Set Type 15–15
15–13 Bachman Diagram of a Forked Set Type 15–16
15–14 Forked Set Occurrence 15–18
15–15 Bachman Diagram of a Many-to-Many Relationship Between Two Types of Records 15–19
15–16 Many-to-Many Relationship Between Two Types of Records 15–20
15–17 Bachman Diagram of a Many-to-Many Relationship Between Records of the Same Type 15–21
15–18 Current of PART_USES and PART_USED_ON 15–21
15–19 Retain PART_USES Currency 15–22
15–20 COMPONENT Is Connected to Both Set Types 15–23
15–21 Finding the Stool Legs While Keeping STOOL Current of PART_USES 15–23
15–22 Completed Bill of Materials 15–24
15–23 Occurrence Diagram of a Many-to-Many Relationship Between Records of the Same Type 15–24
15–24 Sample Parts Breakdown Report 15–25
15–25 One-to-Many Relationship Between Records of the Same Type 15–26
15–26 Sample Data Prior to Update 15–27
15–27 Sample Data After Update 15–28
15–28 Database Relationships 15–29
15–29 Currency Status by Executable DML Statement 15–34
15–30 Physical Representation of a Realm Without a RETAINING Clause 15–35
15–31 Physical Representation of a Realm with a RETAINING Clause 15–36
15–32 State of KEEPLIST-1 After Executing Line 000160 15–37
15–33 State of KEEPLIST-1 After Executing Lines 000190 and 000200 15–37
15–34 Transactions and Quiet Points 15–39
15–35 Using the COMMIT Statement 15–42
15–36 Occurrences of the RESPONSIBLE_FOR Set Type 15–43
15–37 Modifying Members of Sorted Sets 15–45
15–38 After Modifying MEMBER_B and Using RETAINING 15–46
15–39 Occurrence Diagram Prior to RECONNECT 15–47
15–40 Occurrence Diagram After RECONNECT 15–48
15–41 Results of an ERASE ALL 15–49
15–42 Results of an ERASE Record-Name (with Both OPTIONAL and FIXED Retention Classes) 15–50
15–43 Record Locking 15–52
15–44 Using FIND NEXT and FETCH NEXT Loops 15–53
15–45 Split Screen After FETCH FIRST PART USING PART_ID 15–62
15–46 Split Screen After FETCH NEXT WITHIN PART_SUPPLY 15–63
15–47 Split Screen After MODIFY SUP_RATING 15–64
15–48 Split Screen After FETCH OWNER WITHIN VENDOR_SUPPLY 15–64
15–49 Split Screen After FETCH CURRENT PART RETAINING PART_SUPPLY 15–66

15–50 Split Screen After FETCH NEXT WITHIN PART_SUPPLY 15–67
15–51 Split Screen After FETCH 2 WITHIN PART_SUPPLY 15–68
15–52 PARTSS1-PROGRAM Compiler Listing 15–70
15–53 PARTSS3-PROGRAM Compiler Listing 15–74
16–1 Sample Layout Worksheet 16–2
16–2 Components of a Report 16–4
16–3 Subtotals, Crossfoot Totals, and Rolled Forward Totals 16–6
16–4 Logical Page Area for a Conventional Report 16–8
16–5 A 20-Line Logical Page 16–11
16–6 A Double-Spaced Master Listing 16–12
16–7 Logical Page Areas for a Linage File Report 16–15
16–8 A 28-Line Logical Page 16–17
16–9 A 20-Line Logical Page 16–21
16–10 Presentation Order for a Logical Page 16–27
16–11 Sample Report Using All Seven Report Groups 16–28
16–12 First GENERATE Statement 16–39
16–13 Subsequent GENERATE Statements 16–40
16–14 TERMINATE Statement 16–41
16–15 CUSTMAST1.LIS 16–48
16–16 CUSTMAST2.LIS 16–55
16–17 CUSTMAST3.LIS 16–63
16–18 CUSTMAST4.LIS 16–70
16–19 CUSTMAST5.LIS 16–79
16–20 Printing Labels Four-Up 16–81
16–21 Printing Labels Four-Up in Sort Order 16–83
17–1 Adding Information to a Master File with a Video Form 17–2
17–2 Screen Before the ERASE Statement Executes 17–4
17–3 Screen After the ERASE Statement Executes 17–5
17–4 Positioning the Data on Line 19, Column 5 17–6
17–5 Cursor Positioning Using the PLUS Option 17–8
17–6 Screen Display with Character Attributes 17–10
17–7 Sample Run of Program CONVERT 17–13
17–8 Accepting Data with the ON EXCEPTION Option 17–15
17–9 Screen Display of NUM-DATA Using the PROTECTED Option 17–17
17–10 Accepting Data with the NO ECHO Option 17–18
17–11 Accepting Data with the DEFAULT Phrase 17–21
17–12 VAX COBOL Control Keys on the Standard VT52 Keypad 17–25
17–13 VAX COBOL Control Keys on the Standard VT100 Keypad 17–26
17–14 VAX COBOL Control Keys on the Standard VT200 and VT300 Keypad 17–27
17–15 Screen Display of Program SPECIAL 17–31
17–16 Form with ACCEPT WITH EDITING Phrase 17–34
18–1 Run Unit with Three Separately Compiled Programs 18–2
18–2 Run Unit with a Main Program and Two Contained Programs 18–2
18–3 Run Unit with Three Separately Compiled Programs and Two Contained Programs 18–3
18–4 Sharing Execution Control from a Main Program to Multiple Subprograms 18–6
18–5 CALL Statement Nesting 18–6
18–6 Accessing Another Program's Data Division 18–10
18–7 Defining Data Names in Linkage Section 18–12

18–8 Using the COMMON Clause . . . 18–14
18–9 Sharing USE Procedures . . . 18–17
18–10 Executing Declaratives with Contained Programs (Rule 1) . . . 18–18
18–11 Executing Declaratives Within Contained Programs (Rule 2) . . . 18–19
20–1 Execution of a SEARCH ALL Statement . . . 20–5
20–2 Sharing Record Areas . . . 20–9
20–3 Two-Level Primary Index . . . 20–14
B–1 Listing of Program CALL1 . . . B–2
B–2 Listing of Program CALL2 . . . B–4

Tables

2–1 VMS File Maintenance Commands . . . 2–4
2–2 COBOL Command Qualifiers and Defaults . . . 2–7
2–3 Common LINK Qualifiers and Defaults . . . 2–31
2–4 LINK Positional Qualifiers . . . 2–33
3–1 Debugger Command Summary . . . 3–5
4–1 Numeric Relational Operator Descriptions . . . 4–3
4–2 Sign Tests . . . 4–4
4–3 Numeric Editing . . . 4–8
4–4 ROUNDING . . . 4–9
5–1 Relational Operator Descriptions . . . 5–3
5–2 Nonnumeric Elementary Moves . . . 5–7
5–3 Data Movement with Editing Symbols . . . 5–8
5–4 Data Movement with the JUSTIFIED Clause . . . 5–9
6–1 Subscripting Rules for a Multidimensional Table . . . 6–16
6–2 Subscripting with Data Names . . . 6–16
7–1 Results of Sample Overflow Statements . . . 7–5
7–2 Values Moved into the Receiving Items Based on the Sending Item Value . . . 7–7
7–3 Handling a Short Sending Item . . . 7–8
7–4 Results of Delimiting with an Asterisk . . . 7–8
7–5 Results of Delimiting Multiple Receiving Items . . . 7–9
7–6 Results of Delimiting with Two Asterisks . . . 7–10
7–7 Results of Delimiting with ALL Asterisks . . . 7–11
7–8 Results of Delimiting with ALL Double Asterisks . . . 7–11
7–9 Results of Multiple Delimiters . . . 7–12
7–10 Values Resulting from Implicit Redefinition . . . 7–20
7–11 Relationship Among INSPECT Argument, Delimiter, Item Value, and Argument Active Position . . . 7–23
7–12 LEADING Delimiter of the Inspection Operation . . . 7–25
7–13 Results of the Scan with Separate Tallies . . . 7–26
8–1 VAX COBOL File Organizations—Advantages and Disadvantages . . . 8–12
9–1 Valid I/O Statements for Sequential Files . . . 9–3
10–1 Valid I/O Statements for Relative Files . . . 10–3
11–1 Valid I/O Statements for Indexed Files . . . 11–3
13–1 File-Sharing Environment Codes . . . 13–9
16–1 Results of Group Indicating . . . 16–85
17–1 Cursor Positioning Requirements for ERASE Options . . . 17–3
17–2 Available Character Attributes by Terminal Type . . . 17–9

17–3 VAX COBOL Characters Returned for Cursor Positioning, Program Function, Function, and Auxiliary Keypad Keys 17–22
17–4 Key Functions for the EDITING Phrase 17–31
18–1 VAX Register Usage 18–24
18–2 Run-Time Library Facilities 18–25
18–3 System Services 18–26
18–4 VAX COBOL Implementation 18–30
B–1 Information Contained in an Error Message B–12
D–1 Relationship Among VAX COBOL Modules, Subsets, and Levels D–3
D–2 PERFORM ... VARYING ... AFTER Identifier Values D–9
D–3 Table Values After a MOVE Statement D–10
D–4 New and Revised I-O Status Codes D–11
E–1 CDD/Plus Data Types E–21

Preface

Objectives

This book and its companion volume, the *VAX COBOL Reference Manual*, describe the VAX COBOL language and its programming system. This manual describes how to use VAX COBOL under the VMS operating system. The *VAX COBOL Reference Manual* describes the concepts and rules of the VAX COBOL language.

Intended Audience

This documentation set is designed for the experienced COBOL programmer. It does not attempt to teach the COBOL language, fundamental programming, or system concepts. Textbooks and Digital courses are available for those purposes.

Associated Documents

If you are unfamiliar with the VMS operating system, refer to the VMS documentation for the following information:

- Basic information on the VMS operating system
- Detailed information about how to use the Digital Command Language (DCL)
- A summary description and glossary that provides an overview of the VMS system

The VAX documentation on VAX architecture provides detailed information about the family of VAX computers and VAX data types.

Additional prerequisites are described at the beginning of each chapter or appendix, if appropriate.

Document Structure

The *VAX COBOL User Manual* is divided into three parts:

PART I	Developing VAX COBOL Programs
PART II	Using VAX COBOL Features on VMS
PART III	VAX COBOL Programming Options and Performance Considerations

Conventions

The following conventions are used in this manual:

Conventions	Meaning
RET	A symbol with a 1- to 3-character abbreviation indicates that you must press a key on the terminal; for example, RET and TAB indicate that you press the RETURN key and the TAB key on your terminal.
CTRL/x	The symbol CTRL/x indicates that you hold down the key labeled CTRL while you simultaneously press another key; for example, CTRL/C, CTRL/O.
. . .	A vertical series of periods, or an ellipsis, means that not all the data you would enter is shown. All program examples are shown in Digital terminal format, rather than in ANSI standard format.
quotation mark	The term quotation mark is used to refer to the double quotation mark character (").
apostrophe	The term apostrophe is used to refer to the single quotation mark character (').
$	The dollar sign ($) is used to represent the system prompt. Your system might use a different symbol for the system prompt.
user input	In examples in hardcopy versions of this book, user input (what you enter) is shown in red. In online versions, user input is shown in bold.

The VAX COBOL documentation to which this manual belongs refers to these Digital products by their abbreviated names:

- VAX CDD/Plus software is referred to as CDD/Plus.
- VAX DBMS software is referred to as VAX DBMS.
- VAX DEC/Test Manager software is referred to as DEC/Test Manager.
- VAX DEC/Code Management System software is referred to as CMS.
- The VAX Language-Sensitive Editor is referred to as LSE.
- VAX Record Management Services software is referred to as RMS.
- The VAX Text Processing Utility is referred to as VAXTPU.
- The VAX Source Code Analyzer is referred to as SCA.
- The Program Design Facility is referred to as PDF.

Summary of Technical Changes

This section briefly describes the technical changes and new features for VAX COBOL Versions 4.3, 4.2, 4.1, and 4.0. For detailed information on specific changes, refer to the Release Notes for the specific version.

VAX COBOL Versions 4.0 and higher are based on the 1985 ANSI COBOL standard. This manual reflects changes to the VAX COBOL compiler made in these versions. It also includes corrections, additions, clarifications, and other minor improvements.

Version 4.3

The following list briefly describes the technical changes for Version 4.3 of VAX COBOL. For more information, refer to the *VAX COBOL Release Notes*, Version 4.3.

- Support for the VAX Program Design Facility (PDF), including the addition of the /DESIGN command line qualifier. (See Chapter 2 and Appendix E.)
- Support for the VAX Source Code Analyzer (SCA) Version 2.0. (See Appendix E.)
- Support for the DECwindows Compiler Interface (DWCI). (See Chapter 2.)
- Relaxed datatype restrictions of IF SUCCESS/FAILURE and SET SUCCESS/FAILURE statements.
- Support for floating point literals. (See Chapter 1 of the *VAX COBOL Reference Manual.*)
- Support for the three new RMS-CURRENT special registers. (See Chapter 12 and Appendix B and Chapter 1 of the *VAX COBOL Reference Manual.*)
- Addition of VAX COBOL I/O extensions for Descending Key and Duplicate Primary Key.
- Relaxed datatype restrictions for ACCEPT WITH EDITING phrase. (See Chapter 17.)
- Addition of the IDENT clause. (See Chapter 3 of the *VAX COBOL Reference Manual.*)
- Support for Vertical Form Unit (VFU) printers. (See Chapter 8 and Chapter 6 of the *VAX COBOL Reference Manual.*)

Version 4.2

The following list briefly describes the technical changes for Version 4.2 of VAX COBOL. For more information, refer to the online *VAX COBOL Release Notes*, for Version 4.2.

- Support for the VAX License Management Facility (LMF)
- Support for the VAX Source Code Analyzer (SCA). This support includes the addition of the /ANALYSIS_DATA command line qualifier. (See Appendix E.)
- Support for CDD/Plus. This support includes the addition of the /DEPENDENCY_DATA command line qualifier. (See Appendix E.)
- Support for the MULTIPLE FILE TAPE clause.
- Addition of the /INSTRUCTION_SET command line qualifier. (See Appendix D.)
- Addition of the STREAM phrase to the COMMIT statement.
- Addition of the STREAM phrase to the ROLLBACK statement.
- User translatable message file.
- Addition of the EDITING phrase to the ACCEPT statement.

Version 4.1

Version 4.1 of VAX COBOL was a maintenance release and contained no new features. For more information, refer to the online *VAX COBOL Release Notes*, for Version 4.1.

Version 4.0

The following list briefly describes the technical changes for VAX COBOL Version 4.0. For more information, refer to the online *VAX COBOL Release Notes*, for Version 4.0.

- Support for the following syntax constructs has been added:
  - Multistream DBMS access
  - CLASS clause in the SPECIAL-NAMES paragraph
  - START REGARDLESS OF LOCK
  - FETCH, FIND, and STORE with the DB-KEY option
  - Quadword indexed keys
  - CALL literal with ON EXCEPTION phrase
  - Multiple arguments for INSPECT ... ALL/LEADING
  - Conditional NOTs for the following phrases: AT END-OF-PAGE, AT END, INVALID KEY, ON EXCEPTION, ON OVERFLOW, and ON SIZE ERROR
  - The REPLACE statement which allows you to replace source program text
  - Ability to initialize tables using a VALUE clause subordinate to an OCCURS clause
- The optional word TO is now permitted in the ADD ... GIVING statement.
- The figurative constant ALL literal is now permitted in the DISPLAY statement.
- The optional word ALSO is now permitted in the EVALUATE statement.

- An EXIT PROGRAM statement in the body of a main program causes control to be transferred to the next statement.
- The category phrase in the INITIALIZE statement can be repeated.
- The text being replaced using a COPY REPLACING statement cannot consist entirely of a separator comma or semicolon.
- Lines that have been replaced can be viewed in LSE.
- Lines that have been copied can be viewed in the debugger and LSE.
- A paragraph name can be specified in the INPUT and OUTPUT phrases of the SORT and MERGE statements.
- CURRENCY SIGN cannot be a figurative constant.
- CURRENCY SIGN can now be the character L or E.
- A RELATIVE KEY data item cannot contain the symbol P in its PICTURE clause.
- Files with LINAGE clauses cannot be opened in EXTEND mode.
- The size of a variable-length item in an OCCURS DEPENDING ON statement involved in a MOVE is determined by the value of the OCCURS DEPENDING ON item.
- P digit positions are zero for PIC P items in new cases.
- There are new and revised I-O status codes.
- The NO REWIND phrase of the CLOSE statement cannot be specified with the REEL or UNIT phrase.
- The order of evaluation of the identifiers in the PERFORM ... VARYING ... AFTER construct has changed (Format 4 only).
- The ADVANCING PAGE phrase of the WRITE statement cannot be specified with the END-OF-PAGE phrase.
- Shared sequential records can be any length, including variable-length records.
- Scope delimiters no longer must be specified only in terminator clauses such as AT END, ON OVERFLOW, or ON EXCEPTION.
- New options for the /STANDARD command line qualifier enable you to generate code using VAX COBOL Version 3.4 or Version 4.0 rules for certain constructs. (See Appendix D.)
- A new command line qualifier (/FLAGGER) allows you to specify a FIPS level beyond which the FIPS flagging facility produces informational diagnostics. (See Appendix D.)
- The relational operators GREATER THAN OR EQUAL TO and LESS THAN OR EQUAL TO have been added.
- Subscripts and reference modifications for the DIVIDE, STRING, UNSTRING, and INSPECT statements are evaluated differently.
- Optional files can be opened in OUTPUT, I-O, and EXTEND mode.
- There are changes to the rules for opening nonoptional files in I-O and EXTEND mode.
- The COBOL–81 SUBSET Flagger has been updated.
- Support for the PDP–11 Version 4.4 Translator has been eliminated.

- The evaluation of conditional compilation lines takes place after COPY processing instead of before.
- The format of replaced COPY text in the listing file may be different.
- Divide by zero is a continuable error.
- The PROCEDURE DIVISION is now optional in VAX COBOL programs.
- COBOL HELP reflects changes made to the compiler.

Incompatibilities with COBOL–81

The COBOL–81 language is a Digital COBOL compiler that runs under several PDP–11 operating systems. While the COBOL–81 language is a subset of VAX COBOL, there are some architectural differences between the PDP–11 and the VAX processors. The following list gives the known architectural differences.

- INDEX data items in the COBOL–81 language are 2 bytes long; in VAX COBOL, they are 4 bytes long. INDEX data items cause an incompatibility only if they are stored in files. Such files are not directly transferable between the COBOL–81 language and the VAX COBOL language.
- The COBOL–81 compiler aligns COMPUTATIONAL data items on word boundaries by default; the VAX COBOL compiler does not. In order to create files that are directly transferable between COBOL–81 and VAX COBOL, always use the SYNCHRONIZED clause on COMPUTATIONAL data items within record descriptions.
- The VAX processor traps many cases of invalid data in a decimal numeric field that the PDP–11 processor does not. Keep this in mind when debugging COBOL–81 programs. Also, programs that appear to have run without error on the PDP–11 may produce errors on the VAX processor due to invalid decimal data.

Acknowledgement

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein are: FLOW-MATIC (trademark of Unisys Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems, copyrighted 1958, 1959, by Unisys Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

They have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Chairman of the CODASYL COBOL Committee, P.O. Box 3609, Norfolk, VA 23514.

Part I
Developing VAX COBOL Programs

Chapter 1

Overview of VAX COBOL

This brief overview highlights the features of the VAX implementation of COBOL (COmmon Business-Oriented Language). COBOL is widely used throughout the world for business data processing. The features of VAX COBOL listed here are described fully in subsequent chapters of this manual and in the *VAX COBOL Reference Manual*.

VAX COBOL is a high-performance language for commercial application development that runs under the VMS operating system. Version 4.0 of VAX COBOL is based on the *1985 ANSI COBOL Standard X3.23–1985* and *Federal Information Processing Standard 21-2* (FIPS PUB 21–2). The FIPS standard identifies the ANSI standard as the standard adopted by the U.S. federal government and as the criteria on which federal validation is based. Version 4.0 of VAX COBOL also contains Digital extensions to COBOL, including screen handling at the source language level.

VAX COBOL Version 4.0 continues to be highly compatible with previous versions. However, due to changes necessitated by the 1985 ANSI COBOL standard, some differences exist between Version 4.0 and previous versions of VAX COBOL. To minimize the impact of these changes, a new command line qualifier has been added to Version 4.0 of VAX COBOL. For more information see Section 2.5.2 and Appendix D.

The following list highlights some of VAX COBOL's features:

- Some features of traditional structured programming languages are provided by the VAX COBOL compiler, making programs easier to develop, understand, and maintain. These features include scope delimiters, the EVALUATE statement, and in-line PERFORM statements.
- Support for all data types specified in the 1985 ANSI COBOL standard.
- Support for the COBOL data manipulation language interface to VAX DBMS, the Digital CODASYL-compliant database management system.
- Support for the VMS Debugger used for program development.
- Support for multistream DBMS.
- FIPS flagging facility that identifies VAX COBOL extensions, syntax designated as obsolete, or at a specified FIPS level.
- New reserved words and syntax constructs.

VAX COBOL takes full advantage of the VMS operating system facilities and the VAX Information Architecture. VAX COBOL is also integrated with many other Digital products. In particular, VAX COBOL supports the following:

- VAX standard calling procedures, which allow VAX COBOL programs to call (and be called by) other programs written in VAX COBOL, other VAX languages (such as BASIC, FORTRAN, and MACRO), system services, common run-time library subroutines, and screens produced by VAX Forms handling products. VAX COBOL also provides support for contained programs. Contained programs can share resources, such as files, variables, and symbolic characters.

- Record definitions included from CDD/Plus.
- Creation of source code with the VAX Language-Sensitive Editor (LSE) and the use of other VAX productivity tools, such as the VAX COBOL GENERATOR that enables you to automatically generate VAX COBOL source code.
- Extensive online language help.
- Exchange of data with other systems using DECnet.

Version 4.0 of VAX COBOL shares some common syntax with COBOL–81, making it easy to develop error-free COBOL–81 applications for PDP–11 systems. The VAX COBOL compiler also accepts source programs coded in either ANSI standard format or Digital terminal format. You can use the REFORMAT Utility to convert programs written in either format, making conversion and migration of programs written on other systems more efficient.

Part I of this manual shows you how to begin using VAX COBOL on the VMS operating system and how to develop programs at the DCL command level. Part II describes using advanced features of VAX COBOL, and Part III discusses VAX COBOL programming options and enhancements.

Chapter 2

Developing VAX COBOL Programs at DCL Command Level

Developing a VAX COBOL program involves a number of steps, including choosing a reference format for your source program and then creating, compiling, linking, and running it. You can accomplish each of the last four steps by using DCL commands.

This chapter explains how to choose a reference format and how to develop and run a VAX COBOL program at the DCL command level.

2.1 Choosing a Reference Format

Before you can compile a COBOL program, you must decide on a reference format and prepare your source program for input to the compiler. The VAX COBOL compiler accepts source programs written in either terminal reference format or ANSI reference format. However, you cannot mix reference formats in the same compilation unit, even when copying text from a COBOL library. Note that when copying text from CDD/Plus, the COBOL compiler translates the record descriptions into the reference format of the source program.

2.1.1 Terminal Reference Format

Digital recommends that you use terminal format, a Digital optional format, when you create source files from interactive terminals. The compiler accepts it as the default reference format.

Terminal format eliminates the line number and identification fields of ANSI format and allows horizontal tab characters and short lines. This format saves disk space and decreases compile time. Because the spacing requirements of terminal format are more flexible than ANSI format, it is usually easier to edit source programs written in this format.

The following explains the structure and content of a terminal reference source line:

Character Positions	Contents
1 to 4	Area A
5 to 256	Area B
end of line	Margin R

NOTE

> While the maximum size of a terminal line is 256 characters, a source listing line contains 132 characters. However, the first 7 columns of the source listing contain the line numbers of the source code, leaving only 125 spaces per line for text. Therefore, although a terminal line greater than 132 characters compiles, the source listing line shows only the first 125 characters.

You can use the TAB key or the space bar to position source entries in a line. Pressing the RETURN key signifies the end of a line. Terminal format treats the end of each line as Margin R. You must enter continuation (–), comment (*), or skip-to-top-of-page (/) characters in position 1, and conditional compilation characters in columns 1 and 2 (see Chapter 3 for information on conditional compilation characters). For more information about using the TAB key, refer to the *VAX COBOL Reference Manual.*

2.1.2 ANSI Reference Format

ANSI format (defined in the *VAX COBOL Reference Manual*) is useful on a card-oriented system or in an application where strict ANSI compliance is desired. To select ANSI format, specify the command qualifier /ANSI_FORMAT at compile time. You can choose this format if your COBOL program was written for a compiler that used ANSI format.

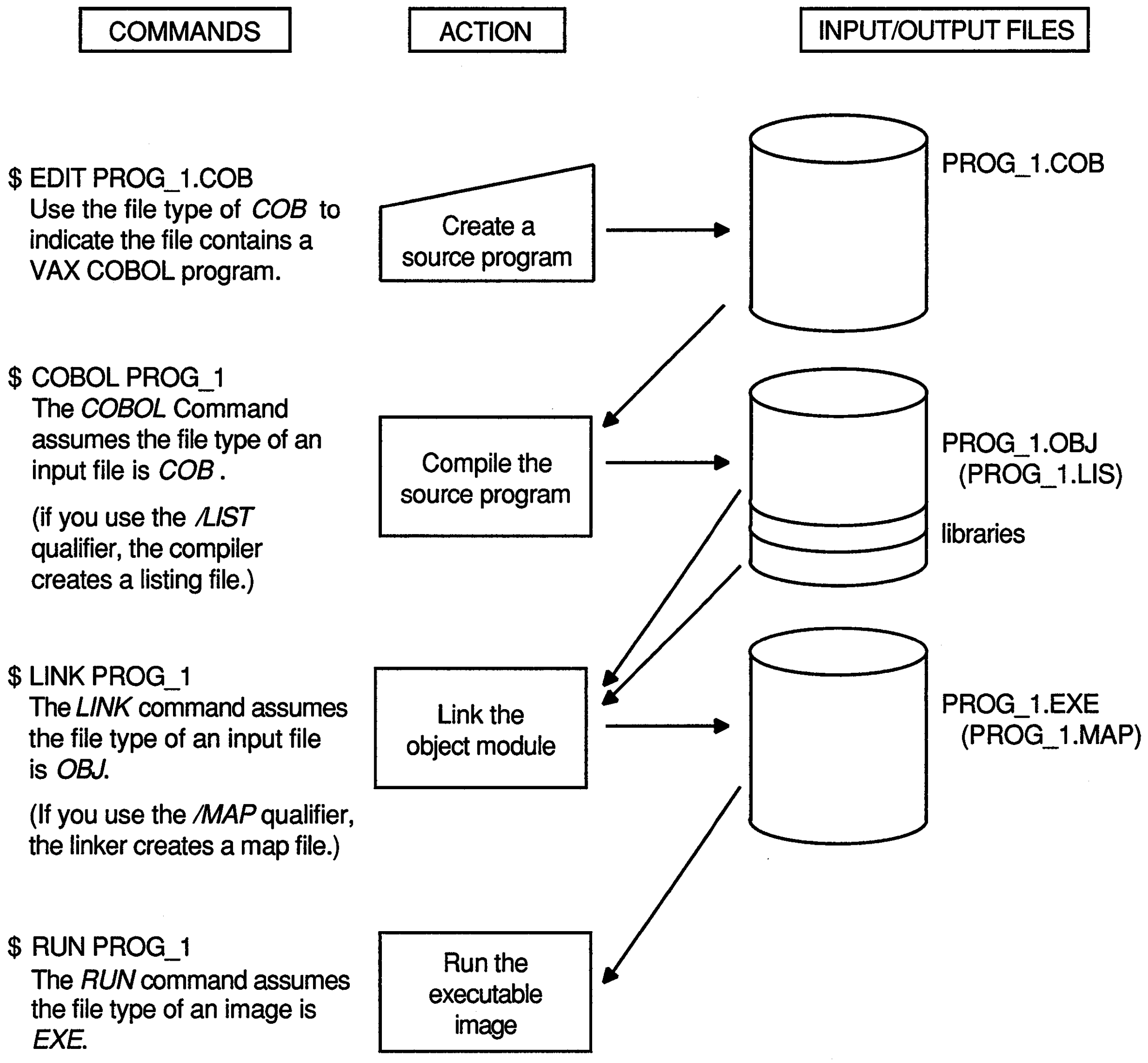
2.1.3 Converting Between Reference Formats

The REFORMAT Utility allows you to convert a terminal format program to ANSI format and vice versa. You can also use REFORMAT to match the formats of source files and COBOL library files when their formats are not the same. See Chapter 19 for a description of the REFORMAT Utility.

2.2 DCL Commands for Program Development

This section briefly describes the DCL commands that are used to create, compile, link, and run a VAX COBOL program on a VMS system. These commands are shown in Figure 2–1. The commands are described in detail later in this chapter.

Figure 2–1: DCL Commands for Developing Programs


COMMANDS
ACTION
INPUT/OUTPUT FILES
$ EDIT PROG_1.COB
Use the file type of COB to indicate the file contains a VAX COBOL program.
Create a source program
PROG_1.COB
$ COBOL PROG_1
The COBOL Command assumes the file type of an input file is COB.
(if you use the /LIST qualifier, the compiler creates a listing file.)
Compile the source program
PROG_1.OBJ
(PROG_1.LIS)
libraries
$ LINK PROG_1
The LINK command assumes the file type of an input file is OBJ.
(If you use the /MAP qualifier, the linker creates a map file.)
Link the object module
PROG_1.EXE
(PROG_1.MAP)
$ RUN PROG_1
The RUN command assumes the file type of an image is EXE.
Run the executable image

ZK-6304-GE

The following example shows each of the commands shown in Figure 2–1 executed in sequence.

```
$ EDIT/EDT PROG_1.COB
$ COBOL PROG_1
$ LINK PROG_1
$ RUN PROG_1
```

To create a VAX COBOL source program at DCL level, you must invoke a text editor. In the previous example, the VAX EDT editor is invoked to create the source program PROG_1.COB. You can, however, use another editor, such as the VAX Text Processing Utility (VAXTPU) or the VAX Language-Sensitive Editor (LSE). COB is used as the file type to indicate that you are creating a VAX COBOL source program. COB is the default file type for all VAX COBOL source programs.

For more information on editors, refer to the appropriate VMS documentation or online help.

When you compile your program with the COBOL command, you do not have to specify the file type; VAX COBOL searches for COB by default.

If your source program compiles successfully, the VAX COBOL compiler creates an object file with the file type OBJ.

However, if the VAX COBOL compiler detects errors in your source program, the system displays each error on your screen and then displays the DCL prompt. You can then reinvoke your text editor to correct each error.

You can include command qualifiers with the COBOL command. Command qualifiers cause the VAX COBOL compiler to perform additional actions. In the following example, the /LIST qualifier causes the VAX COBOL compiler to produce a listing file.

```
$ COBOL/LIST PROG_1
```

The COBOL command qualifiers are explained in Section 2.5.2.

Once your program has compiled successfully, you invoke the VMS Linker to create an executable image file. The VMS Linker uses the object file produced by VAX COBOL as input to produce an executable image file as output.

You can specify command qualifiers with the DCL command LINK. The LINK command qualifiers are explained in Section 2.6.2 and Section 2.6.3.

Once the executable image file has been created, you can run your program with the DCL command RUN.

Table 2–1 provides a brief explanation of VMS file maintenance commands.

Table 2–1: VMS File Maintenance Commands

Category	Command	Function
Creating Files	CREATE	Creates a file from records or data that follows in the input stream; for example, lines entered from a terminal or placed in a batch input file.
	EDIT [/editor]	Invokes one of the VMS interactive editing programs; for example, EDT or LSE.
Correcting and modifying files	EDIT [/editor]	Invokes one of the VMS interactive editors to make changes or additions to a disk file.
Cataloging and organizing files	CREATE/DIRECTORY	Establishes a new directory or hierarchy of directories to catalog files.
	DIRECTORY	Lists files and information about them. Can list files with common file names, or file types, files in one or more directories, files created since a certain date, and so on.
	LIBRARY	Creates and maintains libraries of COPY text modules and libraries of object modules.
	RENAME	Changes the directory a file is cataloged in; or changes the file name, file type, or version number of a file or files.
	SET DEFAULT	Changes the current default device or directory.

(continued on next page)

Table 2–1 (Cont.): VMS File Maintenance Commands

Category	Command	Function
Copying and backing up files	ALLOCATE INITIALIZE MOUNT COPY	Provides device handling and control commands that let you access data written on nonsystem disks, on magnetic tapes, or on punched cards; or to output data to a disk or tape. COPY copies the content of a file or files to another file or files.
Deleting files	DELETE	Removes the directory entry of the file, making the contents of the file inaccessible.
	PURGE	Deletes a specified number of earlier versions of a file or group of files.

2.3 Creating a VAX COBOL Program

To create and modify a VAX COBOL program, you must invoke a text editor. VMS provides you with two text editors: the Digital Standard Editor (EDT) and the VAX Text Processing Utility (VAXTPU). However, other editors, such as the VAX Language-Sensitive Editor (LSE), may be available on your system.

For more information on the editors available, check with your system administrator and refer to the appropriate VMS documentation.

2.4 Using the COPY Statement in Your Source Program

When you create a source program, VAX COBOL allows you to include frequently used text from a VAX Librarian file, a COBOL library file, or CDD/Plus. You gain access to modules in libraries with the COPY statement, in which you specify explicitly the library that contains the library file.

You can also use the COPY FROM DICTIONARY statement to gain access to a data dictionary. The COPY FROM DICTIONARY statement allows you to copy CDD/Plus record descriptions into your source program as COBOL record descriptions.

The COPY statement allows many separate programs to share common source text, reducing development and testing time as well as storage requirements. For example, an application may consist of many separately compiled programs that share the same structure declaration or external variable declarations. It is convenient to maintain only one copy of the declaration of the variables and to include this declaration in each source program with the COPY statement.

The COPY statement causes the compiler to read the file or module specified by that COPY statement during the compilation of a source program. When the compiler reaches the end of the included text, it resumes reading from the previous input file.

Before you can copy record descriptions from CDD/Plus, you must create the record descriptions using the Common Data Dictionary Language (CDDL) or Common Dictionary Operator (CDO).

For more information on using CDD/Plus and creating and maintaining text libraries, refer to Appendix E, the *VAX COBOL Reference Manual*, and the CDD/Plus documentation.

2.5 Compiling a VAX COBOL Program

The primary functions of the VAX COBOL compiler are to:

- Detect errors in your source program
- Display each error on your terminal screen
- Generate machine language instructions from valid source statements
- Group these language instructions into an object module for the linker

When the compiler creates an object module, it provides the linker with the following information:

- The name of the entry point. It takes this name from the program name in the first PROGRAM-ID paragraph in the source program.
- A list of variables that are declared in the module. The linker uses this information when it binds two or more modules together and must resolve references to the same names in the modules.
- Traceback information. Traceback information is used by the system default condition handler when an error occurs that is not handled by the program itself. The traceback information permits the default handler to display a list of the active blocks in the order of activation; this is an aid in program debugging.
- If requested (with the /DEBUG qualifier), a symbol table and a source line correlation table. A symbol table is a list of the names of all external and internal variables within a module, with definitions of their locations. The source line correlation table associates lines in your source file with lines in your program. The compiler creates these tables only if you specifically request them. The tables are of primary help when you use the VMS Debugger.
- If requested (with the /DIAGNOSTICS qualifier), a diagnostics file that is used within an LSE review session.

To invoke the VAX COBOL compiler, you use the DCL command COBOL (explained in Section 2.5.1). With the COBOL command, you can specify command qualifiers. The next two sections discuss in detail the COBOL command and its command qualifiers.

DECwindows Compiler Interface

If you are working from a workstation running DECwindows, the **DECwindows Compiler Interface (DWCI)** enables you to compile source code either from FileView or from within VAX LSE. DWCI is a menu-driven interface that allows you to select compilation options and save the selections as separate configurations for future use. DWCI also contains an extensive online Help facility, to help you make selections.

2.5.1 The COBOL Command

When you compile your source program, use the COBOL command at the DCL prompt. The COBOL command has the following format:

```
COBOL[/command-qualifier] ... {file-spec [/file-qualifier] ...} ...
```

/command-qualifier
The name of a qualifier that indicates a specific action for the compiler to perform on the file or files listed. When a qualifier appears directly after the COBOL command, it affects all files listed. However, when a qualifier appears after a file specification, it affects only the file that immediately precedes it. When files are concatenated, however, these rules do not apply.

file-spec
Indicates the name of the input source file that contains the program or module to be compiled. You are not required to specify a file type; the VAX COBOL compiler assumes the file to be of the default file type COB. If you do not provide a file specification with the COBOL command, the system prompts you for one.

You can supply more than one file specification by separating the file specifications with either a comma (,) or a plus sign (+). If you separate the file specifications with commas, the files are compiled individually. Compiling files in this way allows you to control the number of source files affected by each qualifier. In the following example, the VAX COBOL compiler creates an object file for each source file but creates only a listing file for the source files entitled PROG_1 and PROG_3.

```
$ COBOL/LIST PROG_1, PROG_2/NOLIST, PROG_3
```

If you separate file specifications with plus signs, VAX COBOL appends each of the specified source files and creates one object file and one listing file. For instance, in the following example, only one object file, PROG_1.OBJ, and one listing file, PROG_1.LIS, are created. Both files are named after the first source file in the list.

```
$ COBOL PROG_1 + PROG_2/LIST + PROG_3
```

Note that all qualifiers specified for a single file in a list of files separated with plus signs affect all files within the specified list.

/file-qualifier
The name of a command-qualifier that indicates a specific action for the compiler to perform on the file that immediately precedes the qualifier. When files are concatenated, however, these rules do not apply.

2.5.2 COBOL Command Qualifiers

When you compile your source code, you can include qualifiers. The qualifiers offer you different options for developing, debugging, and documenting programs. Table 2–2 lists each qualifier and the default.

Table 2–2: COBOL Command Qualifiers and Defaults

Command Qualifier	Default
/[NO]ANALYSIS_DATA	/NOANALYSIS_DATA
/[NO]ANSI_FORMAT	/NOANSI_FORMAT
/[NO]AUDIT	/NOAUDIT
/[NO]CHECK	/NOCHECK
/[NO]CONDITIONALS	/NOCONDITIONALS

(continued on next page)

Table 2–2 (Cont.): COBOL Command Qualifiers and Defaults

Command Qualifier	Default
/[NO]COPY_LIST	/NOCOPY_LIST
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/DEBUG=TRACEBACK
/[NO]DEPENDENCY_DATA	/NODEPENDENCY_DATA
/[NO]DESIGN	/NODESIGN
/[NO]DIAGNOSTICS	/NODIAGNOSTICS
/[NO]FIPS=74	/NOFIPS
/[NO]FLAGGER	/NOFLAGGER
/INSTRUCTION_SET	/INSTRUCTION_SET=DECIMAL_STRING
/[NO]LIST	/NOLIST (interactive)
	/LIST (batch)
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]MAP	/NOMAP
/[NO]OBJECT	/OBJECT
/[NO]SEQUENCE_CHECK	/NOSEQUENCE_CHECK
/[NO]STANDARD	/STANDARD=85
/[NO]TRUNCATE	/NOTRUNCATE
/[NO]WARNINGS	/WARNINGS=OTHER

The following text explains each VAX COBOL command line qualifier. Square brackets ([ ]) indicate that the enclosed item is optional. If you specify more than one option for a single qualifier, you must separate each option with a comma and enclose the options in parentheses. A vertical bar (|) between options indicates that you can choose only one of the options listed.

/[NO]ANALYSIS_DATA[=file-spec]

The /[NO]ANALYSIS_DATA qualifier indicates whether or not a .ANA file is created during compilation. The .ANA file is used with the VAX Source Code Analyzer (SCA).

The default is /NOANALYSIS_DATA. For more information, refer to Appendix E and the SCA documentation.

/[NO]ANSI_FORMAT

The /[NO]ANSI_FORMAT qualifier indicates whether the source program is in ANSI (conventional) format or in Digital terminal format.

For ANSI format, the compiler expects 80-character program lines with optional sequence numbers in character positions 1 to 6, indicators in position 7, Area A beginning in position 8, Area B beginning in position 12, and the Identification Area in positions 73 to 80.

By default, the compiler assumes that the source file is in terminal format; that is, Area A begins in record position 1 and Area B in position 5.

/[NO]AUDIT[=(string,...)]

The /[NO]AUDIT qualifier specifies whether user-supplied text is included in a history list entry when a compilation accesses CDD/Plus. /AUDIT without a value specifies that a standard history list entry is created with no additional text. To include more than one line of text, you can enclose up to 64 strings, separated by commas, in parentheses. If /AUDIT is specified, the compiler leaves history list entries in CDD/Plus for database and for COPY FROM DICTIONARY records processed by the compiler. This qualifier also leaves a history list in CDD/Plus for information put in the dictionary as a result of specifying the /DEPENDENCY_DATA qualifier. Only one user-supplied string is included in these entries, even though up to 64 can be specified.

The default, /NOAUDIT, specifies that no history list entry is created.

/[NO]CHECK[=(ALL,[NO]PERFORM,[NO]BOUNDS,NONE)]

The /[NO]CHECK qualifier controls whether the system checks PERFORM statements, indexes, subscripts, reference modification, and the OCCURS DEPENDING ON depending item for specific run-time errors.

Incorrect use of PERFORM statements can produce unpredictable results. If you use the PERFORM parameter and violate either of the following rules, the system generates a run-time error message and aborts the program:

- A paragraph or section that is the subject of a currently active PERFORM statement must be exited before that paragraph or section can be the subject of another PERFORM statement.
- Nested PERFORM ranges (active PERFORM paragraphs or sections containing PERFORMs that execute other paragraphs or sections) must be exited in reverse order of execution.

The BOUNDS option checks the range of subscripts, indexes, and the depending item in the DEPENDING ON phrase of the OCCURS clause. The system generates a run-time error message and aborts the program if it detects one of these errors:

- If DEPENDING ON is not specified and a subscript or index is greater than the upper bound or less than or equal to zero
- If DEPENDING ON is specified and a subscript or index is greater than the depending item or less than or equal to zero
- If a depending item is less than the low bound or greater than the upper bound, and either of these conditions occurs: (1) a subscripted or indexed item references a table; (2) a group containing the table is referenced as a sending item

The /NOCHECK qualifier is equivalent to /CHECK=NONE. If you specify a qualifier parameter, the default options do not change unless individually modified.

/CHECK is equivalent to /CHECK=ALL.

The default is /NOCHECK.

/[NO]CONDITIONALS[=(selector,...)]

The /[NO]CONDITIONALS qualifier controls whether the conditional compilation lines in a source program are compiled or are treated as comments.

Specifying /CONDITIONALS results in all conditional compilation lines being compiled.

Specifying /CONDITIONALS=(selector,...) results in the selected conditional compilation lines being compiled. The conditional-line-selector-list is a parenthesized list of one or more alphabetic characters from A to Z. Chapter 3 discusses COBOL program debugging using conditional compilation lines.

Specifying the default /NOCONDITIONALS results in all conditional compilation lines being treated as comments during compilation.

/[NO]COPY_LIST

The /[NO]COPY_LIST qualifier controls whether source statements included by COPY statements are printed in the listing file. The /COPY_LIST qualifier has no effect unless the /LIST qualifier is also specified.

/NOCOPY_LIST suppresses the listing of text copied from library files; only the COPY statement appears in the listing file.

The default is /NOCOPY_LIST.

/[NO]CROSS_REFERENCE[=(ALPHABETICAL,DECLARED)]

The /[NO]CROSS_REFERENCE qualifier controls whether the source listing includes a cross-reference listing.

If you specify /CROSS_REFERENCE without an option or with the ALPHABETICAL option, the compiler sorts data names and procedure names in alphabetical order and lists them with the source program line numbers on which they appear.

Specifying /CROSS_REFERENCE=DECLARED produces a listing of data names and procedure names in order of declaration.

Specifying /CROSS_REFERENCE=(ALPHABETICAL, DECLARED) produces a listing of data names and procedure names in both alphabetical and declared order in the same compilation.

On the listing, the number sign (#) indicates the source line containing the data name's definition. The asterisk (*) indicates a line on which the associated data item is modified. The /CROSS_REFERENCE qualifier has no effect unless the /LIST qualifier is also specified.

The default is /NOCROSS_REFERENCE.

/[NO]DEBUG[=(ALL,[NO]SYMBOLS,[NO]TRACEBACK,NONE)]

The /[NO]DEBUG qualifier controls whether the compiler produces traceback and local symbol table information for the VMS Debugger. /DEBUG allows you to refer to data items by data name, and to Procedure Division locations by line number, paragraph name, and section name. You can also view source lines from source files and files included by simple COPY statements. The debugger cannot reference source lines from CDD/Plus or any line in which text has been replaced. The /DEBUG qualifier can generate both traceback and symbol table information.

- /DEBUG=ALL is equivalent to /DEBUG=(TRACEBACK,SYMBOLS).
- /DEBUG=SYMBOLS produces a symbol table that allows you to refer to data items by data name and to source lines by line number.
- /DEBUG=TRACEBACK produces traceback information only.
- /DEBUG=NONE is equivalent to /NODEBUG.
- /DEBUG without a value is equivalent to /DEBUG=ALL.

If you specify a qualifier parameter, the default options do not change unless individually modified.

The default is /DEBUG=TRACEBACK.

Chapter 3 discusses COBOL program debugging using the VMS Debugger.

/[NO]DEPENDENCY_DATA

The /[NO]DEPENDENCY_DATA qualifier controls whether or not a compiled module entity is stored in CDD/Plus. This qualifier also controls whether or not a CDD/Plus relationship is established between the compiled module entity and the following:

- All CDD/Plus CDO format dictionary entities specified in COPY FROM DICTIONARY statements
- All CDO format dictionary entities explicitly specified in RECORD DEPENDENCY statements
- The object file created by the compilation

/NODEPENDENCY_DATA indicates that a compiled module entity and CDD/Plus relationships will not be recorded in CDD/Plus.

The default is /NODEPENDENCY_DATA.

For more information, see Appendix E.

/[NO]DESIGN=(COMMENTS,PLACEHOLDERS)

The /[NO]DESIGN qualifier indicates whether or not the compiler will enable Program Design Facility (PDF) processing.

/DESIGN=COMMENTS instructs the compiler to perform PDF comment processing.

/DESIGN=PLACEHOLDERS instructs the compiler to allow PDF placeholders in place of COBOL syntax.

The default is /NODESIGN. If you specify /DESIGN without an option, the default is /DESIGN=(COMMENTS,PLACEHOLDERS). Refer to Appendix E, Section E.1 for additional information.

/[NO]DIAGNOSTICS[=file-spec]

The /[NO]DIAGNOSTICS qualifier controls whether a diagnostic file containing compiler messages and diagnostics information is created. The diagnostic file is reserved for use by Digital. The VAX Language-Sensitive Editor uses the diagnostic file to display diagnostic messages and to position the cursor on the line and column where a source error exists. The default file type for a diagnostic file is DIA.

The default is /NODIAGNOSTICS.

/[NO]FIPS=74

The /[NO]FIPS qualifier allows validation in accordance with the *Federal Information Processing Standard 21–1* (FIPS-PUB 21–1) issued by the U.S. National Bureau of Standards.

The command line qualifier /FIPS=74 supports the *Federal Information Processing Standard 21–1* (FIPS-PUB 21–1) interpretation of File Status and the intermediate arithmetic data item.

FIPS-PUB 21–1 specifies that a File Status of 10 be returned when reporting At End conditions; thus, when you use the /FIPS=74 qualifier, that File Status value is returned.

When /FIPS=74 is specified, the compiler generates code that adheres to the ANSI-74 standard rules for P (picture) characters. If, in any operation involving conversion of data from one form of internal representation to another, the data item being converted is described with the PICTURE character P, each digit position described by a P is considered to contain the value zero, and the size of the data item is considered to include the digit positions so described.

The default, /NOFIPS, causes the compiler to use the algebraic value of the data item described with a P in only certain types of operations:

- Any operation requiring a numeric sending operand
- A MOVE statement where the sending operand is a numeric or numeric edited data item and its PICTURE character-string contains the symbol P, and the receiving operand is numeric or numeric edited
- A comparison operation where both operands are numeric

In all other operations, the digit position specified with the symbol P is ignored and not counted in the size of the operand.

The following table compares the File Status values that are returned when you use or do not use the /FIPS=74 qualifier. Note that these At End File Status values apply to any file organization accessed sequentially. Also note that /FIPS=74 and /NOFIPS only apply when you also specify /STANDARD=V3.

FILE STATUS VALUES

	/FIPS=74	/NOFIPS
The file has no next logical record.	10	13
An optional file was not present.	10	15
The program did not establish a valid next record.	10	16

In addition, the rules on arithmetic operations found in FIPS-PUB 21–1 implicitly require the compiler to keep a 19-digit temporary number and to round on the twentieth digit for add and subtract operations. When you use the /FIPS=74 qualifier, the compiler follows the FIPS-PUB 21–1 rules for arithmetic operations.

/[NO]FLAGGER[=(HIGH_FIPS,INTERMEDIATE_FIPS,MINIMUM_FIPS, OBSOLETE, OPTIONAL_FIPS,REPORT_WRITER,SEGMENTATION, SEGMENTATION_1)]

In accordance with the *Federal Information Processing Standards Publication 21-2* (FIPS-PUB 21-2) issued by the U.S. National Bureau of Standards, VAX COBOL allows you to specify a FIPS level of COBOL syntax beyond which informational diagnostics are generated. To receive the diagnostics, you must specify the /FLAGGER qualifier as well as the /WARNINGS=ALL or /WARNINGS=INFORMATION qualifier.

The /FLAGGER qualifier can be useful when a target system's compiler is known to have a lower level of FIPS syntax support.

When you compile a program using the /FLAGGER qualifier with its options, you receive diagnostic messages for syntax in the source program that is:

- Not within the FIPS validation level you selected
- Within the optional module you selected
- An obsolete language element as defined by the ANSI 1985 standard for the COBOL language
- A Digital extension to the COBOL language

The default is /NOFLAGGER. Also, the /FLAGGER qualifier cannot be specified with /STANDARD=V3. For more information on the /FLAGGER qualifier, refer to Appendix D.

/INSTRUCTION_SET[=[NO]DECIMAL_STRING | GENERIC]

The /INSTRUCTION_SET qualifier indicates whether or not the compiler will optimize code using different portions of the VAX instruction set.

/INSTRUCTION_SET=DECIMAL_STRING instructs the compiler to optimize the code for VAX processors that include the decimal string subset instructions in the hardware.

/INSTRUCTION_SET=NODECIMAL_STRING instructs the compiler to optimize the code for VAX processors that emulate the decimal string subset instructions in the software.

/INSTRUCTION_SET=GENERIC offers a compromise between the other two settings in cases where the instruction set of the target processor is unknown. For more information see Appendix D.

You can choose only one option for this qualifier; multiple options are not allowed. However, you do not have to compile all the modules of an application with the same /INSTRUCTION_SET option value.

NOTE

> Regardless of the /INSTRUCTION_SET option you select, your VAX COBOL program will run on any VAX processor. The /INSTRUCTION_SET qualifier is for optimization purposes only.

The default is /INSTRUCTION_SET=DECIMAL_STRING. For more information on the /INSTRUCTION_SET qualifier, see Appendix D.

/[NO]LIST[=file-spec]

The /[NO]LIST qualifier controls whether the compiler produces an output listing. When you specify /LIST, you can control the defaults applied to the output file specification by your placement of the qualifier in the command. The output file type always defaults to LIS.

Note that the /LIST option is required when you want to use /CROSS_REFERENCE, /COPY_LIST, /FLAGGER, /MACHINE_CODE, or /MAP.

If you use the COBOL command in interactive mode, the default is /NOLIST.

If the COBOL command is executed from a batch job, the default is /LIST.

/[NO]MACHINE_CODE

The /[NO]MACHINE_CODE qualifier controls whether the listing file contains compiler-generated machine code. The /MACHINE_CODE qualifier has no effect unless the /LIST qualifier is also specified.

The default is /NOMACHINE_CODE. For more information on the /[NO]MACHINE_CODE qualifier, refer to Section 2.5.5.4.

/[NO]MAP[=(ALPHABETICAL,DECLARED)]

The /[NO]MAP qualifier controls whether the listing contains maps. Specifying /MAP, /MAP=ALPHABETICAL, or /MAP=DECLARED produces a listing of: (1) data names, procedure names, file names, and their attributes; (2) external references such as user-called routines or Run-Time Library routines; and (3) subschema information including records, sets, and realms. Both

/MAP and /MAP=ALPHABETICAL provide maps in alphabetical order; while /MAP=DECLARED provides maps in declared order. In addition, specifying /MAP=(ALPHABETICAL,DECLARED) produces both alphabetical and declared map listings.

The /MAP qualifier has no effect unless the /LIST qualifier is also specified.

The default is /NOMAP.

/[NO]OBJECT[=file-spec]

The /[NO]OBJECT qualifier controls whether the compiler produces an object file.

By default, the compiler produces an object file with the same file name as the input file and a file type of OBJ. However, you can define a different file name or a different file type by specifying /OBJECT= file-spec. See the VMS documentation on DCL for information on output file specification.

/[NO]SEQUENCE_CHECK

The /[NO]SEQUENCE_CHECK qualifier controls whether the contents of columns 1 to 6 of the source lines are in ascending line number sequence. Out-of-sequence lines produce warning diagnostics. Source programs written in terminal format always pass the sequence check.

The default is /NOSEQUENCE_CHECK.

/[NO]STANDARD[=([NO]85,[NO]V3,[NO]PDP11,[NO]SYNTAX)]

The /[NO]STANDARD qualifier addresses the differences between the following:

- Versions 3 and 4 of VAX COBOL
- VAX COBOL and COBOL–81
- ANSI COBOL and the Digital extensions made to VAX COBOL

The /STANDARD=85 and /STANDARD=V3 options provide the user with a switch for selecting generated code that conforms to the ANSI 1985 standard or to Version 3.4 of VAX COBOL in instances where incompatibilities exist.

If you specify /STANDARD=85, the compiler generates code for certain constructs according to the 1985 ANSI COBOL standard.

If you specify /STANDARD=V3, the compiler generates code in the manner of Version 3.4 of VAX COBOL and issues informational diagnostics for language constructs that would cause different run-time results if /STANDARD=85 had been specified. To receive the diagnostics, you must also use the /WARNINGS=ALL or /WARNINGS=INFORMATION qualifier.

Appendix D provides a detailed description of the differences. If you specify /STANDARD=(85,V3), the compiler generates code as if you specified /STANDARD=85.

If you specify /STANDARD=PDP11, the compiler generates informational diagnostics for VAX COBOL language elements that are outside the COBOL–81 subset and indicates syntactic, semantic, and data allocation differences between VAX COBOL and COBOL–81. Note that generated code is not changed when you select this option. Also note that if you specify /STANDARD=PDP11, you must also use /WARNINGS=ALL or /WARNINGS=INFORMATION.

If you use /STANDARD=SYNTAX, the compiler produces informational diagnostics on language features that are Digital extensions. Therefore, /STANDARD=SYNTAX is equivalent to /WARNINGS=STANDARD. Also note that /STANDARD is equivalent to /STANDARD=SYNTAX.

The default is /STANDARD=85.

/[NO]TRUNCATE

The /[NO]TRUNCATE qualifier specifies how the compiler stores values in COMPUTATIONAL receiving items if high-order truncation is necessary.

If you specify /NOTRUNCATE, the compiler truncates values according to the VAX hardware storage unit (word, longword, or quadword) allocated to the receiving item.

If you specify /TRUNCATE, the compiler truncates values according to the number of decimal digits specified by the PICTURE size. Specifying /TRUNCATE increases program execution time.

In this example, the compiler allocates one word (16 bits) of storage to both A and B:

```
    .
    .
    .
01  A   PIC S99 COMP.
01  B   PIC S9999 COMP.
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
   MOVE B TO A.
   ADD B TO A.
```

When you specify /NOTRUNCATE, all 16 bits of B are moved into A. This may result in a stored value larger than the PICTURE-defined size of two decimal digits.

When you specify /TRUNCATE, the compiler observes the PICTURE size of A and stores only the two low-order digits of B in data item A.

The default is /NOTRUNCATE.

/[NO]WARNINGS[=(ALL,[NO]STANDARD,[NO]INFORMATION, [NO]OTHER,NONE)]

The /[NO]WARNINGS qualifier controls the listing of warning-level and informational-level diagnostics. Specifying STANDARD produces informational diagnostics on language features that are Digital extensions. Specifying INFORMATION produces additional informational diagnostics. Specifying OTHER produces warning-level diagnostics. /WARNINGS is equivalent to specifying /WARNINGS=ALL, while /WARNINGS=NONE is equivalent to specifying /NOWARNINGS.

If you specify a qualifier parameter, the default options do not change unless they are individually modified.

You must specify /WARNINGS=ALL or /WARNINGS=INFORMATION when you use /STANDARD=PDP11.

The default is /WARNINGS=OTHER.

For additional information on the rules for qualifiers used on the VMS operating system, see the syntax rules in the VMS documentation on DCL.

2.5.3 Compiling Programs with Conditional Compilation Lines

To debug source code that contains conditional compilation lines, you can use either the /CONDITIONALS qualifier or the WITH DEBUGGING MODE clause. The /CONDITIONALS qualifier is explained in Section 2.5.2. For more information on the /CONDITIONALS qualifier, refer to the *VAX COBOL Reference Manual.*

Using the WITH DEBUGGING MODE clause as part of the SOURCE-COMPUTER paragraph causes the compiler to process all conditional compilation lines in your source program as COBOL text. If you do not specify the WITH DEBUGGING MODE clause, and if the /CONDITIONALS qualifier is not in effect, all conditional compilation lines in your program are treated as comments.

The WITH DEBUGGING MODE clause applies to: (1) the program that specifies it, and (2) any contained program within a program that specifies the clause.

2.5.4 Compiler Error Messages

If there are errors in your source file when you compile your program, the VAX COBOL compiler flags these errors and displays diagnostic messages. To handle these errors, you must reference the diagnostic message, and locate and correct the problem in your source program.

A sample error message looks like this:

```
   12          PROCEDURE DIVISION.
   13          P-NAME
   14              MOVE ABC TO XYZ.
                   1           2
%COBOL-E-ERROR  65, (1) Missing period is assumed
%COBOL-F-ERROR 349, (2) Undefined name
```

In the sample, error pointer (1) points to the closest approximation to where the error occurred (P-NAME has no period). Error pointer (2) points to an undefined name in source line number 14. The two error pointers are followed by two error message lines that each identify, in this order:

- That the VAX COBOL compiler generated the error message
- The severity code (see Appendix B)
- The error message number
- The error pointers
- The error message

Although most diagnostic messages are self-explanatory, Appendix B contains a list of diagnostic messages that require additional explanation.

The following are some common errors to avoid when entering COBOL command lines:

- Omitting the /ANSI_FORMAT qualifier for source programs that are in ANSI format
- Including contradictory qualifiers, such as /MAP without /LIST, or /FIPS=74 and /STANDARD=85
- Omitting version numbers from file specifications when you want to compile a source program that is not the latest version of a source file

- Forgetting to use a file type in the file specification when you do not want the default file type

To examine diagnostic messages that occurred during compilation, you can print the listing file (or type the file to your terminal screen) and search for each occurrence of %COBOL. Section 2.5.5 details how to read a listing file.

2.5.5 Compiler Listings

A compiler listing provides information that can help you debug your VAX COBOL program. To generate a listing file, specify the /LIST qualifier when you compile your VAX COBOL program interactively. For example:

```
$ COBOL/LIST
```

If the program is compiled as a batch job, the listing file is created by default; specify the /NOLIST qualifier to suppress creation of the listing file. (In either case, the listing file is not automatically printed.) By default, the name of the listing file is the name of the source program followed by a file type of LIS. You can include a file specification with the /LIST qualifier to override this default.

A compiler listing generated by the /LIST qualifier has the following major sections:

- Source Program Listing

  The source program section contains the source code plus line numbers generated by the compiler.

- Storage Map

  The storage map section contains summary information on program sections, variables, and arrays.

- Compilation Summary

  The compilation summary section lists the qualifiers used with the COBOL command and the compilation statistics.

When used with the /LIST qualifier, the following COBOL command qualifiers supply additional information in the compiler listing:

- /COPY_LIST
- /CROSS_REFERENCE
- /FLAGGER
- /MACHINE_CODE
- /MAP
- /STANDARD
- /WARNINGS

See Section 2.5.2 for a description of each qualifier's function.

The next three sections describe each major section of the compiler listing for the program TRBLE.COB generated by the following command:

```
$ COBOL/LIST/COPY_LIST/MAP=(ALPH,DECLARED)/CROSS=(ALPH,DECLARED)
/ANSI_FORMAT TRBLE.COB
```

Section 2.5.5.4 displays the compiler listing generated by specifying the /MACHINE_CODE qualifier for the program MCODE.COB.

Section 2.5.5.5 displays the compiler listing for the contained program TESTA.COB.

2.5.5.1 Source Program Listing

The Source Program section of the compiler listing contains the source code plus line numbers generated by the compiler.

The circled numbers on the program listing TRBLE (see Figure 2–2) correspond to the following numbered text explanations:

❶ The program name as declared in PROGRAM-ID.

❷ The date and time of compilation.

❸ The date and time the file specified in circled number 5 was created.

❹ The name, version, and edit level of the COBOL compiler.

❺ The source file specification, (device:[directory]filename.type;version), which can be up to 255 characters long. The file specification will be trimmed to fit in the listing page header. If text from a copy file is listed at page-break time, the copy file's specification will be printed. The text editor page number (the number (n) in parentheses) will be printed if there is room.

❻ Source line numbers assigned by the compiler. The VMS Debugger uses these line numbers as location specifications.

❼ Sequence numbers. These numbers only appear if the file is in ANSI format.

❽ Source text. Although a terminal line can contain 256 characters, a source listing line contains a maximum of 132 characters.

❾ Identification field. If the source file is in ANSI format, this field contains the identification field (positions 73 to 80).

❿ Line origin information field. A space identifies a line as part of the actual program text. If you use the /COPY_LIST command qualifier at compile time, L identifies a line copied from a library file. If you do not use /COPY_LIST, copied lines are not printed on the source listing. A C identifies a line created when a COPY or REPLACE statement pushes program text from its original line to a new line.

⓫ Line replacement information field. A space indicates no replacement took place. An R identifies a line in which text has been replaced.

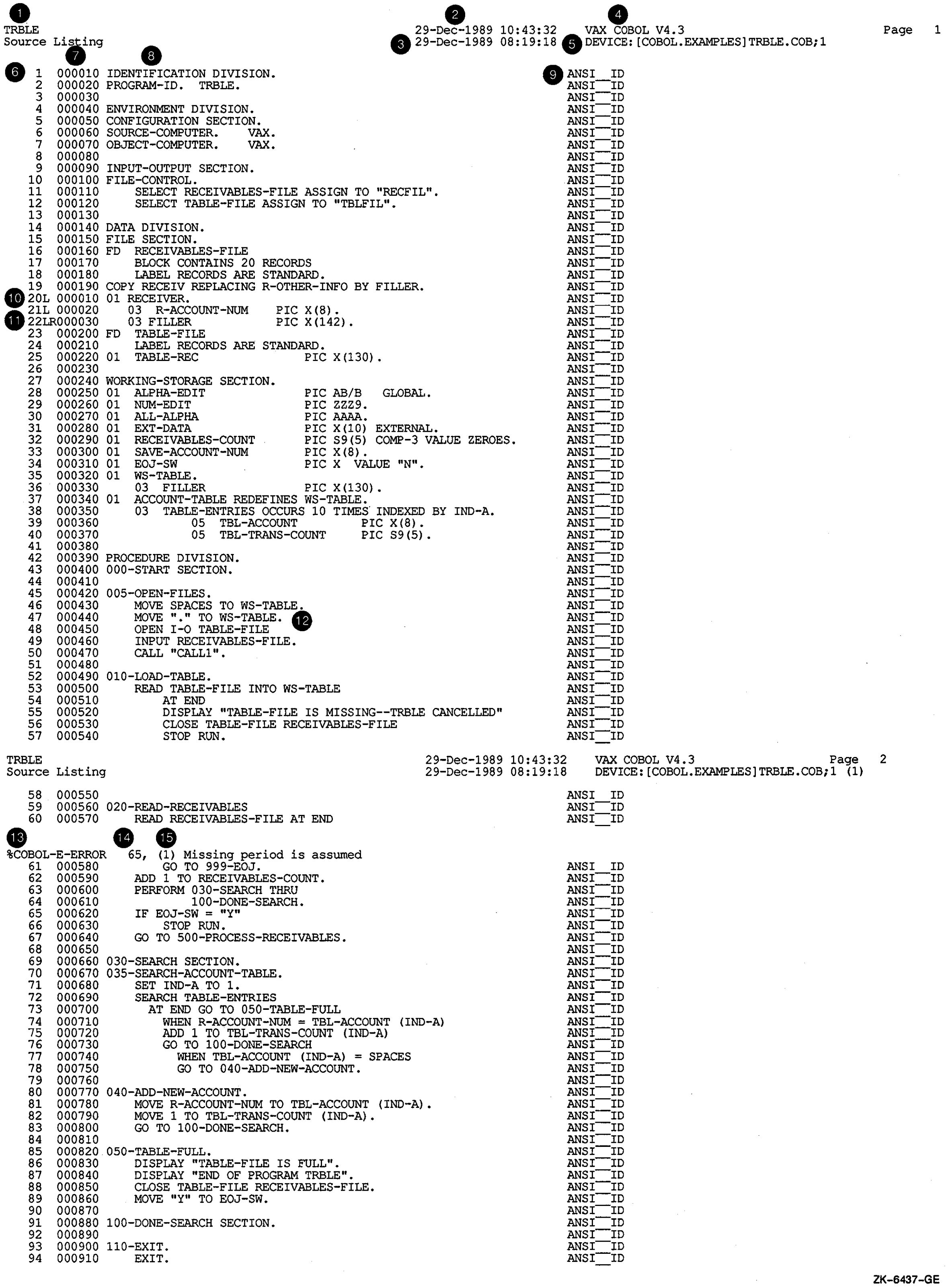
⓬ In the listing file, a period (or other symbol, depending on the specific device) prints in place of any nonprintable ASCII character that was coded in the program.

⓭ Error message line. This line gives the facility name, the error severity code, the error message number, the error pointer, and the error message.

⓮ Error pointer. Points to the closest approximation of where the error occurred.

⓯ Error pointer reference. References the error message to the error pointer.

Figure 2–2: VAX COBOL Source Program Listing

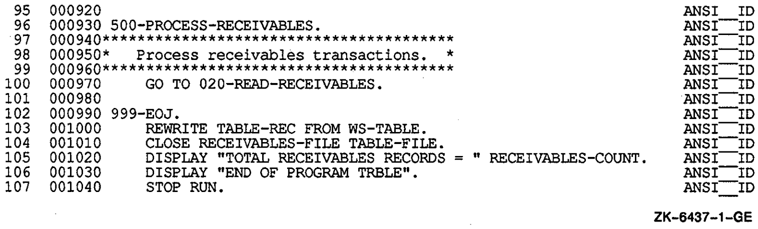


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
ZK-6437-GE

```
TRBLE                                                     29-Dec-1989 10:43:32    VAX COBOL V4.3                             Page   1
Source Listing                                            29-Dec-1989 08:19:18    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1

  1  000010 IDENTIFICATION DIVISION.                                         ANSI__ID
  2  000020 PROGRAM-ID.  TRBLE.                                              ANSI__ID
  3  000030                                                                  ANSI__ID
  4  000040 ENVIRONMENT DIVISION.                                            ANSI__ID
  5  000050 CONFIGURATION SECTION.                                           ANSI__ID
  6  000060 SOURCE-COMPUTER.    VAX.                                         ANSI__ID
  7  000070 OBJECT-COMPUTER.    VAX.                                         ANSI__ID
  8  000080                                                                  ANSI__ID
  9  000090 INPUT-OUTPUT SECTION.                                            ANSI__ID
 10  000100 FILE-CONTROL.                                                    ANSI__ID
 11  000110     SELECT RECEIVABLES-FILE ASSIGN TO "RECFIL".                  ANSI__ID
 12  000120     SELECT TABLE-FILE ASSIGN TO "TBLFIL".                        ANSI__ID
 13  000130                                                                  ANSI__ID
 14  000140 DATA DIVISION.                                                   ANSI__ID
 15  000150 FILE SECTION.                                                    ANSI__ID
 16  000160 FD  RECEIVABLES-FILE                                             ANSI__ID
 17  000170     BLOCK CONTAINS 20 RECORDS                                    ANSI__ID
 18  000180     LABEL RECORDS ARE STANDARD.                                  ANSI__ID
 19  000190 COPY RECEIV REPLACING R-OTHER-INFO BY FILLER.                    ANSI__ID
 20L 000010 01 RECEIVER.                                                     ANSI__ID
 21L 000020    03  R-ACCOUNT-NUM    PIC X(8).                                ANSI__ID
 22LR000030    03 FILLER            PIC X(142).                              ANSI__ID
 23  000200 FD  TABLE-FILE                                                   ANSI__ID
 24  000210     LABEL RECORDS ARE STANDARD.                                  ANSI__ID
 25  000220 01  TABLE-REC               PIC X(130).                          ANSI__ID
 26  000230                                                                  ANSI__ID
 27  000240 WORKING-STORAGE SECTION.                                         ANSI__ID
 28  000250 01  ALPHA-EDIT              PIC AB/B   GLOBAL.                   ANSI__ID
 29  000260 01  NUM-EDIT                PIC ZZZ9.                            ANSI__ID
 30  000270 01  ALL-ALPHA               PIC AAAA.                            ANSI__ID
 31  000280 01  EXT-DATA                PIC X(10) EXTERNAL.                  ANSI__ID
 32  000290 01  RECEIVABLES-COUNT       PIC S9(5) COMP-3 VALUE ZEROES.       ANSI__ID
 33  000300 01  SAVE-ACCOUNT-NUM        PIC X(8).                            ANSI__ID
 34  000310 01  EOJ-SW                  PIC X  VALUE "N".                    ANSI__ID
 35  000320 01  WS-TABLE.                                                    ANSI__ID
 36  000330     03  FILLER              PIC X(130).                          ANSI__ID
 37  000340 01  ACCOUNT-TABLE REDEFINES WS-TABLE.                            ANSI__ID
 38  000350     03  TABLE-ENTRIES OCCURS 10 TIMES INDEXED BY IND-A.          ANSI__ID
 39  000360             05  TBL-ACCOUNT         PIC X(8).                    ANSI__ID
 40  000370             05  TBL-TRANS-COUNT     PIC S9(5).                   ANSI__ID
 41  000380                                                                  ANSI__ID
 42  000390 PROCEDURE DIVISION.                                              ANSI__ID
 43  000400 000-START SECTION.                                               ANSI__ID
 44  000410                                                                  ANSI__ID
 45  000420 005-OPEN-FILES.                                                  ANSI__ID
 46  000430     MOVE SPACES TO WS-TABLE.                                     ANSI__ID
 47  000440     MOVE "." TO WS-TABLE.                                        ANSI__ID
 48  000450     OPEN I-O TABLE-FILE                                          ANSI__ID
 49  000460     INPUT RECEIVABLES-FILE.                                      ANSI__ID
 50  000470     CALL "CALL1".                                                ANSI__ID
 51  000480                                                                  ANSI__ID
 52  000490 010-LOAD-TABLE.                                                  ANSI__ID
 53  000500     READ TABLE-FILE INTO WS-TABLE                                ANSI__ID
 54  000510         AT END                                                   ANSI__ID
 55  000520         DISPLAY "TABLE-FILE IS MISSING--TRBLE CANCELLED"         ANSI__ID
 56  000530         CLOSE TABLE-FILE RECEIVABLES-FILE                        ANSI__ID
 57  000540         STOP RUN.                                                ANSI__ID

TRBLE                                                      29-Dec-1989 10:43:32    VAX COBOL V4.3                   Page   2
Source Listing                                             29-Dec-1989 08:19:18    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

 58  000550                                                                  ANSI__ID
 59  000560 020-READ-RECEIVABLES                                             ANSI__ID
 60  000570     READ RECEIVABLES-FILE AT END                                 ANSI__ID

%COBOL-E-ERROR   65, (1) Missing period is assumed
 61  000580         GO TO 999-EOJ.                                           ANSI__ID
 62  000590     ADD 1 TO RECEIVABLES-COUNT.                                  ANSI__ID
 63  000600     PERFORM 030-SEARCH THRU                                      ANSI__ID
 64  000610             100-DONE-SEARCH.                                     ANSI__ID
 65  000620     IF EOJ-SW = "Y"                                              ANSI__ID
 66  000630         STOP RUN.                                                ANSI__ID
 67  000640     GO TO 500-PROCESS-RECEIVABLES.                               ANSI__ID
 68  000650                                                                  ANSI__ID
 69  000660 030-SEARCH SECTION.                                              ANSI__ID
 70  000670 035-SEARCH-ACCOUNT-TABLE.                                        ANSI__ID
 71  000680     SET IND-A TO 1.                                              ANSI__ID
 72  000690     SEARCH TABLE-ENTRIES                                         ANSI__ID
 73  000700       AT END GO TO 050-TABLE-FULL                                ANSI__ID
 74  000710         WHEN R-ACCOUNT-NUM = TBL-ACCOUNT (IND-A)                 ANSI__ID
 75  000720         ADD 1 TO TBL-TRANS-COUNT (IND-A)                         ANSI__ID
 76  000730         GO TO 100-DONE-SEARCH                                    ANSI__ID
 77  000740           WHEN TBL-ACCOUNT (IND-A) = SPACES                      ANSI__ID
 78  000750           GO TO 040-ADD-NEW-ACCOUNT.                             ANSI__ID
 79  000760                                                                  ANSI__ID
 80  000770 040-ADD-NEW-ACCOUNT.                                             ANSI__ID
 81  000780     MOVE R-ACCOUNT-NUM TO TBL-ACCOUNT (IND-A).                   ANSI__ID
 82  000790     MOVE 1 TO TBL-TRANS-COUNT (IND-A).                           ANSI__ID
 83  000800     GO TO 100-DONE-SEARCH.                                       ANSI__ID
 84  000810                                                                  ANSI__ID
 85  000820 050-TABLE-FULL.                                                  ANSI__ID
 86  000830     DISPLAY "TABLE-FILE IS FULL".                                ANSI__ID
 87  000840     DISPLAY "END OF PROGRAM TRBLE".                              ANSI__ID
 88  000850     CLOSE TABLE-FILE RECEIVABLES-FILE.                           ANSI__ID
 89  000860     MOVE "Y" TO EOJ-SW.                                          ANSI__ID
 90  000870                                                                  ANSI__ID
 91  000880 100-DONE-SEARCH SECTION.                                         ANSI__ID
 92  000890                                                                  ANSI__ID
 93  000900 110-EXIT.                                                        ANSI__ID
 94  000910     EXIT.                                                        ANSI__ID
```

(continued on next page)

Figure 2–2 (Cont.): VAX COBOL Source Program Listing



```
 95  000920                                                                  ANSI_ID
 96  000930 500-PROCESS-RECEIVABLES.                                         ANSI_ID
 97  000940****************************************                          ANSI_ID
 98  000950*   Process receivables transactions.  *                          ANSI_ID
 99  000960****************************************                          ANSI_ID
100  000970     GO TO 020-READ-RECEIVABLES.                                  ANSI_ID
101  000980                                                                  ANSI_ID
102  000990 999-EOJ.                                                         ANSI_ID
103  001000     REWRITE TABLE-REC FROM WS-TABLE.                             ANSI_ID
104  001010     CLOSE RECEIVABLES-FILE TABLE-FILE.                           ANSI_ID
105  001020     DISPLAY "TOTAL RECEIVABLES RECORDS = " RECEIVABLES-COUNT.    ANSI_ID
106  001030     DISPLAY "END OF PROGRAM TRBLE".                              ANSI_ID
107  001040     STOP RUN.                                                    ANSI_ID
```

ZK-6437-1-GE

2.5.5.2 Storage Map Portion of Compiler Listing

The storage map portion of a compiler listing contains summary information on program sections, variables, and arrays.

If you specified the /MAP qualifier, the storage map contains the following information:

- Data names, procedure names, and file names and the attributes of each
- External references
- Subschema information including records, sets, and realms

If you specified the /CROSS_REFERENCE qualifier, the storage map also contains the following cross-reference information:

- Program lines where symbols are defined and initialized
- Program lines where the values of symbols are modified
- Program lines where symbols are actual arguments
- Number of times a symbol occurs in each line

The circled numbers on the program listing TRBLE (see Figure 2–3) correspond to the following numbered text explanations:

❶ File names map. Provides file- and record-specific information. Use the /MAP or /MAP=ALPHABETICAL command qualifier for files that name information in alphabetical order. Use the /MAP=DECLARED command qualifier for files that name information in order of declaration. Use /MAP=(ALPHABETICAL,DECLARED) for files that name information in both alphabetical and declared order.

❷ A list of the file names described in the File Section (File Description (FD) and Sort/Merge Description (SD) entries).

❸ A list of the file's organization as specified in the SELECT clause.

❹ A list of the access mode as specified in the ACCESS MODE clause or the access mode default.

❺ A list of BLOCK CONTAINS attributes and whether they specify number of records (R) or number of characters (C).

❻ A list of the number of characters in a file's records. For variable-length records, the list contains minimum and maximum record length.

❼ A list of record formats. Record formats can be fixed, variable, or print.

❽ Record area. Lists: (1) a record's maximum record length, and (2) its hexadecimal offset location relative to the program section number's (PSECT) beginning location. The PSECT number is the numeric field that precedes the offset location field. PSECT numbers and names are on the Compilation Summary page of a source listing. A location of ** indicates an unreferenced file.

❾ File connector location. Specifies the beginning location of an internal data structure used by the compiled code and the Run-Time Library (RTL). A location of ** indicates an unreferenced file.

❿ Data names map. Lists data items and their attributes. You obtain this listing by specifying the /MAP command qualifier. You can obtain the listing in alphabetical order, declared order, or in both orders (in one compilation) depending on which /MAP option you select (see circled number 1).

⓫ The source line number where the data item is defined.

⓬ The data item's level number.

⓭ The name of the data item.

⓮ Location. Lists the data item's PSECT number and hexadecimal offset location relative to the PSECT's beginning location. PSECT numbers and names are in the Compilation Summary page of a source listing. If the letter L follows the PSECT number, then: (1) the data item is defined in a LINKAGE SECTION, (2) the ordinal position specified in the USING phrase for the record containing the data item is indicated by the PSECT number, and (3) the hexadecimal offset of the data item relative to the record's beginning is identified by location. A location of ** indicates an unreferenced data item. Storage is not allocated for unreferenced data items.

⓯ The data item's field size. For numeric data items, size is defined by the number of nines (9) associated with it in the PICTURE character string.

⓰ The number of bytes allocated to the data item. For numeric values, field size and bytes can be different. (See data name RECEIVABLES-COUNT.)

⓱ Usage. Corresponds to the USAGE clause or implicit usage of the data item. The usage classifications are COMP, COMP-1, COMP-2, COMP-3, DISPLAY, INDEX, and POINTER.

⓲ The category of data described by the data item's PICTURE clause. Category classifications are as follows:

Group	= Group
A	= Alphabetic
AN	= Alphanumeric
ANE	= Alphanumeric Edited
N	= Numeric
NE	= Numeric Edited

⓳ Subs. Lists the number of subscripts required to reference data items.

⓴ Attribute. Indicates whether a data item has an external attribute, a global attribute, or both.

㉑ Procedure names map. Lists procedure names and their attributes. You obtain this listing by specifying the /MAP command qualifier. You can obtain the listing in alphabetical order, declared order, or in both orders (in one compilation), depending on which /MAP option you select (see circled number 1.)

㉒ The source line number, where the procedure name is defined.

㉓ A list of procedure names.

㉔ Location. Lists the procedure name's PSECT number and its hexadecimal offset location relative to the PSECT's beginning. PSECT numbers and names are in the Compilation Summary page of a source listing.

㉕ Lists the procedure name type. Program indicates the PROGRAM-ID name; Section indicates a section name; spaces indicate a paragraph name.

㉖ File names map in declared order. Listed when you use /MAP=DECLARED.

㉗ Data names map in declared order. Listed when you use /MAP=DECLARED.

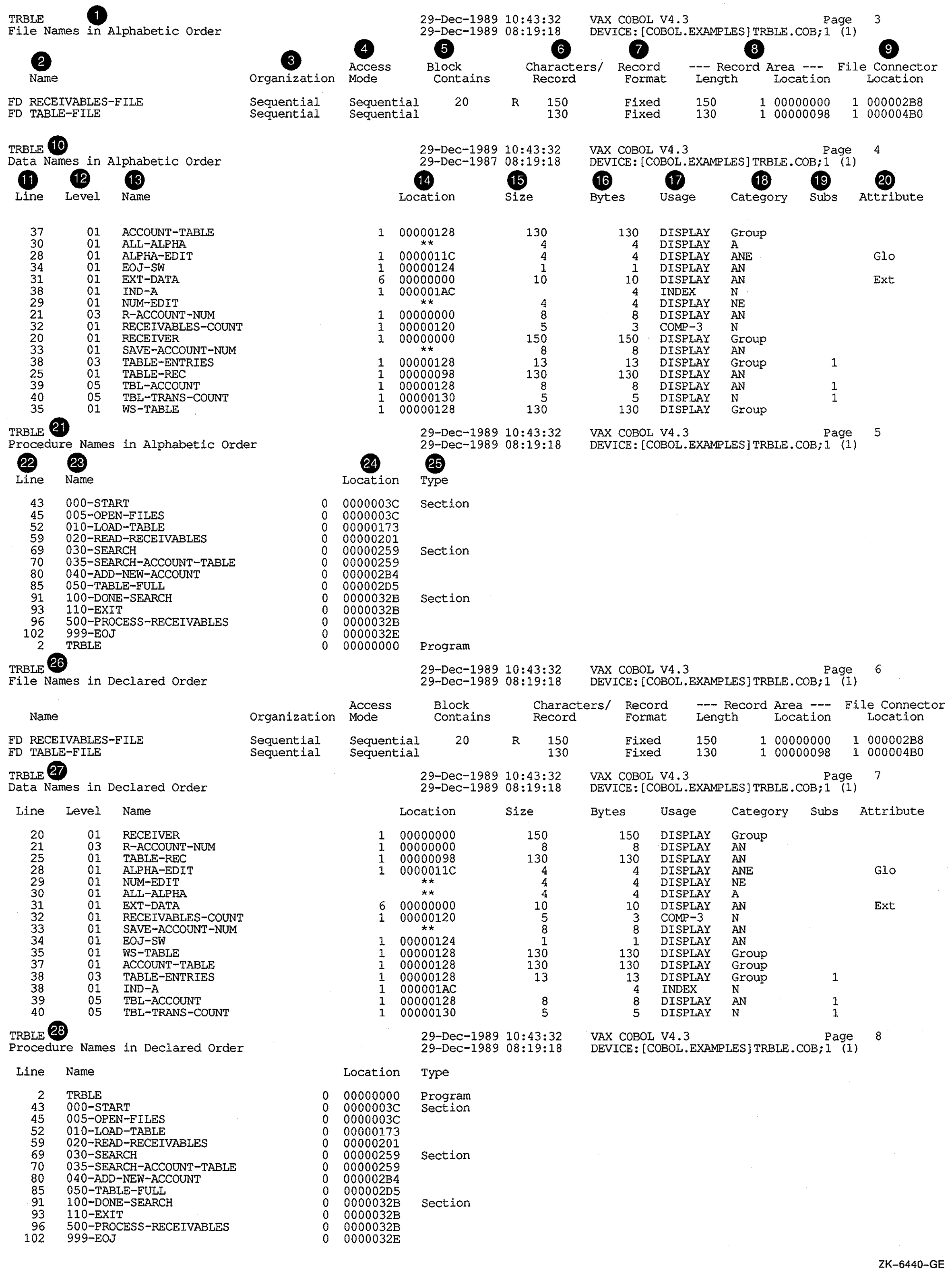
㉘ Procedure names map in declared order. Listed when you use /MAP=DECLARED.

㉙ A cross-reference listing of user-defined names. Specifying the command qualifiers /CROSS_REFERENCE or /CROSS_REFERENCE=ALPHABETICAL produces a sorted listing, while specifying /CROSS_REFERENCE=DECLARED produces a listing in order of declaration. In addition, specifying /CROSS_REFERENCE=(ALPHABETICAL, DECLARED) produces a listing in both alphabetical and declared order. The cross-reference list also includes source line numbers for each item. A source line number followed by a number sign (#) indicates an item's line of definition. Line numbers with an asterisk (*) indicate reference lines in which a destructive reference is made. Line numbers without a number sign or asterisk indicate reference lines.

㉚ A cross-reference listing of user-defined names in declared order. Listed when you use /CROSS_REFERENCE=DECLARED.

㉛ A list of external references. External references can be subprogram calls, calls to the Run-Time Library, or calls to system services. Calls to the Run-Time Library or system services usually originate from compiler-generated object code.

Figure 2–3: Storage Map Portion of VAX COBOL Compiler Listing

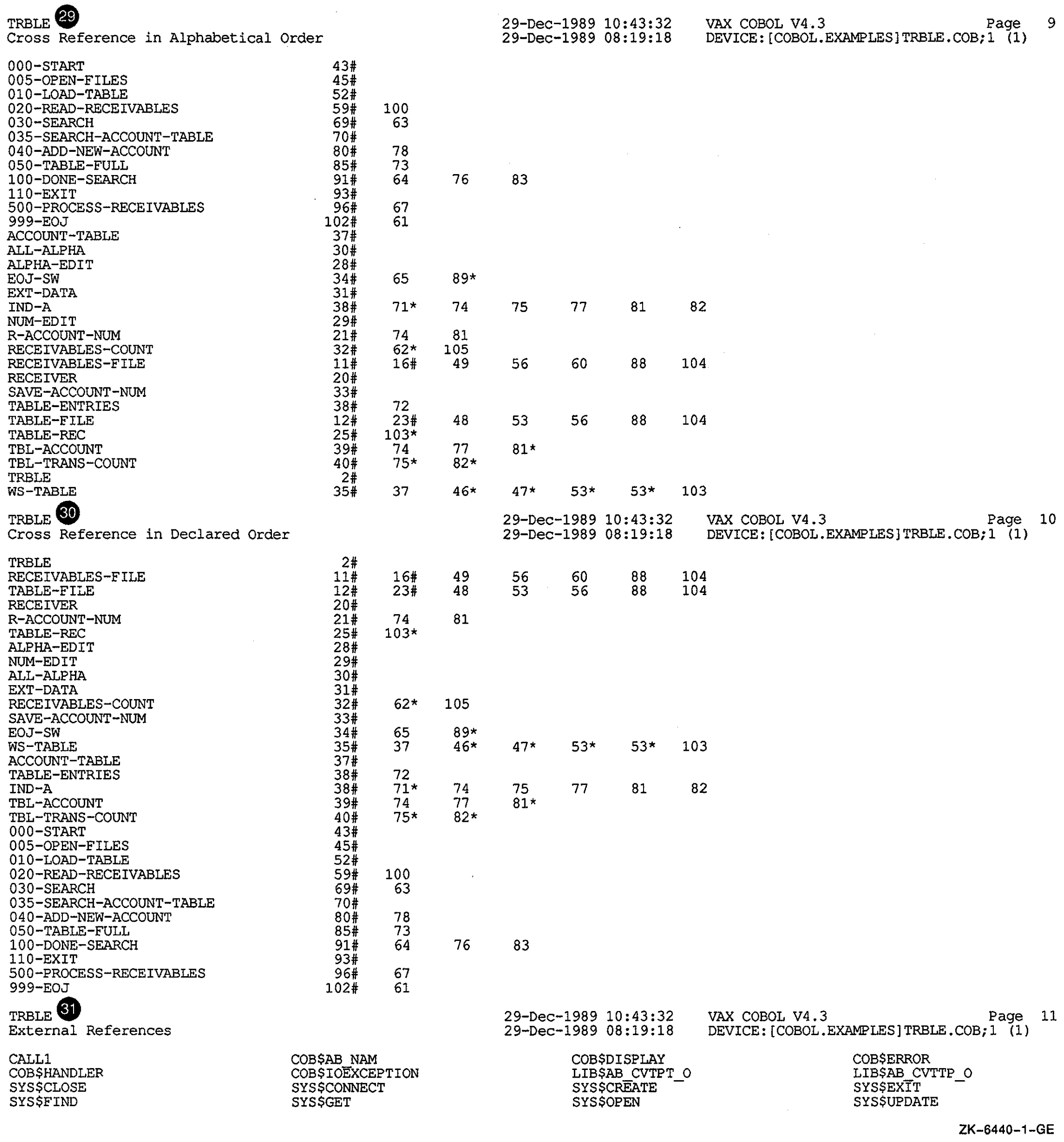


```
TRBLE     ❶                                                   29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   3
File Names in Alphabetic Order                                29-Dec-1989 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

   ❷                                 ❸           ❹        ❺            ❻          ❼            ❽                ❾
   Name                          Organization  Access   Block     Characters/  Record   --- Record Area ---  File Connector
                                               Mode     Contains    Record     Format   Length    Location      Location

FD RECEIVABLES-FILE              Sequential   Sequential   20     R   150      Fixed    150      1 00000000   1 000002B8
FD TABLE-FILE                    Sequential   Sequential              130      Fixed    130      1 00000098   1 000004B0

TRBLE ❿                                                       29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   4
Data Names in Alphabetic Order                                29-Dec-1987 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)
 ⓫     ⓬     ⓭                                         ⓮            ⓯          ⓰         ⓱         ⓲        ⓳      ⓴
 Line  Level  Name                                    Location       Size       Bytes     Usage     Category  Subs   Attribute

   37    01   ACCOUNT-TABLE                         1 00000128        130        130     DISPLAY   Group
   30    01   ALL-ALPHA                                **               4          4     DISPLAY   A
   28    01   ALPHA-EDIT                            1 0000011C          4          4     DISPLAY   ANE                  Glo
   34    01   EOJ-SW                                1 00000124          1          1     DISPLAY   AN
   31    01   EXT-DATA                              6 00000000         10         10     DISPLAY   AN                   Ext
   38    01   IND-A                                 1 000001AC                     4     INDEX     N
   29    01   NUM-EDIT                                 **               4          4     DISPLAY   NE
   21    03   R-ACCOUNT-NUM                         1 00000000          8          8     DISPLAY   AN
   32    01   RECEIVABLES-COUNT                     1 00000120          5          3     COMP-3    N
   20    01   RECEIVER                              1 00000000        150        150     DISPLAY   Group
   33    01   SAVE-ACCOUNT-NUM                         **               8          8     DISPLAY   AN
   38    03   TABLE-ENTRIES                         1 00000128         13         13     DISPLAY   Group      1
   25    01   TABLE-REC                             1 00000098        130        130     DISPLAY   AN
   39    05   TBL-ACCOUNT                           1 00000128          8          8     DISPLAY   AN         1
   40    05   TBL-TRANS-COUNT                       1 00000130          5          5     DISPLAY   N          1
   35    01   WS-TABLE                              1 00000128        130        130     DISPLAY   Group
TRBLE ㉑                                                      29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   5
Procedure Names in Alphabetic Order                           29-Dec-1989 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)
 ㉒    ㉓                                        ㉔         ㉕
 Line  Name                                    Location   Type

   43  000-START                             0 0000003C   Section
   45  005-OPEN-FILES                        0 0000003C
   52  010-LOAD-TABLE                        0 00000173
   59  020-READ-RECEIVABLES                  0 00000201
   69  030-SEARCH                            0 00000259   Section
   70  035-SEARCH-ACCOUNT-TABLE              0 00000259
   80  040-ADD-NEW-ACCOUNT                   0 000002B4
   85  050-TABLE-FULL                        0 000002D5
   91  100-DONE-SEARCH                       0 0000032B   Section
   93  110-EXIT                              0 0000032B
   96  500-PROCESS-RECEIVABLES               0 0000032B
  102  999-EOJ                               0 0000032E
    2  TRBLE                                 0 00000000   Program
TRBLE ㉖                                                      29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   6
File Names in Declared Order                                  29-Dec-1989 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

                                               Access   Block     Characters/  Record   --- Record Area ---  File Connector
   Name                          Organization  Mode     Contains    Record     Format   Length    Location      Location

FD RECEIVABLES-FILE              Sequential   Sequential   20     R   150      Fixed    150      1 00000000   1 000002B8
FD TABLE-FILE                    Sequential   Sequential              130      Fixed    130      1 00000098   1 000004B0
TRBLE ㉗                                                      29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   7
Data Names in Declared Order                                  29-Dec-1989 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

 Line  Level  Name                                    Location       Size       Bytes     Usage     Category  Subs   Attribute

   20    01   RECEIVER                              1 00000000        150        150     DISPLAY   Group
   21    03   R-ACCOUNT-NUM                         1 00000000          8          8     DISPLAY   AN
   25    01   TABLE-REC                             1 00000098        130        130     DISPLAY   AN
   28    01   ALPHA-EDIT                            1 0000011C          4          4     DISPLAY   ANE                  Glo
   29    01   NUM-EDIT                                 **               4          4     DISPLAY   NE
   30    01   ALL-ALPHA                                **               4          4     DISPLAY   A
   31    01   EXT-DATA                              6 00000000         10         10     DISPLAY   AN                   Ext
   32    01   RECEIVABLES-COUNT                     1 00000120          5          3     COMP-3    N
   33    01   SAVE-ACCOUNT-NUM                         **               8          8     DISPLAY   AN
   34    01   EOJ-SW                                1 00000124          1          1     DISPLAY   AN
   35    01   WS-TABLE                              1 00000128        130        130     DISPLAY   Group
   37    01   ACCOUNT-TABLE                         1 00000128        130        130     DISPLAY   Group
   38    03   TABLE-ENTRIES                         1 00000128         13         13     DISPLAY   Group      1
   38    01   IND-A                                 1 000001AC                     4     INDEX     N
   39    05   TBL-ACCOUNT                           1 00000128          8          8     DISPLAY   AN         1
   40    05   TBL-TRANS-COUNT                       1 00000130          5          5     DISPLAY   N          1
TRBLE ㉘                                                      29-Dec-1989 10:43:32   VAX COBOL V4.3                   Page   8
Procedure Names in Declared Order                             29-Dec-1989 08:19:18   DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

 Line  Name                                    Location   Type

    2  TRBLE                                 0 00000000   Program
   43  000-START                             0 0000003C   Section
   45  005-OPEN-FILES                        0 0000003C
   52  010-LOAD-TABLE                        0 00000173
   59  020-READ-RECEIVABLES                  0 00000201
   69  030-SEARCH                            0 00000259   Section
   70  035-SEARCH-ACCOUNT-TABLE              0 00000259
   80  040-ADD-NEW-ACCOUNT                   0 000002B4
   85  050-TABLE-FULL                        0 000002D5
   91  100-DONE-SEARCH                       0 0000032B   Section
   93  110-EXIT                              0 0000032B
   96  500-PROCESS-RECEIVABLES               0 0000032B
  102  999-EOJ                               0 0000032E
```

ZK-6440-GE

(continued on next page)

Figure 2–3 (Cont.): Storage Map Portion of VAX COBOL Compiler Listing



```
TRBLE ㉙                                                     29-Dec-1989 10:43:32    VAX COBOL V4.3                  Page   9
Cross Reference in Alphabetical Order                        29-Dec-1989 08:19:18    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

000-START                              43#
005-OPEN-FILES                         45#
010-LOAD-TABLE                         52#
020-READ-RECEIVABLES                   59#   100
030-SEARCH                             69#    63
035-SEARCH-ACCOUNT-TABLE               70#
040-ADD-NEW-ACCOUNT                    80#    78
050-TABLE-FULL                         85#    73
100-DONE-SEARCH                        91#    64     76     83
110-EXIT                               93#
500-PROCESS-RECEIVABLES                96#    67
999-EOJ                               102#    61
ACCOUNT-TABLE                          37#
ALL-ALPHA                              30#
ALPHA-EDIT                             28#
EOJ-SW                                 34#    65     89*
EXT-DATA                               31#
IND-A                                  38#    71*    74     75     77     81     82
NUM-EDIT                               29#
R-ACCOUNT-NUM                          21#    74     81
RECEIVABLES-COUNT                      32#    62*   105
RECEIVABLES-FILE                       11#    16#    49     56     60     88    104
RECEIVER                               20#
SAVE-ACCOUNT-NUM                       33#
TABLE-ENTRIES                          38#    72
TABLE-FILE                             12#    23#    48     53     56     88    104
TABLE-REC                              25#   103*
TBL-ACCOUNT                            39#    74     77     81*
TBL-TRANS-COUNT                        40#    75*    82*
TRBLE                                   2#
WS-TABLE                               35#    37     46*    47*    53*    53*   103
TRBLE ㉚                                                     29-Dec-1989 10:43:32    VAX COBOL V4.3                  Page  10
Cross Reference in Declared Order                            29-Dec-1989 08:19:18    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

TRBLE                                   2#
RECEIVABLES-FILE                       11#    16#    49     56     60     88    104
TABLE-FILE                             12#    23#    48     53     56     88    104
RECEIVER                               20#
R-ACCOUNT-NUM                          21#    74     81
TABLE-REC                              25#   103*
ALPHA-EDIT                             28#
NUM-EDIT                               29#
ALL-ALPHA                              30#
EXT-DATA                               31#
RECEIVABLES-COUNT                      32#    62*   105
SAVE-ACCOUNT-NUM                       33#
EOJ-SW                                 34#    65     89*
WS-TABLE                               35#    37     46*    47*    53*    53*   103
ACCOUNT-TABLE                          37#
TABLE-ENTRIES                          38#    72
IND-A                                  38#    71*    74     75     77     81     82
TBL-ACCOUNT                            39#    74     77     81*
TBL-TRANS-COUNT                        40#    75*    82*
000-START                              43#
005-OPEN-FILES                         45#
010-LOAD-TABLE                         52#
020-READ-RECEIVABLES                   59#   100
030-SEARCH                             69#    63
035-SEARCH-ACCOUNT-TABLE               70#
040-ADD-NEW-ACCOUNT                    80#    78
050-TABLE-FULL                         85#    73
100-DONE-SEARCH                        91#    64     76     83
110-EXIT                               93#
500-PROCESS-RECEIVABLES                96#    67
999-EOJ                               102#    61
TRBLE ㉛                                                     29-Dec-1989 10:43:32    VAX COBOL V4.3                  Page  11
External References                                          29-Dec-1989 08:19:18    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

CALL1                              COB$AB_NAM                         COB$DISPLAY                        COB$ERROR
COB$HANDLER                        COB$IOEXCEPTION                    LIB$AB_CVTPT_O                     LIB$AB_CVTTP_O
SYS$CLOSE                          SYS$CONNECT                        SYS$CREATE                         SYS$EXIT
SYS$FIND                           SYS$GET                            SYS$OPEN                           SYS$UPDATE
```

ZK-6440-1-GE

2.5.5.3 Compilation Summary

The compilation summary lists the qualifiers used with the COBOL command and the compilation statistics.

The circled numbers on the program listing TRBLE (see Figure 2–4) correspond to the following numbered text explanations.

❶ Compilation summary. A summary of compilation activities.

❷ Program Sections. Describe PSECT attributes.

❸ A list of PSECT numbers and PSECT names.

❹ The bytes allocated for each PSECT.

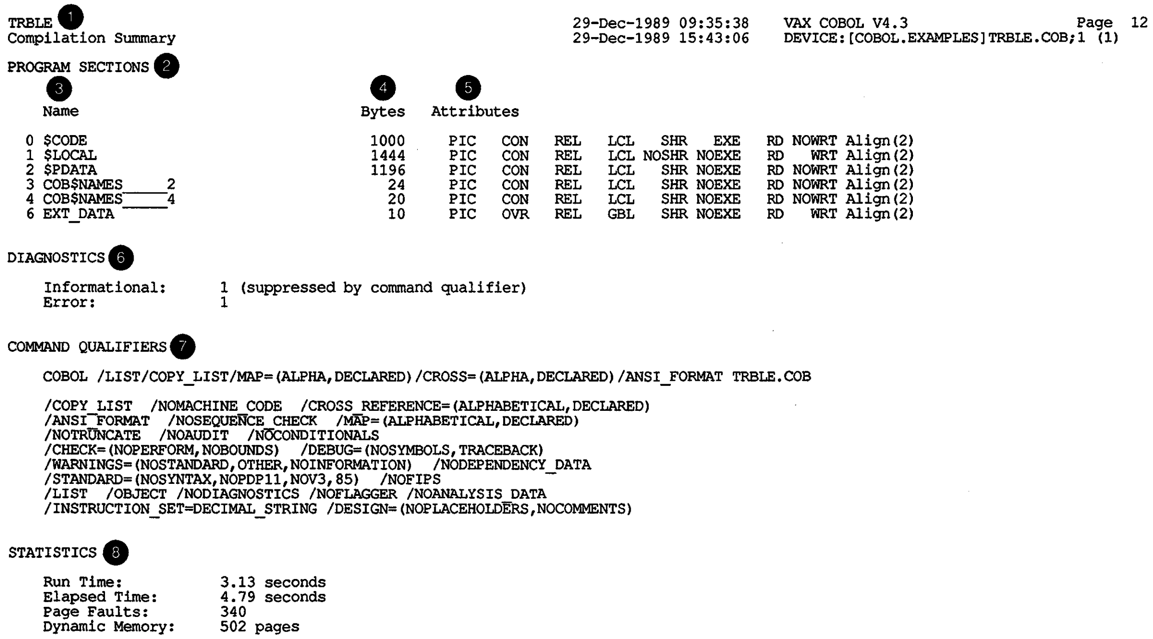
❺ A list of PSECT attributes. For an explanation of PSECT attributes, see the VMS documentation on linking programs.

❻ A summary total, by diagnostic level, of compiler-generated diagnostics. Diagnostics can be Informational (I), Warning (W), Error (E), or Fatal (F).

❼ COBOL command qualifiers. The first line of command qualifiers is the compiler command line. The remaining qualifiers are the command qualifiers and the command qualifier defaults in effect at compile time.

❽ Compile time statistics. These statistics include run or CPU time, elapsed or clock time, the number of page faults, and the number of virtual memory pages used to compile the program.

Figure 2–4: Compilation Summary of a VAX COBOL Source Program Listing



```
TRBLE ❶                                           29-Dec-1989 09:35:38    VAX COBOL V4.3                    Page  12
Compilation Summary                               29-Dec-1989 15:43:06    DEVICE:[COBOL.EXAMPLES]TRBLE.COB;1 (1)

PROGRAM SECTIONS ❷
   ❸                                    ❹      ❺
   Name                                Bytes  Attributes

 0 $CODE                                1000    PIC   CON   REL   LCL   SHR   EXE   RD NOWRT Align(2)
 1 $LOCAL                               1444    PIC   CON   REL   LCL NOSHR NOEXE   RD   WRT Align(2)
 2 $PDATA                               1196    PIC   CON   REL   LCL   SHR NOEXE   RD NOWRT Align(2)
 3 COB$NAMES_____2                        24    PIC   CON   REL   LCL   SHR NOEXE   RD NOWRT Align(2)
 4 COB$NAMES_____4                        20    PIC   CON   REL   LCL   SHR NOEXE   RD NOWRT Align(2)
 6 EXT_DATA                               10    PIC   OVR   REL   GBL   SHR NOEXE   RD   WRT Align(2)

DIAGNOSTICS ❻

   Informational:      1 (suppressed by command qualifier)
   Error:              1

COMMAND QUALIFIERS ❼

   COBOL /LIST/COPY_LIST/MAP=(ALPHA,DECLARED)/CROSS=(ALPHA,DECLARED)/ANSI_FORMAT TRBLE.COB

   /COPY_LIST  /NOMACHINE_CODE  /CROSS_REFERENCE=(ALPHABETICAL,DECLARED)
   /ANSI_FORMAT  /NOSEQUENCE_CHECK  /MAP=(ALPHABETICAL,DECLARED)
   /NOTRUNCATE  /NOAUDIT  /NOCONDITIONALS
   /CHECK=(NOPERFORM,NOBOUNDS)  /DEBUG=(NOSYMBOLS,TRACEBACK)
   /WARNINGS=(NOSTANDARD,OTHER,NOINFORMATION)  /NODEPENDENCY_DATA
   /STANDARD=(NOSYNTAX,NOPDP11,NOV3,85)  /NOFIPS
   /LIST  /OBJECT /NODIAGNOSTICS /NOFLAGGER /NOANALYSIS_DATA
   /INSTRUCTION_SET=DECIMAL_STRING /DESIGN=(NOPLACEHOLDERS,NOCOMMENTS)

STATISTICS ❽

   Run Time:           3.13 seconds
   Elapsed Time:       4.79 seconds
   Page Faults:        340
   Dynamic Memory:     502 pages
```

ZK-6443-GE

2.5.5.4 Compiler Listing Including the /MACHINE_CODE Qualifier

If you specified the /MACHINE_CODE qualifier, your listing includes a section displaying compiler-generated object code. Figure 2–5 shows a compiler listing generated by specifying the /MACHINE_CODE qualifier for the program MCODE.COB.

The circled numbers on the program listing MCODE (see Figure 2–5) correspond to the following numbered text explanations.

❶ The hexadecimal offset location of a machine code instruction or a pseudoinstruction relative to the PSECT's beginning location.

❷ A machine code instruction or a pseudoinstruction.

❸ A machine code instruction or a pseudoinstruction's operands.

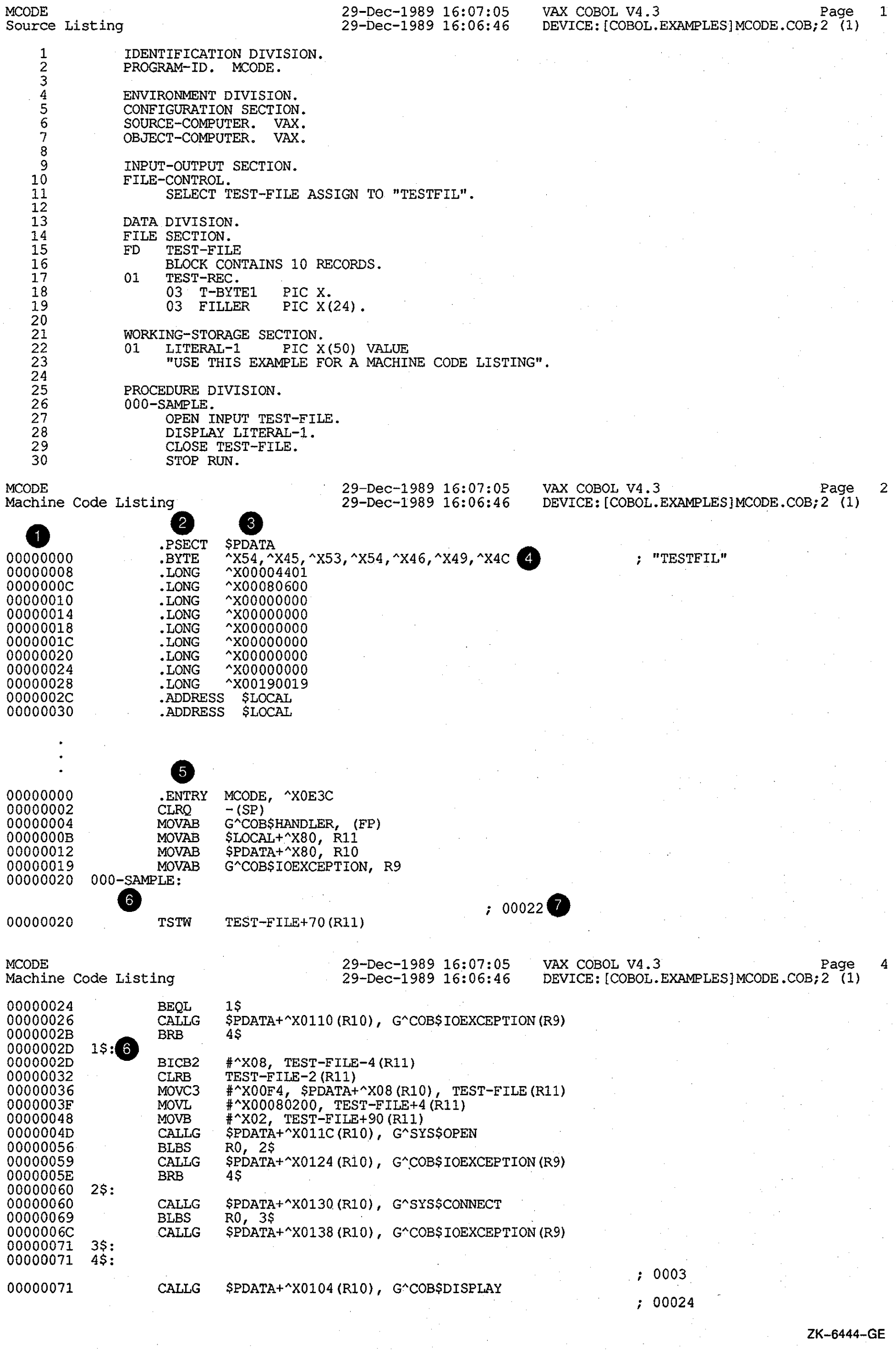
❹ A list of the ASCII representation of literals.

❺ The word ENTRY defines the entry point.

❻ COBOL procedure names and machine-code-generated local labels.

❼ The source line number containing the COBOL statement that generated the machine instructions.

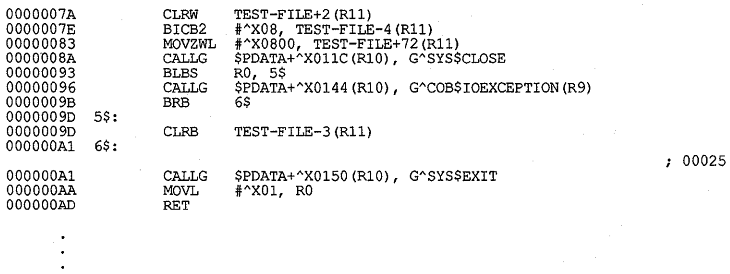
Figure 2–5: VAX COBOL Listing Specifying /MACHINE_CODE Qualifier


1
2
3
4
5
6
7
6
ZK-6444-GE

```
MCODE                                   29-Dec-1989 16:07:05    VAX COBOL V4.3                  Page   1
Source Listing                          29-Dec-1989 16:06:46    DEVICE:[COBOL.EXAMPLES]MCODE.COB;2 (1)

     1        IDENTIFICATION DIVISION.
     2        PROGRAM-ID.  MCODE.
     3
     4        ENVIRONMENT DIVISION.
     5        CONFIGURATION SECTION.
     6        SOURCE-COMPUTER.  VAX.
     7        OBJECT-COMPUTER.  VAX.
     8
     9        INPUT-OUTPUT SECTION.
    10        FILE-CONTROL.
    11             SELECT TEST-FILE ASSIGN TO "TESTFIL".
    12
    13        DATA DIVISION.
    14        FILE SECTION.
    15        FD   TEST-FILE
    16             BLOCK CONTAINS 10 RECORDS.
    17        01   TEST-REC.
    18             03  T-BYTE1    PIC X.
    19             03  FILLER     PIC X(24).
    20
    21        WORKING-STORAGE SECTION.
    22        01   LITERAL-1      PIC X(50) VALUE
    23             "USE THIS EXAMPLE FOR A MACHINE CODE LISTING".
    24
    25        PROCEDURE DIVISION.
    26        000-SAMPLE.
    27             OPEN INPUT TEST-FILE.
    28             DISPLAY LITERAL-1.
    29             CLOSE TEST-FILE.
    30             STOP RUN.

MCODE                                   29-Dec-1989 16:07:05    VAX COBOL V4.3                  Page   2
Machine Code Listing                    29-Dec-1989 16:06:46    DEVICE:[COBOL.EXAMPLES]MCODE.COB;2 (1)

                    .PSECT  $PDATA
00000000            .BYTE   ^X54,^X45,^X53,^X54,^X46,^X49,^X4C              ; "TESTFIL"
00000008            .LONG   ^X00004401
0000000C            .LONG   ^X00080600
00000010            .LONG   ^X00000000
00000014            .LONG   ^X00000000
00000018            .LONG   ^X00000000
0000001C            .LONG   ^X00000000
00000020            .LONG   ^X00000000
00000024            .LONG   ^X00000000
00000028            .LONG   ^X00190019
0000002C            .ADDRESS  $LOCAL
00000030            .ADDRESS  $LOCAL

      .
      .
      .

00000000            .ENTRY  MCODE, ^X0E3C
00000002            CLRQ    -(SP)
00000004            MOVAB   G^COB$HANDLER, (FP)
0000000B            MOVAB   $LOCAL+^X80, R11
00000012            MOVAB   $PDATA+^X80, R10
00000019            MOVAB   G^COB$IOEXCEPTION, R9
00000020  000-SAMPLE:
                                                      ; 00022
00000020            TSTW    TEST-FILE+70(R11)

MCODE                                   29-Dec-1989 16:07:05    VAX COBOL V4.3                  Page   4
Machine Code Listing                    29-Dec-1989 16:06:46    DEVICE:[COBOL.EXAMPLES]MCODE.COB;2 (1)

00000024            BEQL    1$
00000026            CALLG   $PDATA+^X0110(R10), G^COB$IOEXCEPTION(R9)
0000002B            BRB     4$
0000002D  1$:
0000002D            BICB2   #^X08, TEST-FILE-4(R11)
00000032            CLRB    TEST-FILE-2(R11)
00000036            MOVC3   #^X00F4, $PDATA+^X08(R10), TEST-FILE(R11)
0000003F            MOVL    #^X00080200, TEST-FILE+4(R11)
00000048            MOVB    #^X02, TEST-FILE+90(R11)
0000004D            CALLG   $PDATA+^X011C(R10), G^SYS$OPEN
00000056            BLBS    R0, 2$
00000059            CALLG   $PDATA+^X0124(R10), G^COB$IOEXCEPTION(R9)
0000005E            BRB     4$
00000060  2$:
00000060            CALLG   $PDATA+^X0130(R10), G^SYS$CONNECT
00000069            BLBS    R0, 3$
0000006C            CALLG   $PDATA+^X0138(R10), G^COB$IOEXCEPTION(R9)
00000071  3$:
00000071  4$:
                                                                        ; 0003
00000071            CALLG   $PDATA+^X0104(R10), G^COB$DISPLAY
                                                                        ; 00024
```

(continued on next page)

Figure 2–5 (Cont.): VAX COBOL Listing Specifying /MACHINE_CODE Qualifier



```
0000007A          CLRW     TEST-FILE+2(R11)
0000007E          BICB2    #^X08, TEST-FILE-4(R11)
00000083          MOVZWL   #^X0800, TEST-FILE+72(R11)
0000008A          CALLG    $PDATA+^X011C(R10), G^SYS$CLOSE
00000093          BLBS     R0, 5$
00000096          CALLG    $PDATA+^X0144(R10), G^COB$IOEXCEPTION(R9)
0000009B          BRB      6$
0000009D   5$:
0000009D          CLRB     TEST-FILE-3(R11)
000000A1   6$:
                                                                   ; 00025
000000A1          CALLG    $PDATA+^X0150(R10), G^SYS$EXIT
000000AA          MOVL     #^X01, R0
000000AD          RET

        .
        .
        .
```

ZK-6444-1-GE

2.5.5.5 Compiler Listing for a Contained Program

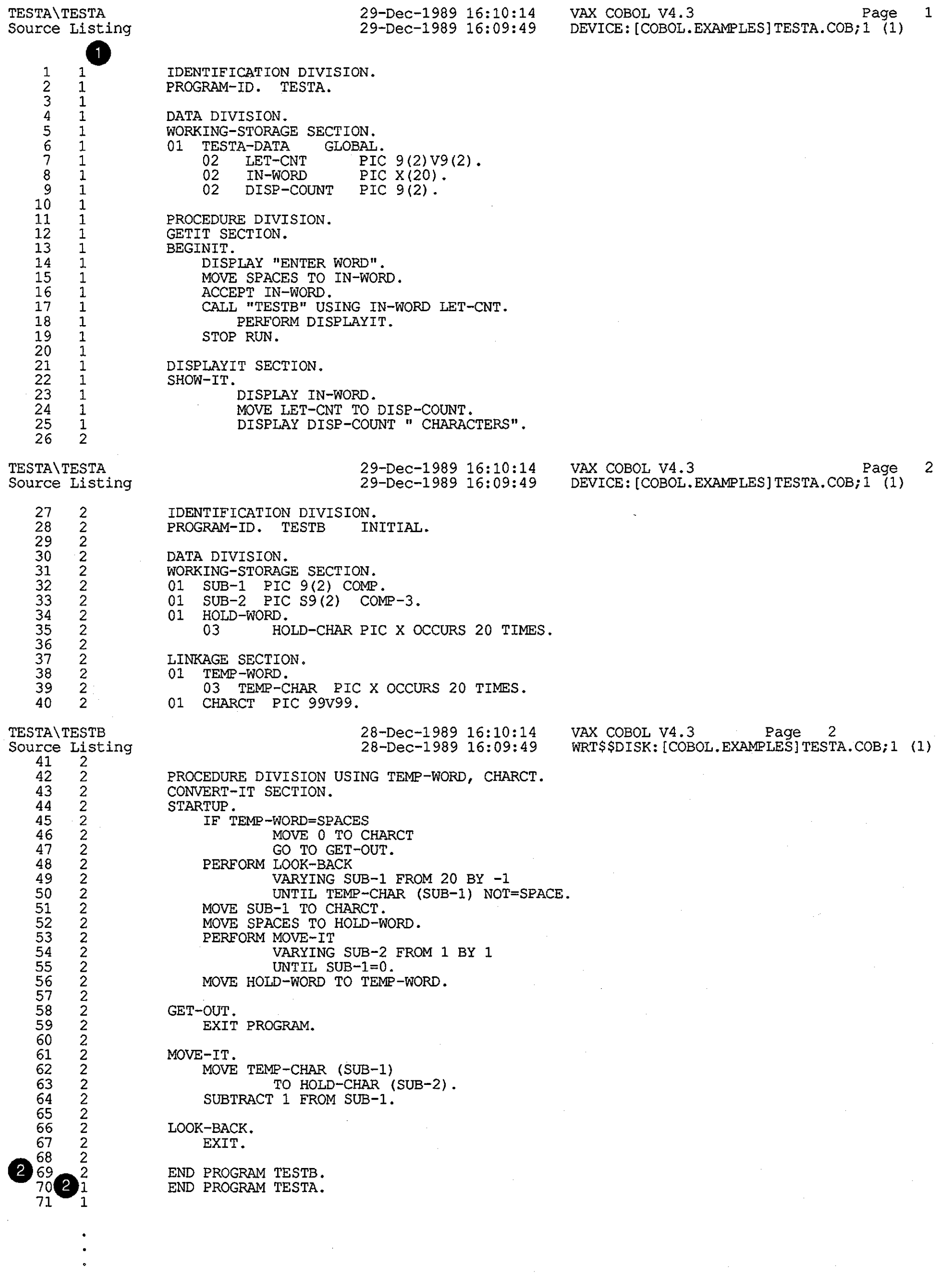
A contained COBOL program listing includes two additional program elements. For additional information on contained programs, see Chapter 18.

The circled numbers on the program listing of TESTA (see Figure 2–6) correspond to the following numbered text explanations:

❶ A number that indicates the nesting level of the program. Number one indicates the containing (main) program.

❷ The number order from greater to lesser associated with the contained program. The END PROGRAM shows the nesting level of the contained program.

Figure 2–6: VAX COBOL Listing of Contained Program



```
TESTA\TESTA                              29-Dec-1989 16:10:14   VAX COBOL V4.3               Page   1
Source Listing                           29-Dec-1989 16:09:49   DEVICE:[COBOL.EXAMPLES]TESTA.COB;1 (1)
        ❶
   1  1        IDENTIFICATION DIVISION.
   2  1        PROGRAM-ID.  TESTA.
   3  1
   4  1        DATA DIVISION.
   5  1        WORKING-STORAGE SECTION.
   6  1        01  TESTA-DATA    GLOBAL.
   7  1            02   LET-CNT       PIC 9(2)V9(2).
   8  1            02   IN-WORD       PIC X(20).
   9  1            02   DISP-COUNT    PIC 9(2).
  10  1
  11  1        PROCEDURE DIVISION.
  12  1        GETIT SECTION.
  13  1        BEGINIT.
  14  1            DISPLAY "ENTER WORD".
  15  1            MOVE SPACES TO IN-WORD.
  16  1            ACCEPT IN-WORD.
  17  1            CALL "TESTB" USING IN-WORD LET-CNT.
  18  1                PERFORM DISPLAYIT.
  19  1            STOP RUN.
  20  1
  21  1        DISPLAYIT SECTION.
  22  1        SHOW-IT.
  23  1                DISPLAY IN-WORD.
  24  1                MOVE LET-CNT TO DISP-COUNT.
  25  1                DISPLAY DISP-COUNT " CHARACTERS".
  26  2
TESTA\TESTA                              29-Dec-1989 16:10:14   VAX COBOL V4.3               Page   2
Source Listing                           29-Dec-1989 16:09:49   DEVICE:[COBOL.EXAMPLES]TESTA.COB;1 (1)

  27  2        IDENTIFICATION DIVISION.
  28  2        PROGRAM-ID.  TESTB    INITIAL.
  29  2
  30  2        DATA DIVISION.
  31  2        WORKING-STORAGE SECTION.
  32  2        01  SUB-1  PIC 9(2) COMP.
  33  2        01  SUB-2  PIC S9(2)  COMP-3.
  34  2        01  HOLD-WORD.
  35  2            03       HOLD-CHAR PIC X OCCURS 20 TIMES.
  36  2
  37  2        LINKAGE SECTION.
  38  2        01  TEMP-WORD.
  39  2            03  TEMP-CHAR  PIC X OCCURS 20 TIMES.
  40  2        01  CHARCT  PIC 99V99.
TESTA\TESTB                              28-Dec-1989 16:10:14   VAX COBOL V4.3       Page   2
Source Listing                           28-Dec-1989 16:09:49   WRT$$DISK:[COBOL.EXAMPLES]TESTA.COB;1 (1)
  41  2
  42  2        PROCEDURE DIVISION USING TEMP-WORD, CHARCT.
  43  2        CONVERT-IT SECTION.
  44  2        STARTUP.
  45  2            IF TEMP-WORD=SPACES
  46  2                    MOVE 0 TO CHARCT
  47  2                    GO TO GET-OUT.
  48  2            PERFORM LOOK-BACK
  49  2                    VARYING SUB-1 FROM 20 BY -1
  50  2                    UNTIL TEMP-CHAR (SUB-1) NOT=SPACE.
  51  2            MOVE SUB-1 TO CHARCT.
  52  2            MOVE SPACES TO HOLD-WORD.
  53  2            PERFORM MOVE-IT
  54  2                    VARYING SUB-2 FROM 1 BY 1
  55  2                    UNTIL SUB-1=0.
  56  2            MOVE HOLD-WORD TO TEMP-WORD.
  57  2
  58  2        GET-OUT.
  59  2            EXIT PROGRAM.
  60  2
  61  2        MOVE-IT.
  62  2            MOVE TEMP-CHAR (SUB-1)
  63  2                    TO HOLD-CHAR (SUB-2).
  64  2            SUBTRACT 1 FROM SUB-1.
  65  2
  66  2        LOOK-BACK.
  67  2            EXIT.
  68  2
❷ 69  2        END PROGRAM TESTB.
  70❷ 1        END PROGRAM TESTA.
  71  1
      .
      .
      .
```

ZK–6445–GE

2.6 Linking a VAX COBOL Program

Once you have compiled a VAX COBOL source program or module, link it using the DCL command LINK. The LINK command combines your object modules into one executable image the VMS operating system can execute. A source program or module cannot run on the VMS operating system until it is linked.

Unlike the VMS operating system, some systems do not have a linker. On these systems, the language compilers resolve symbolic references, and another software component completes the task while loading the program in memory. On VMS systems, however, the linker simplifies the job of each language compiler because the logic necessary to resolve symbolic references need not be duplicated. The main advantage to a system that has a linker, however, is that individual program modules can be separately written and compiled, and then linked together. This includes object modules produced by different language compilers.

When you execute the LINK command, the VMS Linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference
- Allocates virtual memory space for the executable image

When using the LINK command, you may want to use the /DEBUG qualifier. The /DEBUG qualifier appends to the image all the symbol and line number information appended to the object modules. In addition, it appends information on global symbols, and forces the image to run under debugger control when it is executed.

The LINK command produces an executable image by default. However, you can also use the LINK command to obtain shareable images and system images. The /SHAREABLE qualifier directs the linker to produce a shareable image; the /SYSTEM qualifier directs the linker to produce a system image. For more information on using shareable images refer to Section 2.6.6. For a complete discussion of the VMS Linker, refer to the VMS documentation on linking programs.

2.6.1 The LINK Command

The format of the LINK command is as follows:

```
LINK[/command-qualifier] ... {file-spec[/file-qualifier] ...} ...
```

/command-qualifier...
Specifies the output file option or options.

file-spec...
Specifies the input file or files to be linked.

/file-qualifier...
Specifies input file option or options.

If you specify more than one input file, you must separate the input file specifications with a plus sign (+) or a comma (,).

By default, the linker creates an output file with the name of the first input file specified and the file type EXE. Note that when you link multiple files, you must enter the main module ahead of called modules.

The following command line links the object files MAINPROG.OBJ, SUBPROG1.OBJ, and SUBPROG2.OBJ to produce one executable image called MAINPROG.EXE:

```
$ LINK MAINPROG.OBJ, SUBPROG1.OBJ, SUBPROG2.OBJ
```

2.6.2 LINK Command Qualifiers

The LINK command qualifiers can be used to modify the linker's output, as well as to invoke the debugging and traceback facilities. Linker output consists of an image file and an optional map file.

Table 2–3 summarizes some of the most commonly used LINK command qualifiers. A brief description of each qualifier follows this list. For a complete list of LINK qualifiers, refer to the VMS Linker documentation.

Table 2–3: Common LINK Qualifiers and Defaults

LINK Command Qualifiers	Default
/BRIEF	/BRIEF
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]EXECUTABLE	/EXECUTABLE
/FULL	/FULL
/[NO]MAP	/NOMAP (interactive)
	/MAP (batch)
/[NO]TRACEBACK	/TRACEBACK
/[NO]USERLIBRARY	/USERLIBRARY

/BRIEF

The /BRIEF qualifier produces a brief memory allocation map file that contains the following:

- A summary of image characteristics
- A list of object modules included in the image
- A summary of link-time performance statistics

Use /BRIEF only if you also specify /MAP.

Example

```
$ LINK/MAP/BRIEF PROGA
```

/[NO]CROSS_REFERENCE

The /[NO]CROSS_REFERENCE qualifier controls whether the linker produces a symbolic cross-reference on the memory allocation map. The symbolic cross-reference lists each global symbol referenced in the image, its value, and all modules in the image that refer to it. Use /CROSS_REFERENCE only if you also specify /MAP.

The default is /NOCROSS_REFERENCE.

Example

```
$ LINK/MAP/CROSS_REFERENCE PROGA
```

/[NO]DEBUG[=file-spec]

The /[NO]DEBUG qualifier controls whether the linker includes a debugger in the image.

If the object module contains local symbol table information for the debugger, specify /DEBUG to include this information in the image.

You can include the optional file specification to specify a user-defined debugger; the default file type is OBJ. If you specify /DEBUG without a file specification, the default VMS Debugger is linked to the image.

The default is /NODEBUG.

Chapter 3 discusses COBOL program debugging using the VMS Debugger. For more information on using /DEBUG, refer to the VMS Debugger documentation.

/[NO]EXECUTABLE[=file-spec]

The /[NO]EXECUTABLE qualifier controls whether the linker creates an executable image. The /EXECUTABLE qualifier can also supply a file specification for the output image file.

By default, the linker creates an executable image with the same file name as the first input file and a file type of EXE.

Use /NOEXECUTABLE to see the results of linking in less time than the linker would need to create an image file.

Examples

```
$  LINK/EXECUTABLE=NEWPROG.IMG/MAP PROGA
$  LINK/NOEXECUTABLE/MAP PROGA
```

/FULL

Produces a full memory allocation map listing that contains:

- All information contained in the brief listing
- Detailed descriptions of each program and image section in the image file
- Lists of global symbols by name and value

Use /FULL only if you also specify /MAP.

Example

```
$  LINK/MAP/NOEXEC/FULL PROGA
```

/[NO]MAP[=file-spec]

The /[NO]MAP qualifier controls whether the linker produces a memory allocation map listing.

You can provide the file specification to name the map file. Otherwise, the output file name is the same as the name of the first input file, with a file type of MAP.

When you specify /MAP, you can also specify /BRIEF, /FULL, or /CROSS_REFERENCE to control map contents. If you specify none of these qualifiers, the map contains:

- All the information contained in the brief listing
- A list of user-defined global symbols sorted by name
- A list of user-defined program sections

The interactive mode default is /NOMAP. The batch mode default is /MAP.

/[NO]TRACEBACK

The /[NO]TRACEBACK qualifier controls whether the linker includes traceback information in the image file.

By default, the linker includes traceback information so the system can trace the call stack when an error occurs.

If you specify /DEBUG, the linker also assumes /TRACEBACK.

/[NO]USERLIBRARY[=(table,...)]

The /[NO]USERLIBRARY qualifier controls whether the linker searches user-defined default libraries to resolve undefined symbols.

/USERLIBRARY causes the linker to search user-defined default libraries before it searches the system library.

Use /NOUSERLIBRARY to ignore user-defined libraries.

The default is /USERLIBRARY.

2.6.3 Positional Qualifiers

Table 2–4 lists commonly used LINK positional qualifiers. Note that there are no defaults for these qualifiers.

Table 2–4: LINK Positional Qualifiers

LINK Positional Qualifiers	Default
/INCLUDE	None
/LIBRARY	None
/OPTIONS	None

The following text summarizes the LINK positional qualifiers listed in Table 2–4 and provides a brief description of each qualifier. For a complete list and description of LINK positional qualifiers, see the VMS Linker documentation.

/INCLUDE=(module-name[,...])

The /INCLUDE qualifier indicates that the associated file specification refers to an object module library. The default file type is OLB. It also causes the linker to include only the specified modules.

You must specify at least one module name. You can specify more than one module by separating module names with commas and enclosing them in parentheses. Note that if you use a variable name in a CALL statement to refer to an external program, you must explicitly include any of the external modules that might be called by the main program when you link the program.

Using /LIBRARY (LIB) with /INCLUDE causes the linker to search the library for unresolved references after it includes the specified module.

Examples

```
$ LINK PROGA,LIBA/INCLUDE=MODA
```

The linker links PROGA.OBJ and the module MODA from the library file LIBA.OLB to produce PROGA.EXE.

```
$ LINK PROGA,LIBA/INC=(MODA,MODB)/LIB
```

The linker links PROGA.OBJ and the modules MODA and MODB from the library file LIBA.OLB. Because of the /LIBRARY qualifier, the linker will also search LIBA.OLB for any other unresolved references in PROGA.OBJ, MODA, and MODB.

/LIBRARY

The /LIBRARY qualifier indicates that the file specification refers to a library file to resolve undefined symbols in the input files.

If the file specification does not include a file type, the linker assumes OLB as the default. Do not specify a library as the first input file unless you also specify the /INCLUDE qualifier to indicate which library modules are to be included in the image.

Using /INCLUDE with /LIBRARY causes the linker to search the library for unresolved references after it includes the specified modules.

Examples

```
$ LINK PROGA,LIBA/LIBRARY
```

The linker searches LIBA.OLB for unresolved references in PROGA.OBJ to create PROGA.EXE.

```
$ LINK LIBA/LIB/INCLUDE=MOD1/EXEC=PROG
```

The linker includes the module MOD1 from LIBA.OLB, and then searches LIBA.OLB for unresolved references in MOD1. The result is an executable image, PROG.EXE.

/OPTIONS

The /OPTIONS qualifier indicates that the input file contains a list of options to control linking. If the /OPTIONS file specification does not include a file type, the linker assumes OPT as the default file type.

To link a COBOL DML object program with the shareable VAX DBMS Library, you must use an option file in the LINK command. To link a DML object program named DMLPROG.OBJ with the shareable VAX DBMS Library, you would use this command:

```
$ LINK DMLPROG, SYS$LIBRARY:DBMDML/OPT
```

The VMS Linker documentation describes the contents of the option file.

2.6.4 Using an Object Module Library

In a large development effort, programmers often store the object modules for their subprograms in an object module library. By using an object module library, you can make program modules contained in the library available to many other programmers. To link modules contained in an object module library, use the /INCLUDE qualifier and specify the specific modules you want to link. For example:

```
$ LINK GARDEN, VEGETABLES/INCLUDE=(EGGPLANT,TOMATO,BROCCOLI,ONION)
```

This example directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN.

Besides program modules, an object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which the global symbol names are defined. You specify the name of the object module library containing symbol definitions with the /LIBRARY qualifier. When you use the /LIBRARY qualifier during a link operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

In the following example, the linker uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL.

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command DEFINE. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command. See the VMS documentation on DCL for more information about the DEFINE command.

For more information about object module libraries see the VMS documentation on linking programs.

2.6.5 Object Libraries

All VAX COBOL programs reference system-supplied object module libraries when they are linked. These libraries contain routines that provide I/O and other system functions. Additionally, you can use these libraries, or your own, to provide application-specific object modules within your particular environment.

2.6.5.1 Using System-Supplied Object Module Libraries

To use the contents of an object module library, you must do the following:

- Refer to the object module by name in your program in a CALL statement, or VALUE EXTERNAL reference.
- Make sure that the linker can locate the library that contains the object module by ensuring that required software is correctly installed.
- Confirm that required logical names point to the appropriate locations. Make certain that IMAGELIB, STARLET, and VMSRTL are correctly assigned (in most cases, correct results will be obtained if they are deassigned).
- Make sure that your default directory (or LINK/OUTPUT directory) is valid and that you have write privileges to it.

To specify that a linker input file is a library file, use the /LIBRARY qualifier. This qualifier causes the linker to search for a file with the name you specify and a default file type of OLB. If you specify a file that the linker cannot locate, a fatal error occurs and the link terminates.

The sections that follow describe the order in which the linker searches libraries that you specify explicitly, default user libraries, and system libraries.

2.6.5.2 Defining the Search Order for Libraries

When you specify libraries as input for the linker, you can specify as many as you wish; there is no practical limit. More than one library can contain a definition for the same module name. The linker uses the following conventions to search libraries specified in the command string:

- A library is searched only for definitions that are unresolved in the previous input files specified.
- If more than one object module library is specified, the libraries are searched in the order in which they are specified.

For example:

```
$  LINK METRIC,DEFLIB/LIBRARY,APPLIC
```

The library DEFLIB will be searched only for unresolved references in the object module METRIC. It is not searched to resolve references in the object module APPLIC. However, this command can also be entered as follows:

```
$  LINK METRIC,APPLIC,DEFLIB/LIBRARY
```

In this case, DEFLIB.OLB is searched for all references that are not resolved between METRIC and APPLIC. After the linker has searched all libraries specified in the command, it searches default user libraries, if any, and then the default system libraries.

2.6.5.3 Default User Object Module Libraries

You can define one or more of your private object module libraries as default user libraries. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command.

To indicate that a private library is a default user library, enter a DEFINE command as in the following example:

```
$  DEFINE LNK$LIBRARY DEFLIB
```

In this example, LNK$LIBRARY is a logical name and DEFLIB is the name of an object module library (having the file type OLB) that you want the linker to search automatically in all subsequent link operations.

You can establish any object module library as a default user library by creating a logical name for the library. The logical names you must use are LNK$LIBRARY (as in the preceding example), LNK$LIBRARY_1, LNK$LIBRARY_2, and so on, to LNK$LIBRARY_999. When more than one of these logical names exists when a LINK command executes, the linker searches them in numeric order beginning with LNK$LIBRARY.

When one or more logical names exist for default user libraries, the linker uses the following search order to resolve references:

- The process, group, and then system logical name tables are searched for the name LNK$LIBRARY. If the logical name exists in any of these tables, and if it contains the desired reference, the search is ended.
- The process, then group, and then system logical name tables are searched for the name LNK$LIBRARY_1. If the logical name exists in any of these tables, and if it contains the desired reference, the search is ended.

This search sequence occurs for each reference that remains unresolved.

2.6.5.4 System Libraries

The directory identified by the system-defined logical name SYS$LIBRARY contains the following library files:

- IMAGELIB.OLB
- STARLET.OLB
- VMSRTL.EXE

IMAGELIB.OLB contains the global symbols for the shared system images.

STARLET.OLB contains, in object module form, the procedures in VMSRTL.EXE, as well as additional run-time modules required by various compilers and system programs.

The file VMSRTL.EXE contains some of the VMS Run-Time Library routines. The procedures in this library provide many useful functions including:

- Commonly used mathematical and string-handling functions
- Procedures that support code produced by VAX compilers

By default, the linker searches IMAGELIB, then STARLET, to resolve references to external names that are still unresolved after it searches libraries specified in the LINK command and default user libraries.

2.6.6 Shareable Images

You can create VAX COBOL programs to be linked and installed as shareable images. A shareable image is a single copy of a program that can be shared by many users or applications. Using shareable images provides the following benefits:

- Saves system resources, since one physical copy of a set of procedures can be shared by more than one application or user.
- Facilitates the linking of very large applications by allowing you to break down the whole application into manageable segments.
- Allows you to modify one or more sections of a large application without having to relink the entire program.

2.6.6.1 Creating a Shareable Image

A shareable image is created using the /SHARE qualifier of the LINK command.

When you create a VAX COBOL program to be installed as a shareable image, you should consider the concepts of position-dependent code and shareability. These concepts are covered in detail in the documentation on the VMS Linker.

The following list describes one way to create and install a VAX COBOL program as a shareable image:

1. Create the main program used to call the subprogram (which will be installed as a shareable image).
2. Create the subprogram to be installed as a shareable image.
3. Link the shareable image program using the /SHARE qualifier and map the entry point using either an options file or transfer vectors.

4. Copy the shareable image to the SYS$LIBRARY. (This step requires [SYSLIB] access privileges.) Alternatively you can use an assignment statement.
5. Check to see if there is enough global system space for the shareable image.
6. Install the image in a shareable library. (This step requires PRMGBL, SYSGBL, or CMKRNL privileges.)
7. Link the main program with the shareable image.

Once you have completed these steps, you can run the main program to access the subprogram installed as a shareable image.

NOTE

> VAX COBOL programs installed as shareable images cannot contain external files.
>
> When calling a subprogram installed as a shareable image, the program name specified in the CALL statement must be a literal.

More information on shareable images is available in the documentation on the VMS Linker.

The following sample programs and command procedure provide an example of how to create, link, and install a subprogram as a shareable image, as described in the preceding steps.

Example 2–1 shows the main program CALLER.COB and the two subprograms (SUBSHR1.COB and SUBSHR2.COB). Only the subprograms are installed as shareable images.

Example 2–1: Main Program and Subprograms

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
******************************************************************
* This program calls a subprogram installed as a shareable image.*
******************************************************************
PROCEDURE DIVISION.
0.
     CALL "SUBSHR1"
         ON EXCEPTION
             DISPLAY "First CALL failed. Program aborted."
     END-CALL.
     STOP RUN.
END PROGRAM CALLER.
```

(continued on next page)

Example 2–1 (Cont.): Main Program and Subprograms

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBSHR1.

******************************************************************
* This program is linked as a shareable image. When it is called,*
* it calls another program installed as a shareable image.       *
******************************************************************
PROCEDURE DIVISION.
0.
     DISPLAY "Call to SUBSHR1 successful. Calling SUBSHR2.".
     CALL "SUBSHR2"
         ON EXCEPTION
             DISPLAY "Second call failed. Control returned to CALLER."
     END-CALL.
END PROGRAM SUBSHR1.

IDENTIFICATION DIVISION.
PROGRAM-ID. SUBSHR2.
****************************************************************
* This program is linked as a shareable image and is called by *
* another shareable image.                                     *
****************************************************************
PROCEDURE DIVISION.
0.
     DISPLAY "Call to SUBSHR2 successful!".
END PROGRAM SUBSHR2.
```

Example 2–2 shows a command procedure that compiles, links, and installs the sample programs in Example 2–1.

Example 2–2: Command Procedure to Link a Program as a Shareable Image

```
$! Create the main program and subprograms to be installed as shareable
$! images. In this example CALLER.COB is the main program. SUBSHR1.COB
$! and SUBSHR2.COB are the subprograms to be installed as
$! shareable images.
$!
$! Compile the main program and subprograms.
$!
$  COBOL CALLER.COB
$  COBOL SUBSHR1.COB
$  COBOL SUBSHR2.COB
$!
$! Create an options file to map the entry points of the subprograms.
$!
$  COPY SYS$INPUT OPTIONS1.OPT
$  DECK
   UNIVERSAL=SUBSHR1,SUBSHR2
$  EOD
$!
$! Link the subprograms using the /SHARE qualifier to the shareable library
$! and the options file. For more information on options files, refer to
$! the documentation on the VMS Linker.
$!
```

(continued on next page)

Example 2–2 (Cont.): Command Procedure to Link a Program as a Shareable Image

```
$  LINK/SHARE=MYSHRLIB SUBSHR1,SUBSHR2,OPTIONS1/OPT
$!
$! Copy the shareable images to SYS$LIBRARY. To perform this
$! you must have [SYSLIB] access privileges. Alternatively,
$! you can perform the same function by doing a local assignment.
$!
$! COPY MYSHRLIB.EXE SYS$LIBRARY:*
$!      or
$  ASSIGN DEVICE:[DIRECTORY]MYSHRLIB.EXE MYSHRLIB
$!
$! Install the shareable images in a shareable library.
$! This will allow multiple users to use a single copy of the
$! shareable image.
$!
$! If you do not install the shareable library,
$! multiple users will each link to their own run-time copy of
$! the image.
$!
$! Note, to install an image in a shareable library, you must have
$! PRMGBL, SYSGBL, or CMKRNL privileges.
$!
$! Prior to installing the shareable image, check to see if there is
$! enough global symbol space.
$! MCR INSTALL
$! /GLOBAL
$! ^Z
$!
$! Also check to see if there are available global sectors and pages.
$! MCR SYS$GEN
$! /GBLSE
$! /GBLPA
$! ^Z
$!
$! The /WRITE qualifier is required if you want to install writable PSECTS.
$  MCR INSTALL
   device:[directory]MYSHRLIB/SHARE/WRITE
$!
$! Create a second options file to map the main program to the shareable
$! image library.
$  COPY SYS$INPUT OPTIONS2.OPT
$  DECK
   MYSHRLIB/SHAREABLE
$  EOD
$!
$! Link the main program with the shareable image subprograms through the
$! options file.
$  LINK CALLER,OPTIONS2/OPT
$!
$! Now you can run the main program.
```

2.6.6.2 Using Transfer Vectors

Using transfer vectors can be helpful when creating shareable images for the following reasons:

- They make it easy for you to modify the contents of shareable images.
- They allow you to avoid relinking user programs bound to the shareable image if you modify the image.

The command procedure in Example 2–3 shows how to create a transfer vector table and how to link the main program and subprograms (shown in Example 2–1) with the transfer vector table.

Example 2–3: Transfer Vectors

```
$!
$! Create a transfer vector table (TRAVEC.MAR).
$  MACRO /OBJ=TRAVEC SYS$INPUT
    .PSECT TRANSFER_VECTOR
;
;
; The transfer vector table is used to map entry points at
; run time to a shareable library. If you make changes to the
; shareable library, you only have to relink the library.
; You do not have to relink all the programs linked to the
; library.
;
; This example transfer vector table maps the entry points
; of the shareable subprograms: SUBSHR1, SUBSHR2.
;
    .TRANSFER    SUBSHR1
    .MASK        SUBSHR1
    BRW          SUBSHR1+2
    RET
    .QUAD
    .TRANSFER    SUBSHR2
    .MASK        SUBSHR2
    BRW          SUBSHR2+2
    RET
    .QUAD
;
; Note that there must be an entry point for each shareable image.
; Any future additions should be made at the end of the vector.
; The order of the entries must remain intact once established.
; Do not delete any entries (even if the shareable image is deleted).
$
$  LINK/SHARE=MYSHRLIB SUBSHR1,SUBSHR2,TRAVEC
```

Once you have created the transfer vector table, you can install the subprograms and link the main program to the shareable library as shown in Example 2–2.

For more information on transfer vectors, refer to the documentation on the VMS Linker.

2.6.7 Linker Error Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of each error. If any error or fatal error condition occurs, that is, an error with a severity E or F, the linker does not produce an image file.

The messages produced by the linker are descriptive, and you do not usually need additional information to determine the specific error. The following are some common errors that occur during linking:

- An object module has compilation errors.

  This error occurs when you attempt to link a module that generates warnings or errors during compilation. You can usually link compiled modules for which the compiler generated messages, but you should verify that the modules will actually produce the output you expect.

- The input file has a file type other than OBJ and no file type was specified.

  If you do not specify a file type, the linker assumes the file has a file type of OBJ. If the file is not an object file and you do not identify it with the appropriate file type, the linker signals an error message and does not produce an image file.

- You tried to link a nonexistent module.

  The linker signals an error message if you misspell a module name on the command line or if the compilation contains fatal diagnostics.

- A reference to a symbol name remains unresolved.

  An error occurs when you omit required module or library names from the command line and the linker cannot locate the definition for a specified global symbol reference. For example, the main program module OCEAN.OBJ calls the subprogram modules REEF.OBJ, SHELLS.OBJ, and SEAWEED.OBJ. If you specify the following LINK command, an error occurs if SEAWEED.OBJ does not exist in the same directory that the command was issued from:

  ```
  $  LINK OCEAN, REEF, SHELLS
  ```

  This example produces the following error messages:

  ```
  %LINK-W-NUDFSYMS, 1 undefined symbol
  %LINK-I-UDFSYMS,        SEAWEED
  %LINK-W-USEUNDEF, undefined symbol SEAWEED referenced
          in psect $CODE offset %X0000000C
          in module OCEAN file DEVICE$:[COBOL.EXAMPLES]PROG.OBJ;1
  %LINK-W-USEUNDEF, undefined symbol SEAWEED referenced
          in psect $CODE offset %X00000021
          in module OCEAN file DEVICES$:[COBOL.EXAMPLES]PROG.OBJ;1
  ```

  If an error occurs when you link modules, you can often correct the error by reentering the command string and specifying the correct modules or libraries. If an error indicates that a program module cannot be located, you may be linking the program with the wrong VAX COBOL Run-Time Library.

  For a complete list of linker messages, see the VMS documentation on messages.

2.7 Running a VAX COBOL Program

Once you have linked your program, you can use the DCL command RUN to execute it. The RUN command has the following format:

RUN [/[NO]DEBUG] file-spec

/[NO]DEBUG
The /[NO]DEBUG qualifier is optional. Specify the /DEBUG qualifier to request the debugger if the image was not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image was linked with the /DEBUG qualifier and you do not want the debugger to prompt, use the /NODEBUG qualifier. The default action depends on whether the file was linked with the /DEBUG qualifier.

file-spec
Is the name of the file you want to run.

The following example executes the image NOBUGS.EXE without invoking the debugger:

```
$ RUN NOBUGS/NODEBUG
```

See Chapter 3 for more information on debugging programs.

2.7.1 COBOL Run-Time Errors

During execution, an image can generate a fatal error called an exception condition. When an exception condition occurs, the system displays an error message. Run-time errors can also be issued by other facilities such as SORT or VMS.

A run-time error message has the following format:

```
%COBOL-<l>-<mnemonic>, <message>
```

%COBOL
Indicates that the COBOL run-time environment issued the error.

<l>
Indicates severity of error. The severity indicator can be one of the following:

Code	Meaning
I	Informational—Indicating information
W	Warning—Indicating a warning
E	Error—Indicating an error
F	Fatal—Indicating a severe error

<mnemonic>
A 3- to 9-character string that identifies the error.

<message>
Identifies the text of the error.

The following example shows a COBOL run-time error issued by an attempt to divide by zero:

```
%COB-E-DIVBY-ZER, divide by zero; Execution continues
```

For a description of COBOL run-time error messages, use the HELP COBOL ERRORS command.

2.8 Program Switches

Switches exist as the logical name COB$SWITCHES and can be defined for the image, process, group, or system. You can control program execution by defining switches in the SPECIAL-NAMES paragraph and setting them internally (within the image) or externally (outside the image).

2.8.1 Setting Switches Internally

To set switches internally, define them in the SPECIAL-NAMES paragraph and use the SET statement in the PROCEDURE DIVISION to specify switches ON or OFF.

For example:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SWITCH 10 IS MY-SWITCH
      ON IS SWITCH-ON
      OFF IS SWITCH-OFF.
    .
    .
    .
PROCEDURE DIVISION.
000-SET-SWITCH.
    SET MY-SWITCH TO ON.
    .
    .
    .
```

2.8.2 Setting Switches for a Process

To set switches for a process, use the DEFINE or ASSIGN DCL command to change the status of program switches:

```
$ DEFINE COB$SWITCHES "switch-list"
```

where switch-list contains up to 16 switches separated by commas. Set a switch ON by specifying it in the switch-list. A switch is OFF (the default) if you do not specify it in the switch-list.

For example:

```
$ DEFINE COB$SWITCHES "1,5,13"   Sets switches 1, 5, and 13 ON.

$ DEFINE COB$SWITCHES "9,11,16"  Sets switches 9, 11, and 16 ON.

$ DEFINE COB$SWITCHES " "        Sets all switches OFF.
```

2.8.3 Order of Evaluation

The order of evaluation for logical name assignments is image, process, group, system. System and group assignments (including COBOL program switch settings) continue until they are changed or deassigned. Process assignments continue until they are changed, deassigned, or the process ends. Image assignments end when they are changed or the image ends.

2.8.4 Checking and Controlling Switch Settings

You should know the system and group assignments for COB$SWITCHES unless you have defined them for your process or image. You can check switch settings by using this command:

```
$ SHOW LOGICAL COB$SWITCHES
```

Use the DEASSIGN command to remove the switch-setting logical name from your process and reactivate the group or system logical name (if any):

```
$ DEASSIGN COB$SWITCHES
```

To change the status of external switches during execution, do the following:

1. Interrupt the image with a STOP literal COBOL statement.
2. Use a DEFINE command to change switch settings.
3. Continue execution with the CONTINUE command. Be sure not to force the interrupted image to exit by entering a command that executes another image.

2.8.5 Example Using Program Switches

Example 2–4 shows how to use program switches.

Example 2–4: Using Program Switches

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SWITCH 1 ON IS DAILY-DATA
    SWITCH 3 ON IS WEEKLY-DATA
    SWITCH 5 ON IS MONTHLY-DATA
    SWITCH 10 IS MY-SWITCH
      ON IS SWITCH-ON
      OFF IS SWITCH-OFF.
    .
    .
    .
PROCEDURE DIVISION.
000-CHECK-SWITCHES.
    SET MY-SWITCH TO ON.
    IF DAILY-DATA PERFORM 100-DAILY-ROUTINE.
    IF WEEKLY-DATA PERFORM 200-WEEKLY-ROUTINE.
    IF MONTHLY-DATA PERFORM 300-MONTHLY-ROUTINE.
    IF NOT DAILY-DATA AND
       NOT WEEKLY-DATA AND
       NOT MONTHLY-DATA
       PERFORM 400-ANNUAL-ROUTINE.
    IF END-OF-YEAR="Y"
       SET MY-SWITCH TO OFF.
    .
    .
    .
```

If you use this program to process only weekly and monthly data, your DEFINE command would be:

```
$ DEFINE COB$SWITCHES "3,5"
```

Chapter 3

Using the VMS Debugger

This chapter is an introduction to using the VMS Debugger with VAX COBOL programs. It includes the following information:

- An overview of debugger concepts
- Enough information so that you can start using the debugger
- A summary of the debugger commands by function
- A sample terminal session that demonstrates using the debugger to find a bug in a VAX COBOL program

For complete reference information on the VMS Debugger, see the VMS documentation. Online help is available during debugging sessions.

3.1 VMS Debugger Concepts

A debugger is a tool to help you locate run-time errors quickly. It is used with a program already compiled and linked successfully, with no errors reported, that does not run correctly; for example, the program output is obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively, step by step, until you locate the point at which the program stopped working correctly.

The VMS Debugger is a *symbolic* debugger. This means you can refer to program locations by the symbols (names) you used for those locations in your program. You can use the names of variables, paragraphs, sections, and so on. You do not have to use virtual addresses to refer to memory locations.

The debugger recognizes the syntax, expressions, data typing, and other constructs of VAX COBOL, as well as the following other VMS-supported languages:

VAX Ada®
VAX BASIC
VAX BLISS
VAX C
VAX DIBOL
VAX FORTRAN
VAX MACRO-32
VAX PASCAL
VAX PL/I

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

VAX RPG II
VAX SCAN

Therefore, if your program is written in more than one language, you can change from one language to another during a debugging session. The current source language determines the format used for entering and displaying data. It also determines the format for other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

By issuing debugger commands at your terminal, you can do the following:

- Start, stop, and resume the program's execution
- Trace the execution path of the program
- Monitor selected locations, variables, or events
- Examine and modify the contents of variables, or force events to occur
- Test the effect of some program modifications without editing, recompiling, and relinking the program

Such techniques enable you to isolate an error in your code more quickly than you can without the debugger.

Once you have found the error in the program, you can edit the source code and compile, link, and run the corrected version.

3.2 Features of the Debugger

The VMS Debugger provides the following features that help you to debug your programs:

- Online help—You can access help during debug sessions.
- Source code display—You can display source code during debug sessions.
- Screen mode—You can display and capture information in scrollable windows.
- Keypad mode—You can issue commonly used debugger command sequences with VT100, VT52, or LK201 keypads.
- Source editing—You can edit your source code while in a debug session.
- Command procedures—You can issue debug commands from command procedures.
- Symbol definitions—You can define your own symbols to represent commands, address expressions, or values in abbreviated form.
- Initialization files—You can create an initialization file containing commands to tailor your debug session.
- Log files—You can record your debug session to a log file.

3.3 Getting Started with the Debugger

This section explains how to use the debugger and provides VAX COBOL examples. The intent is to enable you to start using the debugger; therefore, only basic functions are covered. For more detailed information, see the VMS documentation on the debugger. Remember that online help is immediately available to you during a debugging session when you type the HELP command at the debugger prompt (DBG>).

3.3.1 Compiling and Linking to Prepare for Debugging

The following example shows how to compile and link a VAX COBOL program consisting of a single compilation unit named INVENTORY.

```
$ COBOL/DEBUG INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the COBOL command causes the compiler to write the debug symbol records associated with INVENTORY into the object module, INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY in debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the /DEBUG qualifier.)

The /DEBUG qualifier on the LINK command causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. The qualifier also causes the VMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify other modules in the LINK command.)

3.3.1.1 Establishing the Debugging Configuration

Before invoking the debugger (as explained in Section 3.3.2), check that the debugging configuration is appropriate for the kind of program you are going to debug.

You can invoke the debugger in either the *default* configuration or the *multiprocess* configuration to debug programs that run in either one or several processes, respectively. The configuration depends on the current definition of the logical name DBG$PROCESS. Thus, before invoking the debugger, enter the DCL command SHOW LOGICAL DBG$PROCESS to determine the definition of DBG$PROCESS.

For programs that run in only one process, DBG$PROCESS either should be undefined, as in the following example, or should have the value DEFAULT:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

If DBG$PROCESS has the value MULTIPROCESS, and you want to debug a program that runs in only one process, enter the following command:

```
$ DEFINE DBG$PROCESS DEFAULT
```

3.3.2 Starting and Ending a Debugging Session

To invoke the debugger, issue the DCL command RUN. The following message will appear on your screen.

```
$ RUN INVENTORY
                     VAX DEBUG Version 5.n-nn

%DEBUG-I-INITIAL, language is COBOL, module set to 'INVENTORY'
DBG>
```

The DBG> prompt indicates that you can type debugger commands. At this point, if you type the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input or an error occurs).

If your program goes into an infinite loop during a debugging session so that the debugger prompt does not reappear, press CTRL/C. This interrupts program execution and returns you to the debugger prompt (pressing CTRL/C does not end the debugging session). For example:

```
DBG> GO
       .
       .
       .
CTRL/C
DBG>
```

You can also press CTRL/C to abort the execution of a debugger command. This is useful if a command takes a long time to complete.

If your program already has a CTRL/C AST service routine enabled, use the SET ABORT_KEY command to assign the debugger's abort function to another CTRL—key sequence.

Pressing CTRL/Y from within a debugging session has the same effect as pressing CTRL/Y during the execution of a program. Control is returned to the DCL command interpreter ($ prompt).

The following message indicates that your program has completed successfully:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

To end a debugging session, type the EXIT command at the DBG> prompt as follows, or press CTRL/Z.

```
DBG> EXIT
$
```

3.3.3 Issuing Debugger Commands

You can issue debugger commands any time you see the DBG> prompt. The debugger commands are summarized in Table 3–1.

To issue a command, type it at the keyboard and press the RETURN key. You can issue several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the line with a hyphen (-).

Alternatively, you can use the numeric keypad to issue certain commands. In addition to the STEP, GO, SHOW CALLS, and EXAMINE commands, several functions that manipulate screen-mode displays are bound to the keys. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. (The PF1 key is commonly known as the GOLD key, and the PF4 key is commonly known as the BLUE key.) To obtain a key's DEFAULT function, press the key. To obtain its GOLD function, first press the PF1 (GOLD) key, and then the key. To obtain its BLUE function, first press the PF4 (BLUE) key, and then the key.

For more information on the debug keypad commands, you can type HELP KEYPAD, or refer to the VMS documentation on the debugger.

Table 3–1 lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

Table 3–1: Debugger Command Summary

Command	Description
	Starting and Ending a Debugging Session
RUN[1]	Invokes the debugger if LINK/DEBUG was used.
RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed.
CTRL/Z or EXIT	Ends a debugging session, executing all exit handlers.
QUIT	Ends a debugging session without executing any exit handlers declared in the program.
CTRL/C	Aborts program execution or a debugger command without interrupting the debugging session.
{ SET SHOW } ABORT_KEY	Assigns the default CTRL/C abort function to another CTRL—key sequence; identifies the CTRL—key sequence currently defined for the abort function.
CTRL/Y—DEBUG[1]	Interrupts a program that is running without debugger control and invokes the debugger.
ATTACH	Passes control of your terminal from the current process to another process.
SPAWN	Creates a subprocess, enabling you to execute DCL commands without ending a debugging session or losing your debugging context.
	Controlling and Monitoring Program Execution
GO	Starts or resumes program execution.
STEP	Executes the program up to the next line, instruction, or specified instruction.
{ SET SHOW } STEP	Establishes or displays the default qualifiers for the STEP command.
{ SET SHOW CANCEL } BREAK	Sets, displays, or cancels breakpoints.
{ SET SHOW CANCEL } TRACE	Sets, displays, or cancels tracepoints.
{ SET SHOW CANCEL } WATCH	Sets, displays, or cancels watchpoints.
SHOW CALLS	Identifies the currently active routine calls.
SHOW STACK	Gives additional information about the currently active routine calls.
CALL	Calls a routine.

[1] This is a DCL command, not a debugger command.

(continued on next page)

Table 3–1 (Cont.): Debugger Command Summary

Command	Description
Examining and Manipulating Data	
EXAMINE	Displays the value of a variable or the contents of a program location.
SET MODE [NO]OPERANDS	Controls whether the address and contents of the instruction operands are displayed when you examine an instruction.
DEPOSIT	Changes the value of a variable or the contents of a program location.
EVALUATE	Evaluates a language or address expression.
Controlling Type Selection	
{SET SHOW CANCEL} RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix.
{SET SHOW CANCEL} TYPE	Establishes the type for program locations that are not associated with a compiler-generated type, displays the type, or restores the type.
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT.
Controlling Symbol Lookup and Symbolization	
SHOW SYMBOL	Displays symbols in your program.
{SET SHOW CANCEL} MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module.
{SET SHOW CANCEL} IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image.
SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are set automatically when the debugger interrupts execution.
{SET SHOW CANCEL} SCOPE	Establishes, displays, or restores the scope for symbol lookup.
SYMBOLIZE	Converts a virtual address to a symbolic address.
SET MODE [NO]LINE	Controls whether program locations are displayed in terms of line numbers or routine-name + byte offset.
SET MODE [NO]SYMBOLIC	Controls whether program locations are displayed symbolically or in terms of numeric addresses.
Displaying Source Code	
TYPE	Displays lines of source code.
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression.
SEARCH	Searches the source code for the specified string.

(continued on next page)

Table 3–1 (Cont.): Debugger Command Summary

Command	Description
Displaying Source Code	
{ SET SHOW } SEARCH	Establishes or displays the default qualifiers for the SEARCH command.
SET STEP [NO]SOURCE	Enables or disables the display of source code after a STEP command has been executed or at a breakpoint, tracepoint, or watchpoint.
{ SET SHOW } MARGINS	Establishes or displays the left and right margin settings for displaying source code.
{ SET SHOW CANCEL } SOURCE	Creates, displays, or cancels a source directory search list.
{ SET SHOW } MAX_SOURCE_ FILES	Establishes or displays the maximum number of source files that can be kept open at one time.
Using Screen Mode	
SET MODE [NO]SCREEN	Enables or disables screen mode.
DISPLAY	Creates or modifies a display.
SCROLL	Scrolls a display.
EXPAND	Expands or contracts a display.
MOVE	Moves a display across the screen.
{ SHOW CANCEL } DISPLAY	Identifies or deletes a display.
{ SET SHOW CANCEL } WINDOW	Creates, identifies, or deletes a window definition.
SELECT	Selects a display for a display attribute.
SHOW SELECT	Identifies the displays selected for each of the display attributes.
SAVE	Saves the current contents of a display and writes it to another display.
EXTRACT	Saves a display or the current screen state and writes it to a file.
{ SET SHOW } TERMINAL	Establishes or displays the height and width of the screen.
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command.
CTRL/W or DISPLAY/REFRESH	Refreshes the screen.
Editing Source Code	
EDIT	Invokes an editor during a debugging session.
{ SET SHOW } EDITOR	Establishes or identifies the editor invoked by the EDIT command.

(continued on next page)

Table 3–1 (Cont.): Debugger Command Summary

Command	Description
Defining Symbols	
DEFINE	Defines a symbol as an address, command, or value.
DELETE	Deletes symbol definitions.
{ SET SHOW } DEFINE	Establishes or displays the default qualifier for the DEFINE command.
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined with the DEFINE command.
Using Keypad Mode	
SET MODE [NO]KEYPAD	Enables or disables keypad mode.
DEFINE/KEY	Creates key definitions.
DELETE/KEY	Deletes key definitions.
SET KEY	Establishes the key definition state.
SHOW KEY	Displays key definitions.
Using Command Procedures and Log Files	
@file-spec	Executes a command procedure.
{ SET SHOW } ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures.
DECLARE	Defines parameters to be passed to command procedures.
{ SET SHOW } LOG	Specifies or identifies the debugger log file.
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged.
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated.
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed.
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command.
Using Control Structures	
FOR	Executes a list of commands while incrementing a variable.
IF	Executes a list of commands conditionally.
REPEAT	Executes a list of commands a specified number of times.
WHILE	Executes a list of commands while a condition is true.
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop.

(continued on next page)

Table 3–1 (Cont.): Debugger Command Summary

Command	Description
Debugging Multiprocess Programs	
CONNECT	Brings a process under debugger control.
DEFINE/PROCESS_GROUP	Assigns a symbolic name to a list of process specifications.
DO	Executes commands in the context of one or more processes.
SET MODE [NO]INTERRUPT	Controls whether execution is interrupted in other processes when it is suspended in some process.
{ SET SHOW } PROCESS	Modifies the multiprocess debugging environment, or displays process information.
Additional Commands	
{ DISABLE ENABLE SHOW } AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled.
{ SET SHOW } EVENT_ FACILITY	Establishes or identifies the current run-time facility for language-specific events.
{ SET SHOW } LANGUAGE	Establishes or displays the current language.
SET MODE [NO]SEPARATE	Controls whether the debugger, when used on a workstation running VWS, creates a separate window for debugger input and output
SET OUTPUT [NO]TERMINAL	Controls whether debugger output, except for diagnostic messages, is displayed or suppressed.
SET PROMPT	Specifies the debugger prompt.
{ SET SHOW } TASK	Modifies the tasking environment or displays task information.
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program.
SHOW MODE	Identifies the current debugger modes established by the SET MODE command (for example, screen mode, step mode)
SHOW OUTPUT	Identifies the current output options established by the SET OUTPUT command

3.4 Notes on VAX COBOL Support

In general, the VMS Debugger supports the data types and operators of VAX COBOL and other debugger-supported languages. However, there are important language-specific limitations. (To get information about the supported data types and operators for any of the languages, type the HELP LANGUAGE command at the DBG> prompt.)

The debugger can show source text included in a program with the COPY REPLACING or REPLACE statement. However, the debugger always shows the original source text instead of the modified source text generated by the COPY REPLACING or REPLACE statement.

The debugger cannot show the original source lines associated with the code for a REPORT section. You can see the DATA SECTION source lines associated with a report, but no source lines are associated with the compiled code that generates the report.

3.5 Sample Debugging Session

This section provides a sample debugging session that demonstrates many of the debugger features.

As you read the debugging section that follows, refer to the code in Example 3–1 to identify source lines. The program, TESTA, accepts a character string from the terminal and passes it to contained program TESTB. TESTB reverses the character string and returns it (and its length) to TESTA.

Example 3–1: Source Code Used in the Sample Debug Session

```
TESTA\TESTA
Source Listing
    1   1       IDENTIFICATION DIVISION.
    2   1       PROGRAM-ID.  TESTA.
    3   1       DATA DIVISION.
    4   1       WORKING-STORAGE SECTION.
    5   1       01  TESTA-DATA        GLOBAL.
    6   1           02   LET-CNT      PIC 9(2)V9(2).
    7   1           02   IN-WORD      PIC X(20).
    8   1           02   DISP-COUNT   PIC 9(2).
    9   1       PROCEDURE DIVISION.
   10   1       GETIT SECTION.
   11   1       BEGINIT.
   12   1           DISPLAY "ENTER WORD".
   13   1           MOVE SPACES TO IN-WORD.
   14   1           ACCEPT IN-WORD.
   15   1           CALL "TESTB" USING IN-WORD LET-CNT.
   16   1               PERFORM DISPLAYIT.
   17   1           STOP RUN.
   18   1       DISPLAYIT SECTION.
   19   1       SHOW-IT.
   20   1               DISPLAY IN-WORD.
   21   1               MOVE LET-CNT TO DISP-COUNT.
   22   1               DISPLAY DISP-COUNT " CHARACTERS".
   23   2       IDENTIFICATION DIVISION.
   24   2       PROGRAM-ID.  TESTB    INITIAL.
   25   2       DATA DIVISION.
   26   2       WORKING-STORAGE SECTION.
   27   2       01  SUB-1  PIC 9(2) COMP.
   28   2       01  SUB-2  PIC S9(2)  COMP-3.
   29   2       01  HOLD-WORD.
   30   2           03      HOLD-CHAR PIC X OCCURS 20 TIMES.
   31   2       LINKAGE SECTION.
   32   2       01  TEMP-WORD.
   33   2           03  TEMP-CHAR  PIC X OCCURS 20 TIMES.
   34   2       01  CHARCT  PIC 99V99.
   35   2       PROCEDURE DIVISION USING TEMP-WORD, CHARCT.
   36   2       CONVERT-IT SECTION.
   37   2       STARTUP.
   38   2           IF TEMP-WORD = SPACES
   39   2                   MOVE 0 TO CHARCT
   40   2                   GO TO GET-OUT.
   41   2           PERFORM LOOK-BACK
```

(continued on next page)

Example 3–1 (Cont.): Source Code Used in the Sample Debug Session

```
 42   2                 VARYING SUB-1 FROM 20 BY -1
 43   2                 UNTIL TEMP-CHAR (SUB-1) NOT = SPACE.
 44   2         MOVE SUB-1 TO CHARCT.
 45   2         MOVE SPACES TO HOLD-WORD.
 46   2         PERFORM MOVE-IT
 47   2                 VARYING SUB-2 FROM 1 BY 1
 48   2                 UNTIL SUB-1 = 0.
 49   2         MOVE HOLD-WORD TO TEMP-WORD.
 50   2     GET-OUT.
 51   2         EXIT PROGRAM.
 52   2     MOVE-IT.
 53   2         MOVE TEMP-CHAR (SUB-1)
 54   2                 TO HOLD-CHAR (SUB-2).
 55   2         SUBTRACT 1 FROM SUB-1.
 56   2     LOOK-BACK.
 57   2         EXIT.
 58   2     END PROGRAM TESTB.
 59   1     END PROGRAM TESTA.
 60   1
```

The following debugging session does not show the location of program errors; it is designed to show only the use of debugger features.

1. The RUN command starts the session. If you compile and link the program with /DEBUG, you do not need to use the /DEBUG qualifier in the RUN command.

   When you give the RUN command, the debugger displays its standard header, showing that the default language is COBOL and the default scope and module are your main program. The debugger returns control with the prompt, DBG>.

   ```
   $  RUN TESTA
                VAX DEBUG Version 5.n-nn
   %DEBUG-I-INITIAL, language is COBOL, module set to 'TESTA'
   %DEBUG-I-NOTATMAIN, type GO to get to start of main program
   DBG>
   ```

2. Use the GO command to get to the start of the main program.

   ```
   DBG>  GO
   ```

3. Set a breakpoint.

   ```
   DBG>  SET BREAK %LINE 41
   ```

4. Begin execution with the GO command. The debugger displays the execution starting point, and the image continues until TESTA displays its prompt and waits for a response.

   ```
   DBG>  GO
   ENTER WORD
   ```

5. Enter the word to be reversed. Execution continues until the image reaches the breakpoint at line 41 of the contained program.

   ```
   backward
   break at TESTA\TESTB\CONVERT-IT\STARTUP\%LINE 41
       41:          PERFORM LOOK-BACK
   ```

6. Set two breakpoints. When the debugger reaches line 55 of TESTB, it executes the commands in parentheses, displays the two data items, and resumes execution.

   ```
   DBG> SET BREAK %LINE 55 DO (EXAMINE HOLD-WORD;EXAMINE SUB-1;GO)
   DBG> SET BREAK %LINE 49
   ```

7. Display the active breakpoints.

   ```
   DBG> SHOW BREAK
   breakpoint at TESTA\TESTB\CONVERT-IT\STARTUP\%LINE 41
   breakpoint at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
      do (EXAMINE HOLD-WORD;EXAMINE SUB-1;GO)
   breakpoint at TESTA\TESTB\CONVERT-IT\STARTUP\%LINE 49
   ```

8. Use the TYPE command to display the source lines where you set breakpoints.

   ```
   DBG> TYPE 41:55
   module TESTA
       41:             PERFORM LOOK-BACK
       42:                     VARYING SUB-1 FROM 20 BY -1
       43:                     UNTIL TEMP-CHAR (SUB-1) NOT = SPACE.
       44:             MOVE SUB-1 TO CHARCT.
       45:             MOVE SPACES TO HOLD-WORD.
       46:             PERFORM MOVE-IT
       47:                     VARYING SUB-2 FROM 1 BY 1
       48:                     UNTIL SUB-1 = 0.
       49:             MOVE HOLD-WORD TO TEMP-WORD.
       50: GET-OUT.
       51:             EXIT PROGRAM.
       52: MOVE-IT.
       53:             MOVE TEMP-CHAR (SUB-1)
       54:                     TO HOLD-CHAR (SUB-2).
       55:             SUBTRACT 1 FROM SUB-1.
   ```

9. Set a tracepoint at line 15 of TESTA.

   ```
   DBG> SET TRACE %LINE 15
   ```

10. Set a watchpoint on the data item DISP-COUNT. When an instruction tries to change the contents of DISP-COUNT, the debugger returns control to you.

    ```
    DBG> SET WATCH DISP-COUNT
    ```

11. Execution resumes with the GO command. Before line 55 in TESTB executes, the debugger executes the contents of the DO command entered at step 7. It displays the contents of HOLD-WORD and SUB-1, then resumes execution.

```
DBG> GO
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "d                    "
TESTA\TESTB\SUB-1:         8
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "dr                   "
TESTA\TESTB\SUB-1:         7
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "dra                  "
TESTA\TESTB\SUB-1:         6
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "draw                 "
TESTA\TESTB\SUB-1:         5
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "drawk                "
TESTA\TESTB\SUB-1:         4
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "drawkc               "
TESTA\TESTB\SUB-1:         3
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "drawkca              "
TESTA\TESTB\SUB-1:         2
break at TESTA\TESTB\CONVERT-IT\MOVE-IT\%LINE 55
    55:        SUBTRACT 1 FROM SUB-1.
TESTA\TESTB\HOLD-WORD:
    HOLD-CHAR(1:20):            "drawkcab             "
TESTA\TESTB\SUB-1:         1
break at TESTA\TESTB\CONVERT-IT\STARTUP\%LINE 49
    49:       MOVE HOLD-WORD TO TEMP-WORD.
```

12. Deposit the value 10 into data item SUB-1. Notice that SUB-1's usage is COMP.

```
DBG> DEPOSIT SUB-1=10
```

13. Examine the contents of SUB-1.

```
DBG> EXAMINE SUB-1
TESTA\TESTB\SUB-1:    10
```

14. Deposit –42 into data item SUB-2. Notice that SUB-2's usage is COMP-3.

```
DBG> DEPOSIT SUB-2=-42
```

15. SUB-2's contents are now –42.

```
DBG> EXAMINE SUB-2
TESTA\TESTB\SUB-2:     -42
```

16. Examine CHARCT, whose picture is 99V99.

```
DBG> EXAMINE CHARCT
TESTA\TESTB\CHARCT:     8.00
```

17. Deposit four characters into CHARCT.

```
DBG>  DEPOSIT CHARCT=15.00
```

18. CHARCT now contains 15.00.

```
DBG>  EXAMINE CHARCT
TESTA\TESTB\CHARCT:     15.00
```

19. Deposit an integer larger than CHARCT's definition. The debugger returns an error message.

```
DBG>  DEPOSIT CHARCT=2890
%DEBUG-E-DECOVF, decimal overflow at or near DEPOSIT
```

20. Examine the contents of CHARCT.

```
DBG>  EXAMINE CHARCT
TESTA\TESTB\CHARCT:     90.00
```

21. You can examine any character of a subscripted data item by specifying the character position. The following EXAMINE command accesses the fourth character on TEMP-CHAR.

```
DBG>  EXAMINE TEMP-CHAR(4)
TEMP-CHAR of TESTA\TESTB\TEMP-WORD(4):  "k"
```

22. You can use the DEPOSIT command to put a value into any element of a table and examine its contents. In this example, "X" is deposited into the fourth character position of TEMP-CHAR.

```
DBG>  DEPOSIT TEMP-CHAR(4)="X"
DBG>  EXAMINE TEMP-CHAR(4)
TEMP-CHAR of TESTA\TESTB\TEMP-WORD(4):  "X"
```

NOTE

> You can qualify data names in debug commands as you can in COBOL. For example, if you examine IN-WORD while you debug your program, you can use the following DEBUG command:
>
> ```
> EXAMINE IN-WORD OF TESTA-DATA
> ```

23. Deposit a value into CHARCT.

```
DBG>  DEPOSIT CHARCT=8.00
```

24. Resume execution with the GO command. The program TESTA displays the reversed word. When the image reaches line 21 in TESTA, the debugger detects that an instruction changed the contents of DISP-COUNT. Since you set a watchpoint on DISP-COUNT, the debugger displays the old and new values, then returns control to you.

```
DBG>  GO
drawkcab
watch of DISP-COUNT of TESTA\TESTA-DATA at TESTA\DISPLAY-IT
  \SHOW-IT\%LINE 21
    21:        MOVE LET-CNT TO DISP-COUNT.
        old value =  0
        new value =  8
break at TESTA\DISPLAY-IT\SHOW-IT\%LINE 22
   22:  DISPLAY DISP-COUNT " CHARACTERS".
```

25. To see the image's current location, use the SHOW CALLS command.

```
DBG>  SHOW CALLS
module name     routine name             line     rel PC    abs PC
*TESTA           TESTA                    22     00000056  00000656
                 LIB$AB_CVTPT_U                  00000154  00000C58
```

26. Resume execution with the GO command. TESTA executes its final display. The debugger regains control when STOP RUN executes.

    ```
    DBG> GO
    08 CHARACTERS
    %DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL,
      normal successful completion'
    ```

27. At this point, you can either examine the contents of data items or end the session with the EXIT command.

    ```
    DBG> EXIT
    $
    ```

Part II
Using VAX COBOL Features on VMS

Chapter 4

Numeric Data Handling

This chapter describes how VAX COBOL stores, represents, moves, and manipulates numeric data.

4.1 How the Compiler Stores Numeric Data

Understanding how data is stored is particularly important when you define data items to participate in group moves or to be the subject of a REDEFINES clause. When moving a complex record consisting of several levels of subordination, you should be sure that the receiving item is large enough to prevent data truncation. You can also use data storage concepts to minimize storage space, particularly when the data file is large. The storage considerations applicable to table handling are discussed in Chapter 6.

For each numeric data item, VAX COBOL stores the numeric value, a scaling factor (if a V or a P appears in the PICTURE), and a sign (if an S appears in the PICTURE). Each of these subjects is discussed separately in the following sections.

The USAGE clause of a numeric data item specifies the data's internal format in storage. When you do not specify a usage in a PICTURE clause, the default usage is DISPLAY. For further information on internal representations see the USAGE clause tables in the *VAX COBOL Reference Manual*.

All records, and elementary items with level 01 or 77, begin at an address that is a multiple of 4 bytes (a longword boundary). The VAX COBOL compiler tries to locate a data item at the next unassigned byte location. However, some data items must be aligned on a 2-, 4-, or 8-byte boundary.

4.2 Sign Conventions

VAX COBOL numeric items can be signed or unsigned. However, all VAX COBOL arithmetic operations yield signed results. If you store a signed result in an unsigned item, only the absolute value is stored. Thus, unsigned items only contain the value zero or a positive value. The way VAX COBOL stores signed results in signed data items depends on the usage and the presence or absence of the SIGN clause.

Do not use unsigned numeric items in arithmetic operations. They usually cause programming errors and are handled less efficiently than signed numeric items. The following example shows how unsigned numeric items can cause errors.

```
DATA DIVISION
.
.
.
01 A PIC 9(5) COMP VALUE 2.
01 B PIC 9(5) COMP VALUE 5.
```

Then:

```
SUBTRACT B FROM A.        (A = 3)

SUBTRACT 1 FROM A.        (A = 2)
```

However:

```
COMPUTE A = (A - B) - 1     (A = 4)
```

The absence of signs for the numeric items A and B results in two different answers after parallel arithmetic operations have been done. This occurs because internal temporaries (required by the COMPUTE statement) are signed. Thus, the result of (A–B) within the COMPUTE statement is –3; –3 and –1 is –4 and the value of A then becomes 4.

4.3 Invalid Values in Numeric Items

All VAX COBOL arithmetic operations store valid values in their result items. However, it is possible to store data in numeric items that do not conform to the data definitions of those items. For example, you can place signed values into unsigned items and place nonnumeric or improperly signed data into signed numeric display items. This can happen when you use invalid input data or redefined items or perform group moves.

The results of arithmetic operations that use invalid data in numeric items are undefined.

4.4 Evaluating Numeric Items

VAX COBOL provides several kinds of conditional expressions used for evaluating numeric items. These conditional expressions include the following:

- The numeric relation condition that compares the item's contents to another numeric value
- The sign condition that examines the item's sign to see if it is positive or negative
- The class condition that inspects the item's digit positions for valid numeric characters
- The success/failure condition that checks the return status codes of COBOL and non-COBOL procedures for success or failure conditions

The following sections explain these conditional expressions in detail.

4.4.1 Numeric Relation Tests

A numeric relation test compares two numeric quantities and determines if the specified relation between them is true. For example, the following statement compares item FIELD1 to item FIELD2 and determines if the numeric value of FIELD1 is greater than the numeric value of FIELD2.

```
IF FIELD1 > FIELD2 ...
```

If the relation condition is true, the program control takes the true path of the statement.

Table 4–1 describes the relational operators.

Table 4–1: Numeric Relational Operator Descriptions

Operator	Description
IS [NOT] GREATER THAN IS [NOT] >	The first operand is greater than (or not greater than) the second operand.
IS [NOT] LESS THAN IS [NOT] <	The first operand is less than (or not less than) the second operand.
IS [NOT] EQUAL TO IS [NOT] =	The first operand is equal to (or not equal to) the second operand.
IS GREATER THAN OR EQUAL TO IS >=	The first operand is greater than or equal to the second operand.
IS LESS THAN OR EQUAL TO IS <=	The first operand is less than or equal to the second operand.

Comparison of two numeric operands is valid regardless of their USAGE clauses.

The length of the literal or arithmetic expression operands (in terms of the number of digits represented) is not significant. Zero is a unique value, regardless of the sign.

Unsigned numeric operands are assumed to be positive for comparison. The results of relation tests involving invalid (nonnumeric) data in a numeric item are undefined.

4.4.2 Numeric Sign Tests

The sign test compares a numeric quantity to zero and determines if it is greater than (positive), less than (negative), or equal to zero. Both the relation test and the sign test can perform this function. For example, consider the following relation test:

```
IF FIELD1 > 0 ...
```

Now consider the following sign test:

```
IF FIELD1 POSITIVE ...
```

Both of these tests accomplish the same thing and always arrive at the same result. The sign test, however, shortens the statement and makes it more obvious that the sign is being tested.

If the item being tested contains a sign (whether carried as an overpunched character or as a separate character), the test checks it for a valid sign value. If the character position carrying the sign contains an invalid sign value, the NUMERIC class test rejects the item, and program control takes the false path of the IF statement.

Table 4–2 shows the sign tests and their equivalent relation tests.

Table 4–2: Sign Tests

Sign Test	Equivalent Relation Test
IF FIELD1 POSITIVE ...	IF FIELD1 > 0 ...
IF FIELD1 NOT POSITIVE ...	IF FIELD1 NOT > 0 ...
IF FIELD1 NEGATIVE ...	IF FIELD1 < 0 ...
IF FIELD1 NOT NEGATIVE ...	IF FIELD1 NOT < 0 ...
IF FIELD1 ZERO ...	IF FIELD1 = 0 ...
IF FIELD1 NOT ZERO ...	IF FIELD1 NOT = 0 ...

Sign tests do not execute faster or slower than relation tests because the compiler substitutes the equivalent relation test for every correctly written sign test.

4.4.3 Numeric Class Tests

The class test inspects an item to determine if it contains numeric or alphabetic data. For example, the following statement determines if FIELD1 contains numeric data:

```
IF FIELD1 IS NUMERIC ...
```

If the item is numeric, the test condition is true, and program control takes the true path of the statement.

Both relation and sign tests determine only if an item's contents are within a certain range. Therefore, certain items in newly prepared data can pass both the relation and sign tests and still contain data preparation errors.

The NUMERIC class test checks alphanumeric or numeric DISPLAY or COMP-3 usage items for valid numeric digits.

The ALPHABETIC class test check is not valid for an operand described as numeric.

4.4.4 Success/Failure Tests

The success/failure condition tests the return status codes of COBOL and non-COBOL procedures for success or failure conditions.

You can use the SET statement to initialize or alter the status of status-code-id, which must be a word or longword COMP integer.

The SUCCESS class condition is true if you specify that the status-code-id IS SUCCESS. The FAILURE class condition is true if you specify that the status-code-id IS FAILURE.

Example 4–1 shows a success/failure test.

Example 4–1: Success/Failure Test

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAIN-PROG.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   RETURN-STATUS      PIC S9(9) COMP.
PROCEDURE DIVISION.
.
.
.
    CALL "PROG-1" GIVING RETURN-STATUS.
    IF RETURN-STATUS IS FAILURE PERFORM FAILURE-ROUTINE.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-1.
.
.
.
WORKING-STORAGE SECTION.
01  RETURN-STATUS     PIC S9(9) COMP.
PROCEDURE DIVISION GIVING RETURN-STATUS.
.
.
.
    IF NUM-1 = NUM-2
              SET RETURN-STATUS TO SUCCESS
    ELSE
              SET RETURN-STATUS TO FAILURE.
.
.
.
    EXIT PROGRAM.
END PROGRAM PROG-1.
END PROGRAM MAIN-PROG.
```

4.5 Using the MOVE Statement

The MOVE statement moves the contents of one item into another item. The following sample MOVE statement moves the contents of item FIELD1 into item FIELD2:

```
MOVE FIELD1 TO FIELD2.
```

This section considers MOVE statements as applied to numeric and numeric edited data items.

4.5.1 Elementary Numeric Moves

If both items of a MOVE statement are elementary items and the receiving item is numeric, it is an elementary numeric move. The sending item can be either numeric or alphanumeric. The elementary numeric move converts the data format of the sending item to the data format of the receiving item.

An alphanumeric sending item can be either of the following:

- An elementary data item
- Any alphanumeric literal other than the figurative constants SPACE, QUOTE, LOW-VALUE, or HIGH-VALUE

The elementary numeric move accepts the figurative constant ZERO and considers it to be equivalent to the numeric literal 0. It treats alphanumeric sending items as unsigned integers of DISPLAY usage.

If necessary, the numeric move operation converts the sending item to the data format of the receiving item and aligns the sending item's decimal point on that of the receiving item. Then it moves the sending item's digits to the corresponding receiving item's digits.

If the sending item has more digit positions than the receiving item, the decimal point alignment operation truncates the sending item, with resulting loss of digits.

The end truncated (high-order or low-order) depends upon the number of sending item digit positions that find matches on each side of the receiving item's decimal point.

If the receiving item has fewer digit positions on both sides of the decimal point, the operation truncates both ends of the sending item. Thus, if an item described as PIC 999V999 is moved to an item described as PIC 99V99, it loses one digit from the left end and one from the right end.

In the following example, the caret (^) indicates the assumed decimal scaling position:

```
01 AMOUNT1 PIC 99V99 VALUE ZEROS.
   .
   .
   .
   MOVE 123.321 TO AMOUNT1.
```

```
Before execution:  00^00
After execution:   23^32
```

If the sending item has fewer digit positions than the receiving item, the move operation supplies zeros for all unfilled digit positions. The caret (^) indicates the assumed stored decimal scaling position:

```
01  TOTAL-AMT PIC 999V99 VALUE ZEROS.
    .
    .
    .
    MOVE 1 TO TOTAL-AMT.
```

```
Before execution:   000^00

After execution:    001^00
```

The following statements produce the same results:

```
MOVE 001.00 TO TOTAL-AMT.

MOVE "1" TO TOTAL-AMT.
```

Consider the following two MOVE statements and their truncating and zero-filling effects:

Statement	TOTAL-AMT After Execution
MOVE 00100 TO TOTAL-AMT	100^00
MOVE "00100" TO TOTAL-AMT	100^00

Literals with leading or trailing zeros have no advantage in space or execution speed in VAX COBOL, and the zeros are often lost by decimal point alignment.

The MOVE statement's receiving item dictates how the sign will be moved. When the receiving item is a signed numeric item, the sign from the sending item is placed in it. If the sending item is unsigned, a positive sign is placed in the receiving item.

4.5.2 Elementary Numeric Edited Moves

An elementary numeric move to a numeric edited receiving item is considered an elementary numeric edited move. The sending item of an elementary numeric edited move can be either numeric or alphanumeric. When the sending item is numeric edited, de-editing is implied to establish the item's unedited numeric value, which may be signed; then the unedited numeric value is moved to the receiving field. Alphanumeric sending items in numeric edited moves are considered unsigned DISPLAY usage integers.

A numeric edited item PICTURE can contain 9, V, and P, but to qualify as numeric edited, it must also contain one or more of the other editing symbols: Z, B, and the asterisk (*). For a complete listing and description of these symbols see the *VAX COBOL Reference Manual.*

The numeric edited move operation first converts the sending item to DISPLAY usage and aligns both items on their decimal point locations. The sending item is truncated or zero-filled until it has the same number of digit positions on both sides of the decimal point as the receiving item. The operation then moves the sending item to the receiving item, following the VAX COBOL editing rules.

The rules allow the numeric edited move operation to perform any of the following editing functions:

- Suppress leading zeros with either spaces or asterisks
- Float a currency sign and a plus or minus sign through suppressed zeros, inserting the sign at either end of the item
- Insert zeros and spaces
- Insert commas and a decimal point (or decimal points and a comma if DECIMAL-POINT IS COMMA)

Table 4–3 illustrates several of these functions, which are invoked by the statement:

```
MOVE FLD-B TO TOTAL-AMT.
```

Assume that FLD-B is described as S9999V99. Note that the caret (^) indicates an assumed decimal point. Also, overpunch signs (the sign of the number encoded into the rightmost digit) are used in two FLD-B data examples.

Table 4–3: **Numeric Editing**

FLD-B	TOTAL-AMT	
	PICTURE String	**Contents After MOVE**
0023^00	ZZZZ.99	23.00
0085^9P	++++.99	-85.97
1234^00	Z,ZZZ.99	1,234.00
0012^34	$,$$$.99	$12.34
0000^34	$,$$9.99	$0.34
1234^00	$$,$$$.99	$1,234.00
0012^34	$$9,999.99	$0,012.34
0012^34	$$$$,$$$.99	$12.34
0000^00	$$$,$$$.$$	
0012^3M	++++.99	-12.34
0012^34	$***,***.99	$*****12.34
1234^56	Z,ZZZ.99+	1,234.56+
-6543^21	$,$ $$,$$$.99DB	$6,543.21DB[1]

[1] The output includes DB if a negative value is moved.

The currency symbol ($ or other currency sign) and the editing sign control symbols (+ and –) are the only floating symbols. To float a symbol, enter a string of two or more occurrences of that symbol, one for each character position over which you want the symbol to float.

4.5.3 Common Move Errors

Programmers most commonly make the following errors when writing MOVE statements:

- Placing an incorrect number of replacement characters in a numeric edited item
- Moving nonnumeric data into numeric items with group moves
- Trying to float the currency sign ($) or plus (+) insertion characters past the decimal point to force zero values to appear as .00 instead of spaces (use $$.99 or .99)
- Forgetting that the currency sign ($) or plus sign (+) insertion characters require an additional position on the leftmost end that cannot be replaced by a digit (unlike the asterisk (*) insertion character, which can be completely replaced)

4.6 Using the Arithmetic Statements

The VAX COBOL arithmetic statements allow programs to perform arithmetic operations on numeric data. The following sections explain how to use these statements.

4.6.1 Intermediate Results

Most forms of the arithmetic statements perform their operations in temporary work locations, then move the results to the receiving items, aligning the decimal points and truncating or zero-filling the resultant values. This temporary work item, called the intermediate result item, has a maximum size of 26 numeric digits. The actual size of the intermediate result varies for each statement; it is determined at compile time, based on the sizes of the operands used by the statement.

When the compiler determines that the size of the intermediate result exceeds 26 digits, it uses a software floating-point intermediate item and keeps the most significant 26 digits, bypassing leading zeros. If possible, do fewer complex arithmetic operations that use intermediate temporaries. (See Section 4.6.6.)

4.6.2 Specifying a Truncation Qualifier

The /[NO]TRUNCATE compile-time qualifier specifies how the VAX COBOL compiler stores values in COMPUTATIONAL receiving items if high-order truncation is necessary.

By default (/NOTRUNCATE), VAX COBOL truncates values according to the VAX hardware storage unit (word, longword, or quadword) allocated to the receiving item.

If you specify the /TRUNCATE option, the compiler truncates values according to the number of decimal digits specified by the PICTURE size.

4.6.3 Using the ROUNDED Phrase

Rounding is an important option that you can use with arithmetic operations.

You can use the ROUNDED phrase with any VAX COBOL arithmetic statement. Rounding takes place only when the ROUNDED phrase requests it—and then only if the intermediate result has more low-order digits than the result.

VAX COBOL rounds off by adding a 5 to the leftmost truncated digit of the absolute value of the intermediate result before it stores that result.

Table 4–4 shows several ROUNDING examples.

Table 4–4: ROUNDING

PICTURE clause	Initial Value
03 ITEMA PIC S9(5)V9999.	12345.2222
03 ITEMB PIC S9(5)V99.	54321.11
03 ITEMC PIC S9999.	4321
03 ITEME PIC S99V99 VALUE 9.	9.0
03 ITEMF PIC S99V99 VALUE 24.	24.00

(continued on next page)

Table 4–4 (Cont.): ROUNDING

PICTURE clause		Initial Value
Arithmetic Statement	**Intermediate Results**	**ROUNDED Result**
ADD ITEMA TO ITEMB ROUNDED.	066666.3322	66666.33
MULTIPLY ITEMC BY 10 GIVING ITEMD ROUNDED.	043210	0432
DIVIDE ITEME INTO ITEMF ROUNDED.	02.666	02.67

4.6.3.1 ROUNDED with REMAINDER

The remainder computation uses an intermediate field that is truncated, rather than rounded, when you use the DIVIDE statement with both the ROUNDED and REMAINDER options.

4.6.4 Using the SIZE ERROR Phrase

The SIZE ERROR phrase detects the loss of high-order nonzero digits in the results of VAX COBOL arithmetic operations.

You can use the phrase in any VAX COBOL arithmetic statement.

When the execution of a statement with no SIZE ERROR phrase results in a size error, the high-order digits are truncated and the results are stored without notifying the user. When the same statement includes a SIZE ERROR phrase, the entire result is discarded without altering the receiving items in any way; the SIZE ERROR imperative phrase is then executed.

If the statement contains both ROUNDED and SIZE ERROR phrases, the result is rounded before a size error check is made.

The SIZE ERROR phrase cannot be used with numeric MOVE statements. Thus, if a program moves a numeric quantity to a smaller numeric item, it can lose high-order digits. For example, consider the following move of an item to a smaller item:

```
01 AMOUNT-A PIC S9(8)V99.
01 AMOUNT-B PIC S9(4)V99.
          .
          .
          .
          MOVE AMOUNT-A TO AMOUNT-B.
```

This MOVE operation always loses four of AMOUNT-A's high-order digits. The statement can be tailored either of two ways, as shown in the following example, to determine whether these digits are zero or nonzero.

```
1.  IF AMOUNT-A NOT > 9999.99
        MOVE AMOUNT-A TO AMOUNT-B
        ELSE ...
2.  ADD ZERO AMOUNT-A GIVING AMOUNT-B
        ON SIZE ERROR ...
```

Both alternatives allow the MOVE operation to occur only if AMOUNT-A loses no significant digits. If the value in AMOUNT-A is too large, both avoid altering AMOUNT-B and take the alternate execution path.

You can also use a NOT ON SIZE ERROR phrase to branch to, or perform sections of code only when no size error occurs.

4.6.5 Using the GIVING Phrase

The GIVING phrase moves the intermediate result of an arithmetic operation to a receiving item. The phrase acts exactly like a MOVE statement in which the intermediate result serves as the sending item, and the data item following the word GIVING serves as the receiving item. When a statement contains a GIVING phrase, you can have a numeric edited receiving item.

The GIVING phrase can be used with the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. For example:

```
ADD A,B GIVING C.
```

4.6.6 Multiple Operands in ADD and SUBTRACT Statements

Both the ADD and SUBTRACT statements can contain a series of operands preceding the word TO, FROM, or GIVING.

If there are multiple operands in either of these statements, the operands are added together. The intermediate result of that operation becomes a single operand to be added to, or subtracted from, the receiving item. In the following examples, TEMP is an intermediate result item:

1.	Statement:	ADD A,B,C,D, TO E,F,G,H.
	Equivalent coding:	ADD A B, GIVING TEMP. ADD TEMP, C, GIVING TEMP. ADD TEMP, D, GIVING TEMP. ADD TEMP, E, GIVING E. ADD TEMP, F, GIVING F. ADD TEMP, G, GIVING G. ADD TEMP, H, GIVING H.
2.	Statement:	SUBTRACT A, B, C, FROM D.
	Equivalent coding:	ADD A, B, GIVING TEMP. ADD TEMP, C, GIVING TEMP. SUBTRACT TEMP FROM D, GIVING D.
3.	Statement:	ADD A,B,C,D, GIVING E.
	Equivalent coding:	ADD A,B, GIVING TEMP. ADD TEMP, C, GIVING TEMP. ADD TEMP, D, GIVING E.

As in all VAX COBOL statements, the commas in these statements are optional.

4.6.7 Common Errors in Arithmetic Statements

Programmers most commonly make the following errors when using arithmetic statements:

- Using an alphanumeric item in an arithmetic statement. The MOVE statement allows data movement between alphanumeric items and certain numeric items, but arithmetic statements require that all items be numeric.
- Writing the ADD or SUBTRACT statements without the GIVING phrase, and attempting to put the result into a numeric edited item.

- Subtracting a 1 from a numeric counter that was described as an unsigned quantity and then testing for a value less than zero.
- Forgetting that the MULTIPLY statement, without the GIVING phrase, stores the result back into the second operand (multiplier).
- Performing a series of calculations that generates an intermediate result larger than 26 digits when the final result will have 18 or fewer digits. You can prevent this problem by interspersing divisions with multiplications or by dropping nonsignificant digits after multiplying large numbers or numbers with many decimal places.
- Performing an operation on an item that contains a value greater than the precision of its data description. This can happen only if the item was overwritten by a group move or redefinition.
- Forgetting that you must specify the ROUNDED phrase for each item in an arithmetic statement containing multiple receiving items.
- Forgetting that the ON SIZE ERROR phrase applies to all receiving items in an arithmetic statement containing multiple receiving items. Only those receiving items for which a size error condition is raised are left unaltered. The ON SIZE ERROR imperative statement is executed after all the receiving items are processed.
- Controlling a loop by adding to a numeric counter that was described as PIC 9, and then testing for a value of 10 or greater to exit the loop.
- Forgetting that ROUNDING is done before the ON SIZE ERROR test.

4.7 Arithmetic Expression Processing

VAX COBOL provides the arithmetic statements ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE, and the facilities of arithmetic expressions using the +, –, *, /, and ** operators. You can perform a given arithmetic computation in any of several ways. For example, if you want to compute a salesman's total yearly sales as the sum of the four individual sales quarters, you might use this sample code:

```
      .
      .
      .
MOVE 1ST-SALES TO TEMP.

ADD 2ND-SALES TO TEMP.
ADD 3RD-SALES TO TEMP.
ADD 4TH-SALES TO TEMP, GIVING TOTAL-SALES.
      .
      .
      .
```

In this example, a series of single ADD statements computes the final value of TOTAL-SALES by holding the partial sums in a temporary location called TEMP, which you defined in the Data Division of the program. You specify the class, usage, and number of integer and decimal places to be maintained.

Another possible solution to the problem is as follows:

```
      .
      .
      .
ADD 1ST-SALES, 2ND-SALES, 3RD-SALES, 4TH-SALES
                    GIVING TOTAL-SALES.
      .
      .
      .
```

In this example, the program computes TOTAL-SALES using a single ADD statement. As in the previous example, an intermediate result is required to develop the partial sums of the four quarterly sales quantities. However, in this example, the compiler defines the intermediate result in a manner transparent to the source program. It allocates storage for and assigns various attributes to this result according to the rules defined by VAX COBOL. For more information refer to the *VAX COBOL Reference Manual.*

In the next example, consider another computational method:

```
      .
      .
      .
COMPUTE TOTAL-SALES = 1ST-SALES + 2ND-SALES + 3RD-SALES + 4TH-SALES.
      .
      .
      .
```

This sample coding uses a single COMPUTE statement with an embedded arithmetic expression. Again, an intermediate result is required and is defined by the compiler. The compiler generates the intermediate result using the following rule:

Arithmetic operations are combined without restrictions on composites of operands and/or receiving items.

(See information about arithmetic operations and rules in the *VAX COBOL Reference Manual.*)

Chapter 5

Nonnumeric Data Handling

COBOL programs hold their data in items whose sizes are described in their source programs. The size of these items is thus fixed during compilation for the lifespan of the resulting object program.

Items in a COBOL program belong to any of three data classes—alphanumeric, alphabetic, or numeric. Numeric items contain only numeric values. Alphabetic items contain only A to Z (uppercase or lowercase) and space characters. Alphanumeric items can contain the following types of values:

- All alphabetic
- All numeric
- A mixture of alphabetic and numeric
- Any character from the ASCII character set

The data description of an item specifies which class that item belongs to.

Classes are further subdivided into categories. Alphanumeric items can be numeric edited, alphanumeric edited, or alphanumeric. Every elementary item, except for an index data item, belongs to one of the classes and its categories. The class of a group item is treated as alphanumeric regardless of the classes of subordinate elementary items.

If the data description of an alphanumeric item specifies that certain editing operations be performed on any value that is moved into it, that item is called an alphanumeric edited or a numeric edited item.

As you read this chapter, keep in mind the distinction between the class or category of a data item and the actual value that the item contains.

Sometimes the text refers to alphabetic, alphanumeric, and alphanumeric edited data items as nonnumeric data items to distinguish them from items that are specifically numeric.

Regardless of the class of an item, it is usually possible at run time to store an invalid value in the item. Thus, nonnumeric ASCII characters can be placed in an item described as numeric, and an alphabetic item can be loaded with nonalphabetic characters.

5.1 Data Organization

A VAX COBOL record consists of a set of data description entries that describe record characteristics; it must have an 01 or 77 level number. A data description entry can be either a group item or an elementary item.

All of the records used by VAX COBOL programs (except for certain registers and switches) must be described in the Data Division of the source program. The compiler allocates memory space for these items (except for Linkage Section items) and fixes their size at compilation time.

The following sections explain how the compiler handles group and elementary data items.

5.1.1 Group Items

A group item is a data item that is followed by one or more elementary items or other group items, all of which have higher-valued level numbers than the group to which they are subordinate.

The size of a group item is the sum of the sizes of its subordinate elementary items. The compiler considers all group items to be alphanumeric DISPLAY items regardless of the class and usage of their subordinate elementary items.

5.1.2 Elementary Items

An elementary item is a data item that has no subordinate data item.

The size of an elementary item is determined by the number of symbols that represent character positions contained in the PICTURE character-string. For example, consider this record description:

```
01 TRANREC.
   03 FIELD-1 PIC X(7).
   03 FIELD-2 PIC S9(5)V99.
```

Both elementary items require seven bytes of memory; however, item FIELD-1 contains seven alphanumeric characters while item FIELD-2 contains seven decimal digits, an operational sign, and an implied decimal point. Operations on such items are independent of the mapping of the item into memory words (32-bit words that hold four 8-bit bytes). An item can begin in the leftmost or rightmost byte of a word with no effect on the function of any operations that refer to that item.

In effect, the compiler sees memory as a continuous array of bytes, not words. This becomes particularly important when you are defining a table using the OCCURS clause (see Chapter 6).

Records, and elementary items with a 77 level number automatically begin on a longword boundary (multiple of 4 bytes).

5.2 Special Characters

VAX COBOL allows you to handle any of the 128 characters of the ASCII character set as alphanumeric data, even though many of the characters are control characters, which usually direct input/output devices. Generally, alphanumeric data manipulations attach no meaning to the 8th bit of an 8-bit byte. Thus, you can move and compare these control characters in the same manner as alphabetic and numeric characters.

Although the object program can manipulate all ASCII characters, certain control characters cannot appear in nonnumeric literals since the compiler uses them to delimit the source text.

You can place special characters into items of the object program by defining symbolic characters in the SPECIAL-NAMES paragraph or by using the EXTERNAL clause. See the *VAX COBOL Reference Manual* for information on these two topics.

The ASCII character set listed in the *VAX COBOL Reference Manual* indicates the decimal value for any ASCII character.

5.3 Testing Nonnumeric Items

The following sections describe the relation and class tests as they apply to nonnumeric items.

5.3.1 Relation Tests of Nonnumeric Items

An IF statement with a relation condition (greater than, less than, equal to) can compare the value in a nonnumeric data item with another value and use the result to alter the flow of control in the program.

An IF statement with a relation condition compares two operands. Either of these operands can be an identifier or a literal, but they cannot both be literals. If the stated relation exists between the two operands, the relation condition is true.

When coding a relational operator, leave a space before and after each reserved word. When the reserved word NOT is present, the compiler considers it and the next key word or relational character to be a single relational operator defining the comparison. Table 5–1 shows the meanings of the relational operators.

Table 5–1: Relational Operator Descriptions

Operator	Description
IS [NOT] GREATER THAN IS [NOT] >	The first operand is greater than (or not greater than) the second operand.
IS [NOT] LESS THAN IS [NOT] <	The first operand is less than (or not less than) the second operand.
IS [NOT] EQUAL TO IS [NOT] =	The first operand is equal to (or not equal to) the second operand.
IS GREATER THAN OR EQUAL TO IS >=	The first operand is greater than or equal to the second operand.
IS LESS THAN OR EQUAL TO IS <=	The first operand is less than or equal to the second operand.

5.3.1.1 Classes of Data

VAX COBOL allows comparison of both numeric class operands and nonnumeric class operands; however, it handles each class of data differently. For example, it allows a comparison of two numeric operands regardless of the formats specified in their respective USAGE clauses, but it requires that all other comparisons (including comparisons of any group items) be between operands with the same usage. It compares numeric class operands with respect to their algebraic values and nonnumeric (or numeric and nonnumeric) class operands with respect to a specified collating sequence.

If only one of the operands is numeric, it must be an integer data item or an integer literal, and it must be DISPLAY usage. The manner in which the compiler handles numeric operands depends on the nonnumeric operand.

- If the nonnumeric operand is an elementary item or a literal, the compiler treats the numeric operand as if it had been moved into an alphanumeric data item the same size as the numeric operand and then compared. This causes any operational sign, whether carried as a separate character or as an overpunched character, to be stripped from the numeric item so that it appears to be an unsigned quantity.

  In addition, if the PICTURE character-string of the numeric item contains trailing P characters, indicating that there are assumed integer positions that are not actually present, they are filled with zero digits. Thus, an item with a PICTURE character-string of S9999PPP is moved to a temporary location where it is described as 9999999. If its value is 432J (–4321), the value in the temporary location will be 4321000. The numeric digits take part in the comparison.

- If the nonnumeric operand is a group item, the compiler treats the numeric operand as if it had been moved into a group item the same size as the numeric operand and then compared. This is equivalent to a group move.

  The compiler ignores the description of the numeric item (except for length) and, therefore, includes in its length any operational sign, whether carried as a separate character or as an overpunched character. Overpunched characters are never ASCII numeric digits. They are characters ranging from A to R, left brace ({), or right brace (}). Thus, the sign and the digits, stored as ASCII bytes, take part in the comparison, and zeros are not supplied for P characters in the PICTURE character-string.

The compiler does not accept a comparison between a noninteger numeric operand and a nonnumeric operand. If you try to compare these two items, you receive a diagnostic message at compile time.

5.3.1.2 Comparison Operations

If the two operands are acceptable, the compiler compares them character by character. The compiler starts at the first byte and compares the corresponding bytes until it either encounters a pair of unequal bytes or reaches the last byte of the longer operand.

If the compiler encounters a pair of unequal characters, it considers their relative position in the collating sequence. The operand with the character that is positioned higher in the collating sequence is the greater operand.

If the operands have different lengths, the comparison proceeds as though the shorter operand is extended on the right by sufficient ASCII spaces (decimal 32) to make both operands the same length.

If all character pairs are equal, the operands are equal.

5.3.2 Class Tests for Nonnumeric Items

An IF statement with a class condition (NUMERIC or ALPHABETIC) tests the value in a nonnumeric data item (USAGE DISPLAY only) to determine whether it contains numeric or alphabetic data and uses the result to alter the flow of control in the program. For example:

```
IF ITEM-1 IS NUMERIC...
IF ITEM-2 IS ALPHABETIC...
IF ITEM-3 IS NOT NUMERIC...
```

If the data item consists entirely of the ASCII characters 0 to 9, with or without the operational sign, the class condition is NUMERIC. If the item consists entirely of the ASCII characters A to Z (upper- or lowercase) and spaces, the class condition is ALPHABETIC.

The ALPHABETIC-LOWER test is true if the operand contains any combination of the lowercase alphabetic characters a to z, and the space. Otherwise the test is false.

The ALPHABETIC-UPPER test is true if the operand contains any combination of the uppercase alphabetical characters A to Z, and the space. Otherwise, the test is false.

You can also perform a class test on a data item that you define with the CLASS clause of the SPECIAL-NAMES paragraph.

A class test is true if the operand consists entirely of the characters listed in the definition of the CLASS-NAME in the SPECIAL-NAMES paragraph. Otherwise, the test is false.

When the reserved word NOT is present, the compiler considers it and the next key word as one class condition defining the class test to be executed. For example, NOT NUMERIC determines if an operand contains at least one nonnumeric character.

If the item being tested is described as a numeric data item, it can only be tested as NUMERIC or NOT NUMERIC. The NUMERIC test cannot examine either of the following:

- An item described as alphabetic
- A group item containing elementary items whose data descriptions indicate the presence of operational signs

For further information on using class conditions with numeric items, refer to the *VAX COBOL Reference Manual.*

5.4 Data Movement

Three VAX COBOL statements (MOVE, STRING, and UNSTRING) perform most of the data movement operations required by business-oriented programs. The MOVE statement simply moves data from one item to another. The STRING statement concatenates a series of sending items into a single receiving item. The UNSTRING statement disperses a single sending item into multiple receiving items. Section 5.5 describes the MOVE statement. Chapter 7 describes STRING and UNSTRING.

The MOVE statement handles most data movement operations on character strings. However, it is limited in its ability to handle multiple items. For example, it cannot, by itself, concatenate a series of sending items into a single receiving item or disperse a single sending item into several receiving items.

Two MOVE statements will, however, bring the contents of two items together into a third (receiving) item if the receiving item has been subdivided with subordinate elementary items that match the two sending items in size. If other items are to be concatenated into the third item, and they differ in size from the first two items, then the receiving item requires additional subdivisions (through redefinition).

Example 5–1 demonstrates item concatenation using two MOVE statements.

Example 5–1: Item Concatenation Using Two MOVE Statements

```
01  SEND-1         PIC X(5) VALUE "FIRST".
    .
    .
    .
01  SEND-2         PIC X(6) VALUE "SECOND".
01  RECEIVE-GROUP.
    05  REC-1      PIC X(5).
    05  REC-2      PIC X(6).
PROCEDURE DIVISION.
A00-BEGIN.
    MOVE SEND-1 TO REC-1.
    MOVE SEND-2 TO REC-2.
    DISPLAY RECEIVE-GROUP.
    STOP RUN.
```

The result of the concatenation follows:

```
FIRSTSECOND
```

Two MOVE statements can also disperse the contents of one sending item to several receiving items. The first MOVE statement moves the leftmost end of the sending item to a receiving item; then the second MOVE statement moves the rightmost end of the sending item to another receiving item. (The second receiving item must first be described with the JUSTIFIED clause.) Characters from the middle of the sending item cannot easily be moved to any receiving item without extensive redefinitions of the sending item or a reference modification loop (as with concatenation).

The STRING and UNSTRING statements handle concatenation and dispersion more easily than compound moves. Reference modification handles substring operations more easily than compound moves, STRING, or UNSTRING.

5.5 Using the MOVE Statement

The MOVE statement moves the contents of one item into another. For example:

```
MOVE FIELD1 TO FIELD2
```

```
MOVE CORRESPONDING FIELD1 TO FIELD2
```

FIELD1 is the sending item name, and FIELD2 is the receiving item name.

The first statement causes the compiler to move the contents of FIELD1 into FIELD2. The two items need not be the same size, class, or usage; they can be either group or elementary items. If the two items are not the same length, the compiler aligns them on one end or the other. It also truncates or space-fills the other end. The movement of group items and nonnumeric elementary items is discussed in Section 5.5.1 and Section 5.5.2, respectively.

The MOVE statement alters the contents of every character position in the receiving item.

5.5.1 Group Moves

If either the sending or receiving item is a group item, the compiler considers the move to be a group move. It treats both the sending and receiving items as if they are alphanumeric items.

If the sending item is a group item, and the receiving item is an elementary item, the compiler ignores the receiving item description except for the size description, in bytes, and any JUSTIFIED clause. It conducts no conversion or editing on the receiving item.

5.5.2 Elementary Moves

If both items of a MOVE statement are elementary items, their PICTURE character-strings control their data movement. If the receiving item was described as numeric or numeric edited, the rules for numeric moves control the data movement (see Chapter 4). Nonnumeric receiving items are alphanumeric, alphanumeric edited, or alphabetic.

Table 5–2 shows the valid and invalid nonnumeric elementary moves.

Table 5–2: Nonnumeric Elementary Moves

	Receiving Item Category	
Sending Item Category	**Alphabetic**	**Alphanumeric** **Alphanumeric Edited**
ALPHABETIC	Valid	Valid
ALPHANUMERIC	Valid	Valid
ALPHANUMERIC EDITED	Valid	Valid
NUMERIC INTEGER (DISPLAY ONLY)	Invalid	Valid
NUMERIC EDITED	Invalid	Valid

In all valid moves, the compiler treats the sending item as though it had been described as PIC X(n). A JUSTIFIED clause in the sending item's description has no effect on the move. If the sending item's PICTURE character-string contains editing characters, the compiler uses them only to determine the item's size.

In valid nonnumeric elementary moves, the receiving item controls the movement of data. All of the following characteristics of the receiving item affect the move:

- Its size
- Editing characters in its description
- The JUSTIFIED RIGHT clause in its description

The JUSTIFIED clause and editing characters are mutually exclusive.

When an item that contains no editing characters or JUSTIFIED clause in its description is used as the receiving item of a nonnumeric elementary MOVE statement, the compiler moves the characters starting at the leftmost position in the item and scans across, character by character, to the rightmost position. If the sending item is shorter than the receiving item, the compiler fills the remaining character positions with spaces. If the sending item is longer than the receiving item, truncation occurs on the right.

Numeric items used in nonnumeric elementary moves must be integers in DISPLAY format.

If the description of the numeric data item indicates the presence of an operational sign (either as a character or an overpunched character), or if there are P characters in its character-string, the compiler first moves the item to a temporary location. It removes the sign and fills out any P character positions with zero digits. It then uses the temporary value as the sending item as if it had been described as PIC X(n). The temporary value can be shorter than the original value if a separate sign was removed, or longer than the original value if P character positions were filled in with zeros.

If the sending item is an unsigned numeric class item with no P characters in its character-string, the MOVE is accomplished directly from the sending item, and a temporary item is not required.

If the numeric sending item is shorter than the receiving item, the compiler fills the receiving item with spaces.

5.5.2.1 Edited Moves

This section explains the following insertion editing characters:

B	Blank insertion position
0	Zero insertion position
/	Slash insertion position

When an item with an insertion editing character in its PICTURE character-string is the receiving item of a nonnumeric elementary MOVE statement, each receiving character position corresponding to an editing character receives the insertion byte value. Table 5–3 illustrates the use of such symbols with the following statement, where FIELD1 is described as PIC X(7):

```
MOVE FIELD1 TO FIELD2
```

Table 5–3: Data Movement with Editing Symbols

FIELD1	FIELD2	
	Character-String	Contents After MOVE
070476	XX/99/XX	07/04/76
04JUL76	99BAAAB99	04sJULs76
2351212	XXXBXXXX/XX/	235s1212/ss/
123456	0XB0XB0XB0X	01s02s03s04

Legend: s = space

Data movement always begins at the left end of the sending item and moves only to the byte positions described as A, 9, or X in the receiving item PICTURE character-string. When the sending item is exhausted, the compiler supplies space characters to fill any remaining character positions (not insertion positions) in the receiving item. If the receiving item is exhausted before the last character is moved from the sending item, the compiler ignores the remaining sending item characters.

Any necessary conversion of data from one form of internal representation to another takes place during valid elementary moves, along with any editing specified for, or de-editing implied by, the receiving data item.

5.5.2.2 Justified Moves

A JUSTIFIED RIGHT clause in the receiving item's data description causes the compiler to reverse its usual data movement conventions. It starts with the rightmost characters of both items and proceeds from right to left. If the sending item is shorter than the receiving item, the compiler fills the remaining leftmost character positions with spaces. If the sending item is longer than the receiving item, truncation occurs on the left. Table 5–4 illustrates various PICTURE character-string situations for the following statement:

```
MOVE FIELD1 TO FIELD2
```

Table 5–4: Data Movement with the JUSTIFIED Clause

FIELD1		FIELD2	
PICTURE Character-String	Contents	PICTURE Character-String (and JUST-Clause)	Contents After MOVE
		XX	AB
		XXXXX	ABCss
XXX	ABC	XX JUST	BC
		XXXXX JUST	ssABC

Legend: s = space

If there is no JUSTIFIED clause, data movement follows the rules for aligning data in elementary items.

5.5.3 Multiple Receiving Items

If you write a MOVE statement containing more than one receiving item, the compiler moves the same sending item value to each of the receiving items. It has essentially the same effect as a series of separate MOVE statements, all with the same sending item. For information on subscripted items, see Section 5.5.4. Also, reference modification is evaluated immediately after subscripting or index evaluation. Refer to the *VAX COBOL Reference Manual* for details on reference modification.

The receiving items need have no relationship to each other. The compiler checks the validity of each one independently and performs an independent move operation on each one.

Multiple receiving items on MOVE statements provide a convenient way to set many items equal to the same value, such as during initialization code at the beginning of a section of processing. For example:

```
MOVE SPACES TO LIST-LINE, EXCEPTION-LINE, NAME-FLD.

MOVE ZEROS TO EOL-FLAG, EXCEPT-FLAG, NAME-FLAG.

MOVE 1 TO COUNT-1, CHAR-PTR, CURSOR.
```

5.5.4 Subscripted Moves

Any item (other than a data item that is not subordinate to an OCCURS clause) of a MOVE statement can be subscripted, and the referenced item can be used to subscript another name in the same statement.

For example, when more than one receiving item is named in the same MOVE statement, the order in which the compiler evaluates the subscripts affects the results of the move. Consider the following examples:

```
MOVE FIELD1(FIELD2) TO FIELD2 FIELD3.
```

In this example, the compiler evaluates FIELD1(FIELD2) only once, before it moves any data to the receiving items. It is as if the single MOVE statement were replaced with the following three statements:

```
MOVE FIELD1(FIELD2) TO TEMP.

MOVE TEMP TO FIELD2.

MOVE TEMP TO FIELD3.
```

In the following example, the compiler evaluates FIELD3(FIELD2) immediately before moving the data into it, but after moving the data from FIELD1 to FIELD2.

```
MOVE FIELD1 TO FIELD2 FIELD3(FIELD2).
```

Thus, it uses the newly stored value of FIELD2 as the subscript value. It is as if the single MOVE statement were replaced with the following two statements:

```
MOVE FIELD1 TO FIELD2.

MOVE FIELD1 TO FIELD3(FIELD2).
```

5.5.5 Common Nonnumeric Item MOVE Statement Errors

The compiler considers any MOVE statement that contains a group item (sending or receiving) to be a group move. If an elementary item contains editing characters or a numeric integer, these attributes of the receiving item, which determine the action of an elementary move, have no effect on the action of a group move.

5.5.6 Using the MOVE CORRESPONDING Statement for Nonnumeric Items

The MOVE CORRESPONDING statement allows you to move multiple items from one group item to another group item, using a single MOVE statement. See the *VAX COBOL Reference Manual* for rules concerning the CORRESPONDING phrase. When you use the CORRESPONDING phrase, the compiler performs an independent move operation on each pair of corresponding items from the operands and checks the validity of each. Example 5–2 shows the use of the MOVE CORRESPONDING statement.

The preceding MOVE CORRESPONDING statement is equivalent to the following series of MOVE statements:

Example 5–2: Sample Record Description Using the MOVE CORRESPONDING Statement

```
01 A-GROUP.                          01 B-GROUP.
   02 FIELD1.                           02 FIELD1.
      03 A PIC X.                          03 A PIC X.
      03 B PIC 9.                          03 C PIC XX.
      03 C PIC XX.                         03 E PIC XXX.
      03 D PIC 99.
      03 E PIC XXX.

    MOVE CORRESPONDING
         A-GROUP TO B-GROUP.
```

```
MOVE A OF A-GROUP TO A OF B-GROUP.

MOVE C OF A-GROUP TO C OF B-GROUP.

MOVE E OF A-GROUP TO E OF B-GROUP.
```

5.5.7 Using Reference Modification

You can use reference modification to define a subset of a data item by specifying its leftmost character position and length. Reference modification is valid anywhere an alphanumeric identifier is allowed unless specific rules for a general format prohibit it. The following is an example of a reference modification:

```
WORKING-STORAGE SECTION.
01  ITEMA  PIC X(10)  VALUE IS "XYZABCDEFG".
        .
        .
        .
    MOVE ITEMA(4:3) TO...

     IDENTIFIER             VALUE

ITEMA (4:3)                 ABC
```

For more information on reference modification rules, refer to the *VAX COBOL Reference Manual.*

Chapter 6

Table Handling

6.1 Introduction

This chapter discusses the procedures required to define, initialize, and access tables accurately and efficiently.

A table is one or more repetitions of one element, comprised of one or more data items, stored in contiguous memory locations. You define a table by using an OCCURS clause following a data description entry. The literal integer value you specify in the OCCURS clause determines the number of repetitions, or occurrences, of the data description entry, thus creating a table. VAX COBOL allows you to define from 1 to 48 dimensional tables.

After you have defined a table, you can load it with data. One way to load a table is to use the INITIALIZE statement or the VALUE clause to assign values to the table when you define it (see Figure 6–10).

To access data stored in tables, use subscripted or indexed procedural instructions. In either case, you can directly access a known table element occurrence or search for an occurrence based on some known condition.

6.2 Defining Tables

To define a table you specify an OCCURS clause in a data description entry. You can define either fixed-length tables or variable-length tables. They may furthermore be single or multidimensional. The following sections describe how to use the OCCURS clause and its options.

6.2.1 Defining Fixed-Length, One-Dimensional Tables

To define fixed-length tables, use Format 1 of the OCCURS clause (refer to the *VAX COBOL Reference Manual*). This format is useful when you are storing large amounts of stable or frequently used reference data. Options allow you to define single or multiple keys, or indexes, or both.

A definition of a one-dimensional table is shown in Example 6–1. The integer 2 in the OCCURS 2 TIMES clause determines the number of element repetitions. For the table to have any real meaning, this integer must be equal to or greater than 2.

The organization of TABLE-A is shown in Figure 6–1.

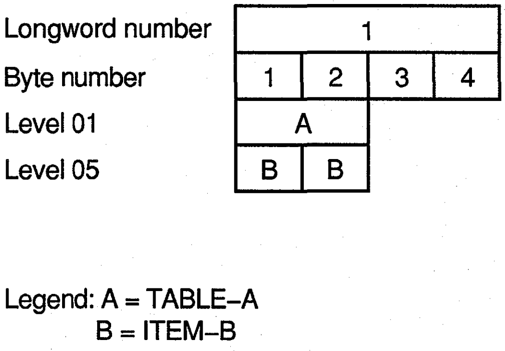
Example 6–1: One-Dimensional Table

```
01  TABLE-A.
    05  ITEM-B PIC X OCCURS 2 TIMES.
```

Example 6–2: Multiple Data Items in a One-Dimensional Table

```
01  TABLE-A.
    05  GROUP-B OCCURS 2 TIMES.
        10  ITEMC PIC X.
        10  ITEMD PIC X.
```

Figure 6–1: Organization of the One-Dimensional Table in Example 6–1



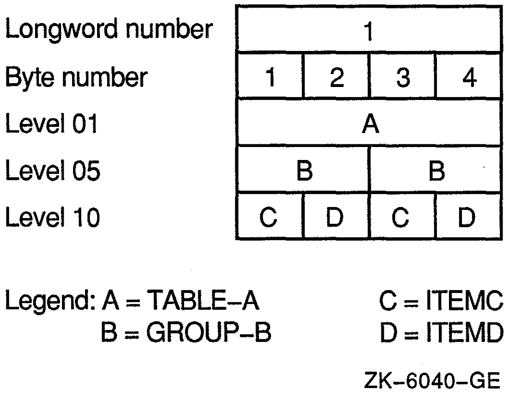
Legend: A = TABLE–A
B = ITEM–B

ZK–6039–GE

Example 6–1 specifies only a single data item. However, you can specify as many data items as you need in the table. Multiple data items are shown in Example 6–2.

The organization of this table is shown in Figure 6–2.

Figure 6–2: Organization of Multiple Data Items in a One-Dimensional Table



Example 6–1 and Example 6–2 both do not use the KEY IS or INDEXED BY optional phrases. The INDEXED BY phrase implicitly defines an index name. This phrase must be used if any Procedure Division statements contain indexed references to the data name that contains the OCCURS clause. The KEY IS phrase means that repeated data is arranged in ascending or descending order according to the values in the data items that contain the OCCURS clause. For further information on these OCCURS clause options, see the *VAX COBOL Reference Manual.*

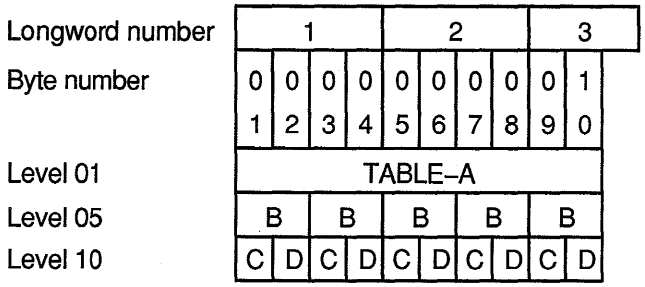
If you use either the SEARCH or the SEARCH ALL statement, you must specify at least one index. The SEARCH ALL statement also requires that you specify at least one key. Specify the search key using the ASCENDING/DESCENDING KEY IS phrase. See Section 6.4.8 for information about the SEARCH statement and Section 6.4.4 for information about indexing. When you use the INDEXED BY phrase, the index is internally defined and cannot be defined elsewhere. Example 6–3 defines a table with an ascending search key and an index.

Example 6–3: Defining a Table with an Index and an Ascending Search Key

```
01  TABLE-A.
    05  ELEMENTB OCCURS 5 TIMES
                 ASCENDING KEY IS ITEMA
                 INDEXED BY INDX1.
        10  ITEMC PIC X.
        10  ITEMD PIC X.
```

The organization of this table is shown in Figure 6–3.

Figure 6–3: Organization of a Table with an Index and an Ascending Search Key



<table>
<tr><td>Longword number</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="3">3</td></tr>
<tr><td>Byte number</td><td>01</td><td>02</td><td>03</td><td>04</td><td>05</td><td>06</td><td>07</td><td>08</td><td>09</td><td>10</td></tr>
<tr><td>Level 01</td><td colspan="10">TABLE–A</td></tr>
<tr><td>Level 05</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td></tr>
<tr><td>Level 10</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td></tr>
</table>

Legend: B = ELEMENTB
C = ITEMC
D = ITEMD

ZK–6041–GE

6.2.2 Defining Fixed-Length, Multidimensional Tables

VAX COBOL allows 48 levels of OCCURS nesting. If you want to define a two-dimensional table, you define another one-dimensional table within each element of the one-dimensional table. To define a three-dimensional table, you define another one-dimensional table within each element of the two-dimensional table, and so on.

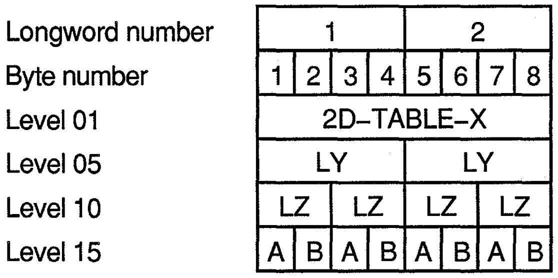
A two-dimensional table is shown in Example 6–4.

Example 6–4: Defining a Two-Dimensional Table

```
01  2D-TABLE-X.
    05  LAYER-Y OCCURS 2 TIMES.
        10  LAYER-Z OCCURS 2 TIMES.
            15  CELLA PIC X.
            15  CELLB PIC X.
```

The organization of this two-dimensional table is shown in Figure 6–4.

Figure 6–4: Organization of a Two-Dimensional Table



Legend: LY = LAYER–Y
LZ = LAYER–Z

A = CELLA
B = CELLB

ZK–6042–GE

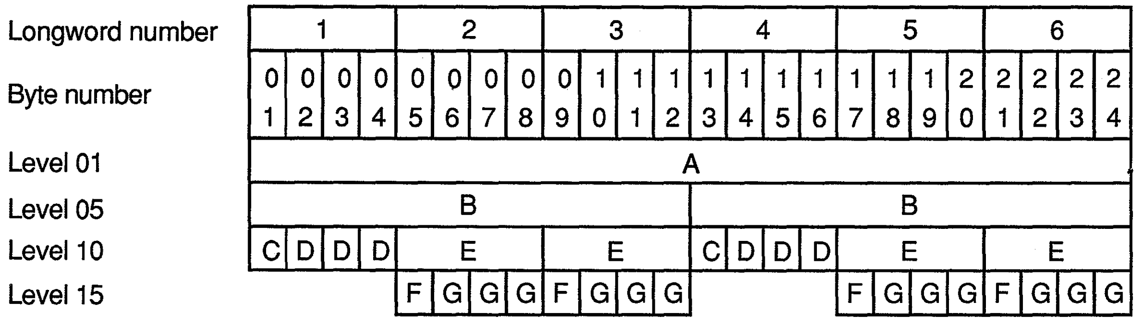
Example 6–5 shows a three-dimensional table.

Example 6–5: Defining a Three-Dimensional Table

```
01 TABLE-A.
    05  LAYER-B OCCURS 2 TIMES.
        10  ITEMC PIC X.
        10  ITEMD PIC X OCCURS 3 TIMES.
        10  ITEME OCCURS 2 TIMES.
            15  CELLF PIC X.
            15  CELLG PIC X OCCURS 3 TIMES.
```

The organization of this three-dimensional table is shown in Figure 6–5.

Figure 6–5: Organization of a Three-Dimensional Table



Legend:			
A	TABLE–A	E	ITEME
B	LAYER–B	F	CELLF
C	ITEMC	G	CELLG
D	ITEMD		

ZK–6043–GE

6.2.3 Defining Variable-Length Tables

To define a variable-length table, use Format 2 of the OCCURS clause (refer to the *VAX COBOL Reference Manual*). Options allow you to define single or multiple keys, or indexes, or both.

Example 6–6 illustrates how to define a variable-length table.

It uses from two to four occurrences depending on the integer value assigned to NUM-ELEM. You specify the table's minimum and maximum size with the OCCURS (minimum size) TO (maximum size) clause. The minimum size value must be equal to or greater than zero and the maximum size value must be greater than the minimum size value. The DEPENDING ON clause is also required when you use the TO clause.

The data-name of an elementary, unsigned integer data item is specified in the DEPENDING ON clause. Its value specifies the current number of occurrences. The data-name in the DEPENDING ON clause must be within the minimum to maximum range.

Unlike fixed-length tables, you can dynamically alter the number of element occurrences in variable-length tables.

By generating the variable-length table in Example 6–6, you are, in effect, saying: "Build a table that can contain at least two occurrences, but no more than four occurrences, and set its present number of occurrences equal to the value specified by NUM-ELEM."

Example 6–6: Defining a Variable-Length Table

```
01  NUM-ELEM PIC 9.
        .
        .
        .
01  VAR-LEN-TABLE.
    05  TAB-ELEM OCCURS 2 TO 4 TIMES DEPENDING ON NUM-ELEM.
        10 A PIC X.
        10 B PIC X.
```

6.2.4 Storage Allocation for Tables

The compiler maps the table elements into memory, following mapping rules that depend on the use of COMP, COMP-1, COMP-2, POINTER, and INDEX data items in the table element and the presence or absence of the SYNCHRONIZED (SYNC) clause with those data items.

The VAX COBOL compiler allocates storage for data items within records according to the rules of the Major-Minor Equivalence technique. This technique ensures that identically defined group items have the same structure, even when their subordinate items are aligned. Therefore, group moves always produce predictable results. For more information, refer to the description of record allocation in the *VAX COBOL Reference Manual.*

Figure 6–6 shows how the table defined in Example 6–7 is mapped into memory.

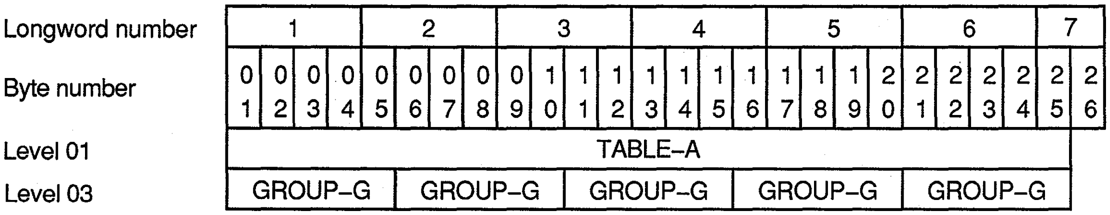
NOTE

> To determine exactly how much space your tables use, specify the /MAP compiler qualifier. This gives you an offset map of both the Data Division and the Procedure Division.

Example 6–7: Sample Record Description Defining a Table

```
01 TABLE-A.
   03 GROUP-G PIC X(5) OCCURS 5 TIMES.
```

Figure 6–6: Memory Map for Example 6–7



ZK–6050–GE

Alphanumeric data items require 1 byte of storage per character. Therefore, each occurrence of GROUP-G occupies 5 bytes. The first byte of the first element is automatically aligned at the left record boundary and the first 5 bytes occupy all of word 1 and part of 2. A memory longword is comprised of 4 bytes. Succeeding occurrences of GROUP-G are assigned to the next 5 adjacent bytes so that TABLE-A is comprised of five 5-byte elements for a total of 25 bytes. Each table element, after the first, is allowed to start in any byte of a word with no regard for word boundaries.

6.2.4.1 Using the SYNCHRONIZED Clause

By default, the VAX COBOL compiler tries to allocate a data item at the next unassigned byte location. However, you can align some data items on a 2-, 4-, or 8-byte boundary by using the SYNCHRONIZED clause. The compiler may then have to skip one or more bytes before assigning a location to the next data item. The skipped bytes, called fill bytes, are gaps between one data item and the next.

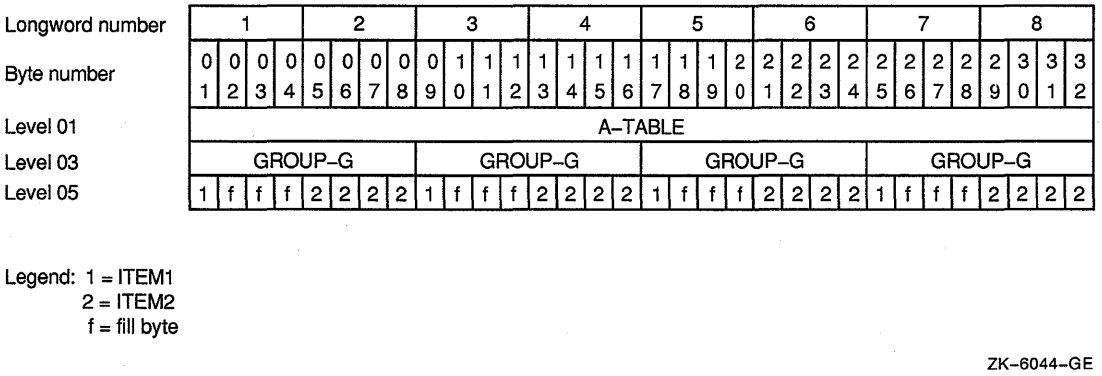
The SYNCHRONIZED clause explicitly aligns COMP, COMP-1, COMP-2, POINTER, and INDEX data items on their natural boundaries: one-word COMP items on 2-byte boundaries, longword items on 4-byte boundaries, and quadword items on 8-byte boundaries. Thus the use of SYNC can have a significant effect on the amount of memory required to store tables containing COMP and COMP SYNC data items.

Example 6–8 describes a table containing a COMP SYNC data item. Figure 6–7 illustrates how it is mapped into memory.

Example 6–8: Record Description Containing a COMP SYNC Item

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S9(5) COMP SYNC.
```

Figure 6–7: Memory Map for Example 6–8



Longword number	1				2				3				4				5				6				7				8			
Byte number	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Level 01	A-TABLE																															
Level 03	GROUP-G								GROUP-G								GROUP-G								GROUP-G							
Level 05	1	f	f	f	2	2	2	2	1	f	f	f	2	2	2	2	1	f	f	f	2	2	2	2	1	f	f	f	2	2	2	2

Legend: 1 = ITEM1
2 = ITEM2
f = fill byte

ZK-6044-GE

Because a 5-digit COMP SYNC item requires one longword (or 4 bytes) of storage, ITEM2 must start on a longword boundary. This requires the addition of 3 fill bytes after ITEM1, and each GROUP-G occupies 8 bytes. In Example 6–8, A-TABLE requires 32 bytes to store four elements of 8 bytes each.

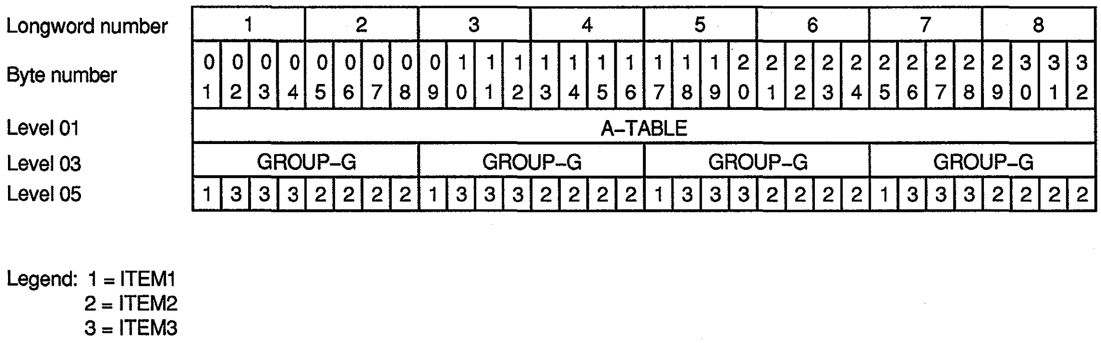
If, in the previous example, you defined ITEM2 as a COMP data item of the same size without the SYNC clause, the storage required would be considerably less. Although ITEM2 would still require one longword of storage, it would be aligned on a byte boundary. No fill bytes would be needed between ITEM1 and ITEM2, and A-TABLE would require a total of 20 bytes.

If you now add a 3-byte alphanumeric item (ITEM3) to Example 6–8 and locate it between ITEM1 and ITEM2 (see Example 6–9), the new item occupies the space formerly occupied by the 3 fill bytes. This adds 3 data bytes without changing the table size, as Figure 6–8 illustrates.

Example 6–9: Adding an Item Without Changing the Table Size

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM3 PIC XXX.
      05 ITEM2 PIC 9(5) COMP SYNC.
```

Figure 6–8: Memory Map for Example 6–9



Longword number	1				2				3				4				5				6				7				8			
Byte number	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Level 01	A-TABLE																															
Level 03	GROUP-G								GROUP-G								GROUP-G								GROUP-G							
Level 05	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2

Legend: 1 = ITEM1
2 = ITEM2
3 = ITEM3

ZK-6045-GE

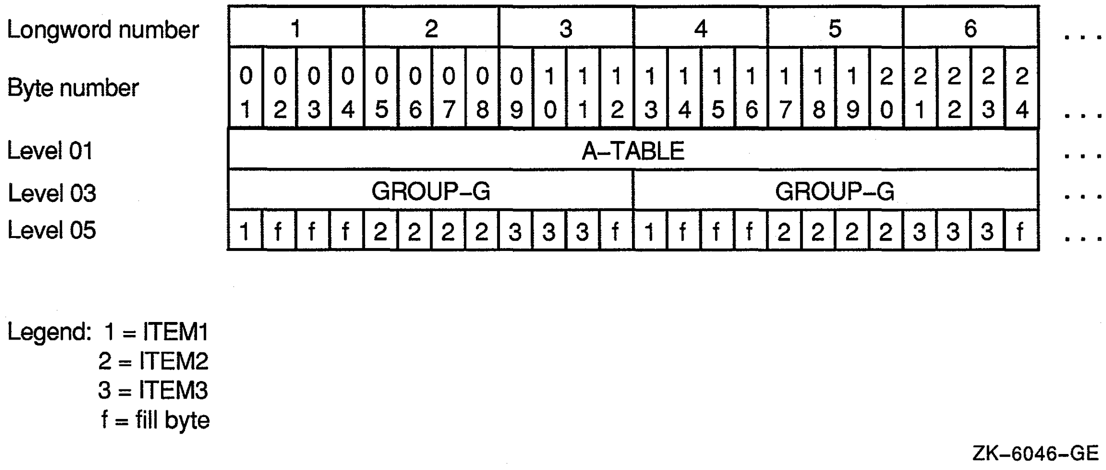
If, however, you place ITEM3 after ITEM2, the additional 3 bytes add their own length plus another fill byte. The additional fill byte is added after the third ITEM3 character to ensure that all occurrences of the table element are mapped in an identical manner. Now, each element requires 12 bytes, and the complete table occupies 48 bytes. This is illustrated by Example 6–10 and Figure 6–9.

Example 6–10: How Adding 3 Bytes Adds 4 Bytes to the Element Length

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC 9(5) COMP SYNC.
      05 ITEM3 PIC XXX.
```

Note that GROUP-G begins on a 4-byte boundary because of the way VAX COBOL allocates memory.

Figure 6–9: Memory Map for Example 6–10



6.3 Initializing Values of Table Elements

You can initialize a table that contains only DISPLAY items to any desired value in either of the following ways:

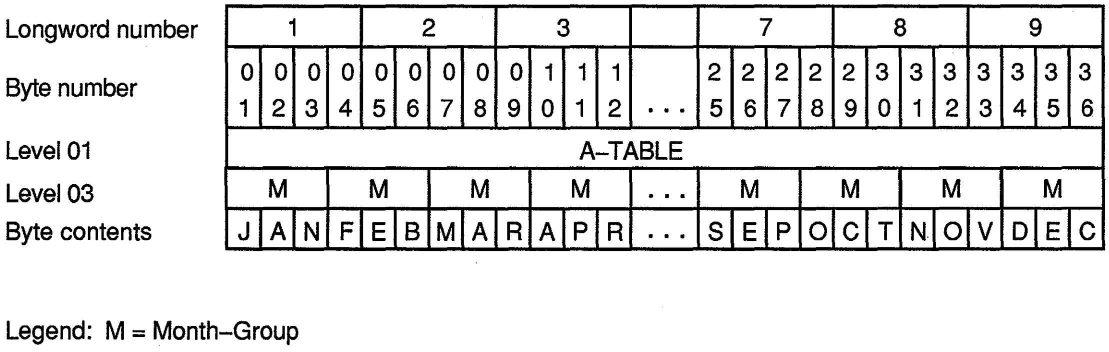
- You can specify a VALUE clause in the record level preceding the record description of the item containing the OCCURS clause.
- You can specify a VALUE clause in a record subordinate to the OCCURS clause.

Example 6–11 and Figure 6–10 provide an example and memory map of a table initialized using the VALUE clause.

Example 6–11: Initializing Tables with the VALUE Clause

```
01 A-TABLE VALUE IS "JANFEBMARAPRMAY
-       "JUNJULAUGSEPOCTNOVDEC".
    03 MONTH-GROUP PIC XXX USAGE DISPLAY
                OCCURS 12 TIMES.
```

Figure 6–10: Memory Map for Example 6–11



Byte number	01	02	03	04	05	06	07	08	09	10	11	12	...	25	26	27	28	29	30	31	32	33	34	35	36
Longword number	1				2				3					7				8				9			
Level 01	A–TABLE																								
Level 03	M			M			M			M			...	M			M			M			M		
Byte contents	J	A	N	F	E	B	M	A	R	A	P	R	...	S	E	P	O	C	T	N	O	V	D	E	C

Legend: M = Month–Group

ZK–6047–GE

If each entry in the table has the same value, you can initialize the table as shown in Example 6–12.

Example 6–12: Initializing a Table with the OCCURS Clause

```
01 A-TABLE.
   03 TABLE-LEG OCCURS 5 TIMES.
      05 FIRST-LEG   PIC X VALUE "A".
      05 SECOND-LEG  PIC S9(9) COMP VALUE 5.
         .
         .
         .
```

In this example, there are five occurrences of each table element. Each element is initialized to the same value as follows:

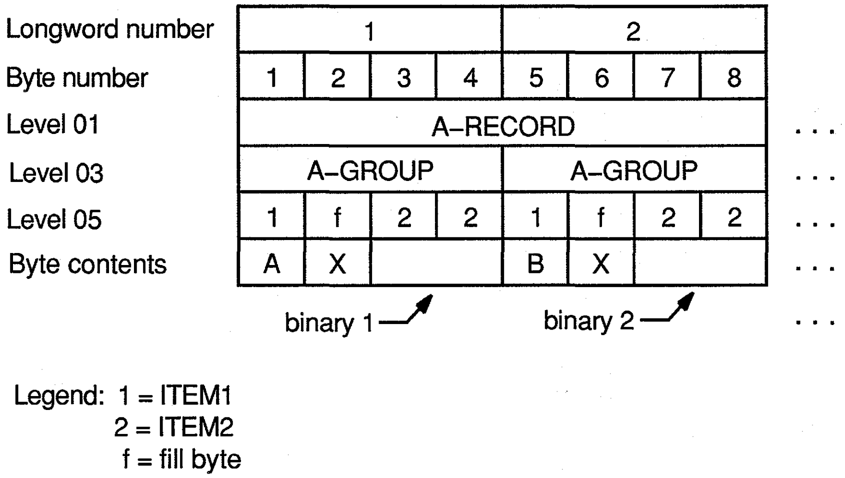
- FIRST-LEG occurs five times; each occurrence is initialized to A.
- SECOND-LEG occurs five times; each occurrence is initialized to 5.

Often a table is too long to initialize using a single literal, or it contains numeric, alphanumeric, COMP, COMP-1, COMP-2, or COMP SYNC items that cannot be initialized. In these situations, you can initialize individual items by redefining the group level that precedes the level containing the OCCURS clause. Consider the sample table descriptions illustrated in Example 6–13 and Example 6–14. Each fill byte between ITEM1 and ITEM2 in Example 6–13 is initialized to X. Figure 6–11 shows how this is mapped into memory.

Example 6–13: Initializing Mixed Usage Items

```
01 A-RECORD-ALT.
   05 FILLER PIC XX VALUE "AX".
   05 FILLER PIC S99 COMP VALUE 1.
   05 FILLER PIC XX VALUE "BX".
   05 FILLER PIC S99 COMP VALUE 2.
   .
   .
   .
01 A-RECORD REDEFINES A-RECORD-ALT.
   03 A-GROUP OCCURS 26 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S99 COMP SYNC.
```

Figure 6–11: Memory Map for Example 6–13



ZK-6048-GE

As shown in Example 6–14 and in Figure 6–12, each FILLER item initializes three 10-byte table elements.

Example 6–14: Initializing Alphanumeric Items

```
01 A-RECORD-ALT.
    03 FILLER PIC X(30) VALUE IS
       "AAAAAAAAAABBBBBBBBBBCCCCCCCCCC".
    03 FILLER PIC X(30) VALUE IS
       "DDDDDDDDDDEEEEEEEEEEFFFFFFFFFF".
    .
    .
    .
01 A-RECORD REDEFINES A-RECORD-ALT.
    03 ITEM1 PIC X(10) OCCURS 26 TIMES.
```

Figure 6–12: Memory Map for Example 6–14

<table>
<tr><td>Longword number</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td><td colspan="4">4</td><td colspan="4">5</td><td colspan="4">6</td><td>. . .</td></tr>
<tr><td>Byte number</td><td>01</td><td>02</td><td>03</td><td>04</td><td>05</td><td>06</td><td>07</td><td>08</td><td>09</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>. . .</td></tr>
<tr><td>Level 01</td><td colspan="24">A–RECORD</td><td>. . .</td></tr>
<tr><td>Level 03</td><td colspan="10">ITEM 1</td><td colspan="10">ITEM 1</td><td colspan="4">ITEM 1</td><td>. . .</td></tr>
<tr><td>Byte contents at initialization time</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>C</td><td>C</td><td>C</td><td>C</td><td>. . .</td></tr>
</table>

ZK–6049–GE

When redefining or initializing table elements, allow space for any fill bytes that might be added due to synchronization. You do not have to initialize fill bytes, but you can do so. If you initialize fill bytes to an uncommon value, you can use them as a debugging aid in situations where a Procedure Division statement refers to the record level preceding the OCCURS clause, or to another record redefining that level.

You can also initialize tables at run time. To initialize tables at run time, use the INITIALIZE statement. This statement allows you to initialize all occurrences of a table element to the same value. For more information on the INITIALIZE statement, refer to the *VAX COBOL Reference Manual.*

Sometimes the length and format of table items are such that they are best initialized using Procedure Division statements such as a MOVE statement to send a value to the table.

6.4 Accessing Table Elements

Once tables have been created using the OCCURS clause, the program must have a method of accessing the individual elements of those tables. Subscripting and indexing are the two methods VAX COBOL provides for accessing individual table elements. To refer to a particular element within a table, follow the name of that element with a subscript or index enclosed in parentheses. The following sections describe how to identify and access table elements using subscripts and indexes.

6.4.1 Subscripting

A subscript can be an integer literal, an arithmetic expression, a data name, or a subscripted data name that has an integer value. The integer value represents the desired element of the table. An integer value of 3, for example, refers to the third element of a table.

6.4.2 Subscripting with Literals

A literal subscript is an integer value, enclosed in parentheses, that represents the desired table element. In Example 6–15, the literal subscript (2) in the MOVE instruction moves the contents of the second element of A-TABLE to I-RECORD.

Example 6–15: Using a Literal Subscript to Access a Table

```
Table Description:

      01 A-TABLE.
         03 A-GROUP PIC X(5)
            OCCURS 10 TIMES.

Instruction:

         MOVE A-GROUP(2) TO I-RECORD.
```

If the table is multidimensional, follow the data name of the desired data item with a list of subscripts, one for each OCCURS clause to which the item is subordinate. The first subscript in the list applies to the first OCCURS clause to which that item is subordinate. This is the most inclusive level, and is represented by A-GROUP in Example 6–16. The second subscript applies to the next most inclusive level and is represented by ITEM3 in the example. Finally, the third subscript applies to the least inclusive level, represented by ITEM5. (Note that VAX COBOL can have 48 subscripts that follow the pattern in Example 6–15.)

In Example 6–16, the subscripts (2,11,3) in the MOVE statements move the third occurrence of ITEM5 in the eleventh repetition of ITEM3 in the second repetition of A-GROUP to I-FIELD5. ITEM5(1,1,1) refers to the first occurrence of ITEM5 in the table, and ITEM5(5,20,4) refers to the last occurrence of ITEM5.

Example 6–16: Subscripting a Multidimensional Table

```
Table Description:

      01 A-TABLE.
         03 A-GROUP OCCURS 5 TIMES.
            05 ITEM1           PIC X.
            05 ITEM2           PIC 99 COMP OCCURS 20 TIMES.
            05 ITEM3 OCCURS 20 TIMES.
               07 ITEM4        PIC X.
               07 ITEM5        PIC XX OCCURS 4 TIMES.
      01 I-FIELD5              PIC XX.

Procedural Instruction:

            MOVE ITEM5(2, 11, 3) TO I-FIELD5.
```

NOTE

Because ITEM5 is not subordinate to ITEM2, an occurrence number for ITEM2 is not permitted in the subscript list (when referencing ITEM3, ITEM4, or ITEM5). The ninth occurrence of ITEM2 in the fifth occurrence of A-GROUP would be selected by ITEM2(5,9).

Table 6–1 shows the subscripting rules that apply to Example 6–16.

Table 6–1: Subscripting Rules for a Multidimensional Table

Name of Item	Number of Subscripts Required to Refer to the Name Item	Size of Item in Bytes (Each Occurrence)
A-TABLE	NONE	1105
A-GROUP	ONE	221
ITEM1	ONE	1
ITEM2	TWO	2
ITEM3	TWO	9
ITEM4	TWO	1
ITEM5	THREE	2

6.4.3 Subscripting with Data Names

You can also use data names to specify subscripts. To use a data name as a subscript, define it with COMP, COMP-1, COMP-2, COMP-3, or DISPLAY usage and with a numeric integer value. If the data name is signed, the sign must be positive at the time the data name is used as a subscript.

A data name that is a subscript can also be subscripted—for example, A(B(C)). Note that for efficiency your subscripts should be S9(5) to S9(9) COMP.

The sample subscripts and data names used in Table 6–2 refer to the table defined in Example 6–16.

Table 6–2: Subscripting with Data Names

Data Descriptions of Subscript Data Names	Procedural Instructions
01 SUB1 PIC 99 USAGE DISPLAY.	MOVE 2 TO SUB1.
01 SUB2 PIC S9(9) USAGE COMP.	MOVE 11 TO SUB2.
01 SUB3 PIC S99.	MOVE 3 TO SUB3.
	MOVE ITEM5(SUB1,SUB2,SUB3) TO I-FIELD5.

6.4.4 Subscripting with Indexes

The same rules apply for specifying indexes as for subscripts, except that the index must be named in the INDEXED BY phrase of the OCCURS clause.

You cannot access index items as normal data items; that is, you cannot use them, redefine them, or write them to a file. However, the SET statement can change their values, and relation tests can examine their values. The index integer you specify in the SET statement must be in the range of one to the integer value in the OCCURS clause. The sample MOVE statement shown in Example 6–17 moves the contents of the third element of A-GROUP to I-FIELD.

Example 6–17: Subscripting with Index Name Items

```
Table Description:

      01 A-TABLE
         03 A-GROUP OCCURS 5 TIMES
            INDEXED BY IND-NAME.
            .
            .
            .
      01 I-FIELD     PIC X(5).

Procedural Instructions:

      SET IND-NAME TO 3.
      MOVE A-GROUP(IND-NAME) TO I-FIELD.
```

NOTE

VAX COBOL initializes the value of all indexes to 1. Initializing indexes is an extension to the ANSI COBOL standard. Users who write COBOL programs that must adhere to standard COBOL should not rely on this feature.

6.4.5 Relative Indexing

To perform relative indexing when referring to a table element, you follow the index name with a plus or minus sign and an integer literal. Although it is easy to use, relative indexing generates additional overhead each time a table element is referenced in this way. The run-time overhead for relative indexing of variable-length tables is significantly greater than that required for fixed-length tables. If any of the range checks reveals an out-of-range index value, program execution terminates, and an error message is issued. You can use the /CHECK command line qualifier to check the range when you compile the program. (See Chapter 2 for more information.)

The following sample MOVE statement moves the fourth repetition of A-GROUP to I-FIELD:

```
SET IND-NAME TO 1.
MOVE A-GROUP(IND-NAME + 3) TO I-FIELD.
```

6.4.6 Index Data Items

Often a program requires that the value of an index be stored outside of that item. VAX COBOL provides the index data item to fulfill this requirement.

Index data items are stored as longword COMP items and must be declared with a USAGE IS INDEX phrase in the item description. Index data items can be explicitly modified only with the SET statement.

6.4.7 Assigning Index Values Using the SET Statement

The SET statement assigns values to indexes associated with tables, so that you can reference particular table elements. Two of the six VAX COBOL SET statement formats are available to you, and are discussed in the following sections. (All six formats are shown in the *VAX COBOL Reference Manual.*)

6.4.7.1 Assigning an Integer Index Value with a SET Statement

When you use the SET statement, the index is set to the value you specify. The most straightforward use of the SET statement is to set an index name to an integer literal value. This example assigns a value of 5 to IND-5:

```
SET IND-5 TO 5.
```

You can also set an index name to an integer data item. For example:

```
SET INDEX-A TO COUNT-1.
```

More than one index can be set with a single SET statement. For example:

```
SET TAB1-IND TAB2-IND TO 15.
```

Table indexes specified in INDEXED BY phrases can be displayed by using the WITH CONVERSION option with the VAX COBOL DISPLAY statement. Also, you can display, move, and manipulate the value of the table index with an index data item. You do this by setting an index data item to the present value of an index. You could, for example, set an index data item and then display its value as shown in the following example:

```
SET INDEX-ITEM TO TAB-IND.
         .
         .
         .
DISPLAY INDEX-ITEM WITH CONVERSION.
```

6.4.7.2 Incrementing an Index Value with the SET Statement

You can use the SET statement with the UP BY/DOWN BY clause to arithmetically alter the value of a index. A numeric literal is added to (UP BY) or subtracted from (DOWN BY) a table index. For example:

```
SET TABLE-INDEX UP BY 12.

SET TABLE-INDEX DOWN BY 5.
```

6.4.8 Identifying Table Elements Using the SEARCH Statement

The SEARCH statement is used to search a table for an element that satisfies a known condition. The statement provides for sequential and binary searches, which are described in the following sections.

6.4.8.1 Implementing a Sequential Search

The SEARCH statement allows you to perform a sequential search of a table. The OCCURS clause of the table description entry must contain the INDEXED BY phrase. If more than one index is specified in the INDEXED BY phrase, the first index is the controlling index for the table search unless you specify otherwise in the SEARCH statement.

The search begins at the current index setting and progresses through the table, checking each element against the conditional expression. The index is incremented by 1 as each element is checked. If the conditional expression is true, the associated imperative statement executes; otherwise, program control passes to the next procedural sentence. This terminates the search, and the index points to the current table element that satisfied the conditional expression.

If no table element is found that satisfies the conditional expression, program control passes to the AT END exit path; otherwise, program control passes to the next procedural sentence.

You can use the optional VARYING phrase of the SEARCH statement by specifying any of the following:

- VARYING index name associated with table search
- VARYING index data item or integer data item
- VARYING index name not associated with table search

Regardless of which method you use, the index specified in the INDEXED BY phrase of the table being searched is incremented. This controlling index, when compared against the allowable number of occurrences in the table, dictates the permissible search range. When the search terminates, either successfully or unsuccessfully, the index remains at its current setting. At this point, you can reference the data in the table element pointed to by the index, unless the AT END condition is true. If the AT END condition is true, and if the /CHECK qualifier has been specified, the compiler issues a run-time error message indicating that the subscript is out of range.

When you vary an index associated with the table being searched, the index name can be any index you specify in the INDEXED BY phrase. It bccomes the controlling index for the search and is the only index incremented. Example 6–18 and Example 6–20 show how to vary an index other than the first index.

When you vary an index data item or an integer data item, either the index data item or the integer data item is incremented. The first index name you specify in the INDEXED BY phrase of the table being searched becomes the controlling index and is also incremented. The index data item or the integer data item you vary does not function as an index; it merely allows you to maintain an additional pointer to elements within a table. Example 6–18 and Example 6–21 show how to vary an index data item or an integer data item.

When you vary an index associated with a table other than the one you are searching, the controlling index is the first index you specify in the INDEXED BY phrase of the table you are searching. Each time the controlling index is incremented, the index you specify in the VARYING phrase is incremented. In this manner, you can search two tables in synchronization. Example 6–18 and Example 6–22 show how to vary an index associated with a table other than the one you are searching.

When you omit the VARYING phrase, the first index you specify in the INDEXED BY phrase becomes the controlling index. Only this index is incremented during a serial search. Example 6–18 and Example 6–23 show how to perform a serial search without using the VARYING phrase.

6.4.8.2 Implementing a Binary Search

You can use the SEARCH statement to perform a nonsequential (binary) table search.

To perform a binary search, you must specify an index name in the INDEXED BY phrase and a search key in the KEY IS phrase of the OCCURS clause of the table being searched.

A binary search depends on the ASCENDING/DESCENDING KEY attributes. If you specify an ASCENDING KEY, the data in the table must either be stored in ascending order or sorted in ascending order prior to the search. For a DESCENDING KEY, data must be stored or sorted in descending order prior to the search.

During a binary search, the first (or only) index you specify in the INDEXED BY phrase of the OCCURS clause of the table being searched is the controlling index. You do not have to initialize an index in a binary search because index manipulation is automatic.

In addition to being generally faster than a sequential search, a binary search allows multiple equality checks.

The following search sequence lists the capabilities of a binary search. At program execution time, the system:

1. Examines the range of permissible index values, selects the median value, and assigns this value to the index.
2. Checks for equality in WHEN and AND clauses.
3. Terminates the search if all equality statements are true. If you use the imperative statement after the final equality clause, that statement executes; otherwise, program control passes to the next procedural sentence, the search exits, and the index retains its current value.
4. Takes the following actions if the equality test of a table element is false:
    a. Executes the imperative statement associated with the AT END statement (if present) when all table elements have been tested. If there is no AT END statement, program control passes to the next procedural statement.
    b. Determines which half of the table is to be eliminated from further consideration. This is based on whether the key being tested was specified as ASCENDING or DESCENDING and whether the test failed because of a greater-than or less-than comparison. For example, if the key values are stored in ascending order, and the median table element being tested is greater than the value of the argument, then all key elements following the one being tested must also be greater. Therefore, the upper half of the table is removed from further consideration and the search continues at the median point of the lower half.
    c. Begins processing all over again at step 1.

A useful variation of the binary search is that of specifying multiple search keys. Multiple search keys allow you to select a specified table element from among several elements that have duplicate low-order keys. An example is a telephone listing where several people have the same last and first names—but different middle initials. All specified keys must be either ascending or descending. Example 6–24 shows how to use multiple search keys.

The table in Example 6–18 is followed by several examples (Example 6–19, Example 6–20, Example 6–21, Example 6–22, and Example 6–23) of how to search it.

Example 6–18: Sample Table

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TEMP-INDEX                          PIC 9(5) USAGE IS INDEX.
01  FED-TAX-TABLES.
    02  ALLOWANCE-DATA.
        03  FILLER                      PIC X(70) VALUE
            "0101440
-           "0202880
-           "0304320
-           "0405760
-           "0507200
-           "0608640
-           "0710080
-           "0811520
-           "0912960
-           "1014400".
    02  ALLOWANCE-TABLE REDEFINES ALLOWANCE-DATA.
        03  FED-ALLOWANCES OCCURS 10 TIMES
            ASCENDING KEY IS ALLOWANCE-NUMBER
            INDEXED BY IND-1.
            04  ALLOWANCE-NUMBER        PIC XX.
            04  ALLOWANCE               PIC 99999.

    02 SINGLES-DEDUCTION-DATA.
        03  FILLER                      PIC X(112) VALUE
            "0250006700000016
-           "0670011500067220
-           "1150018300163223
-           "1830024000319621
-           "2400027900439326
-           "2790034600540730
-           "3460099999741736".
   02   SINGLE-DEDUCTION-TABLE REDEFINES SINGLES-DEDUCTION-DATA.
        03  SINGLES-TABLE OCCURS 7 TIMES
            ASCENDING KEY IS S-MIN-RANGE S-MAX-RANGE
            INDEXED BY IND-2, TEMP-INDEX.
            04  S-MIN-RANGE             PIC 99999.
            04  S-MAX-RANGE             PIC 99999.
            04  S-TAX                   PIC 9999.
            04  S-PERCENT               PIC V99.

    02  MARRIED-DEDUCTION-DATA.
        03  FILLER                      PIC X(119) VALUE
            "04800096000000017
-           "09600173000081620
-           "17300264000235617
-           "26400346000390325
-           "34600433000595328
-           "43300500000838932
-           "50000999991053336".
    02  MARRIED-DEDUCTION-TABLE REDEFINES MARRIED-DEDUCTION-DATA.
        03 MARRIED-TABLE OCCURS 7 TIMES
           ASCENDING KEY IS M-MIN-RANGE M-MAX-RANGE
           INDEXED BY IND-0, IND-3.
           04  M-MIN-RANGE              PIC 99999.
           04  M-MAX-RANGE              PIC 99999.
           04  M-TAX                    PIC 99999.
           04  M-PERCENT                PIC V99.
```

Example 6–19 shows how to perform a serial search.

Example 6–19: A Serial Search

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
SINGLE.
       IF TAXABLE-INCOME < 02500
               GO TO END-FED-COMP.
       SET TEMP-INDEX TO 1.
       SEARCH SINGLES-TABLE VARYING TEMP-INDEX AT END
               GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(TEMP-INDEX)
               MOVE S-TAX(TEMP-INDEX) TO FED-TAX-DEDUCTION
          WHEN TAXABLE-INCOME < S-MAX-RANGE(TEMP-INDEX)
               COMPUTE FED-TAX-DEDUCTION =
                   S-TAX(TEMP-INDEX) + (TAXABLE-INCOME - S-TAX(TEMP-INDEX)) *
                   S-PERCENT(TEMP-INDEX).
```

Example 6–20: Using SEARCH and Varying an Index Other Than the First Index

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
MARRIED.
       IF TAXABLE-INCOME < 04800
               MOVE ZEROS TO FED-TAX-DEDUCTION
               GO TO END-FED-COMP.
       SET IND-3 TO 1.
       SEARCH MARRIED-TABLE VARYING IND-3 AT END
               GO TO TABLE-3-ERROR
          WHEN TAXABLE-INCOME = M-MIN-RANGE(IND-3)
               MOVE M-TAX(IND-3) TO FED-TAX-DEDUCTION
          WHEN TAXABLE-INCOME < M-MAX-RANGE(IND-3)
               COMPUTE FED-TAX-DEDUCTION =
                   M-TAX(IND-3) + (TAXABLE-INCOME - M-TAX(IND-3)) *
                   M-PERCENT(IND-3).
```

Example 6–21: Using SEARCH and Varying an Index Data Item

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
SINGLE.
       IF TAXABLE-INCOME < 02500
               GO TO END-FED-COMP.
       SET TEMP-INDEX TO 1.
       SEARCH SINGLES-TABLE VARYING TEMP-INDEX AT END
               GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(TEMP-INDEX)
               MOVE S-TAX(TEMP-INDEX) TO FED-TAX-DEDUCTION
          WHEN TAXABLE-INCOME < S-MAX-RANGE(TEMP-INDEX)
               MOVE S-TAX(TEMP-INDEX) TO FED-TAX-DEDUCTION
               SUBTRACT S-MIN-RANGE(TEMP-INDEX) FROM TAXABLE-INCOME
               MULTIPLY TAXABLE-INCOME BY S-PERCENT(TEMP-INDEX) ROUNDED
               ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION.
```

Example 6–22: Using SEARCH and Varying an Index Not Associated with the Target Table

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
SINGLE.
        IF TAXABLE-INCOME < 02500
               GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING IND-0 AT END
               GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
               MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
               MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION
               SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
               MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
               ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION.
```

Example 6–23: Doing a Serial Search Without Using the VARYING Phrase

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
FED-DEDUCT-COMPUTATION.
          SET IND-1 TO 1.
          SEARCH FED-ALLOWANCES AT END
                 GO TO TABLE-1-ERROR
            WHEN ALLOWANCE-NUMBER(IND-1) = NR-DEPENDENTS
                 SUBTRACT ALLOWANCE(IND-1) FROM GROSS-WAGE
                     GIVING TAXABLE-INCOME ROUNDED.
          IF MARITAL-STATUS = "M"
                 GO TO MARRIED.
```

Example 6–24: A Multiple-Key Binary Search

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MULTI-KEY-SEARCH.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DIRECTORY-TABLE.
   05 NAMES-NUMBERS.
      10 FILLER                PIC X(30)
         VALUE "SMILEY    HAPPY     T.213-4332".
      10 FILLER                PIC X(30)
         VALUE "SMITH     ALAN      C.881-4987".
      10 FILLER                PIC X(30)
         VALUE "SMITH     CHARLES   J.345-2398".
      10 FILLER                PIC X(30)
         VALUE "SMITH     FREDERICK   745-0223".
      10 FILLER                PIC X(30)
         VALUE "SMITH     HARRY     C.573-3306".
      10 FILLER                PIC X(30)
         VALUE "SMITH     HARRY     J.295-3485".
      10 FILLER                PIC X(30)
         VALUE "SMITH     LARRY     X.976-5504".
      10 FILLER                PIC X(30)
         VALUE "SMITHWOOD ALBERT    J.349-9927".
   05 PHONE-DIRECTORY-TABLE REDEFINES NAMES-NUMBERS OCCURS 8 TIMES
                                 ASCENDING KEY IS LAST-NAME
                                                  FIRST-NAME
                                                  MID-INIT
                                 INDEXED BY DIR-INDX.
         15 LAST-NAME          PIC X(10).
         15 FIRST-NAME         PIC X(10).
         15 MID-INIT           PIC XX.
         15  PHONE-NUM         PIC X(8).
PROCEDURE DIVISION.
MULTI-KEY-BINARY-SEARCH.
    SEARCH ALL PHONE-DIRECTORY-TABLE
           WHEN LAST-NAME(DIR-INDX) = "SMITH"
           AND FIRST-NAME(DIR-INDX) = "HARRY"
           AND MID-INIT(DIR-INDX) = "J."
               NEXT SENTENCE.
```

(continued on next page)

Example 6–24 (Cont.): A Multiple-Key Binary Search

```
DISPLAY-RESULTS.
    DISPLAY LAST-NAME(DIR-INDX)","
            FIRST-NAME(DIR-INDX)
            MID-INIT(DIR-INDX) " "
            PHONE-NUM(DIR-INDX).
```

Chapter 7

Using the STRING, UNSTRING, and INSPECT Statements

This chapter describes the use of the STRING, UNSTRING, and INSPECT statements.

7.1 Concatenating Data Using the STRING Statement

The STRING statement concatenates the contents of one or more sending items into a single receiving item.

The statement has many forms; the simplest is equivalent in function to a nonnumeric MOVE statement. Consider the following example:

```
STRING FIELD1 DELIMITED BY SIZE INTO FIELD2.
```

If the two items are the same size, or if the sending item (FIELD1) is larger, the statement is equivalent to the following statement:

```
MOVE FIELD1 TO FIELD2.
```

If the sending item of the string is shorter than the receiving item, the compiler does not replace unused positions in the receiving item with spaces. Thus, the STRING statement can leave some portion of the receiving item unchanged.

The receiving item of the string must be an elementary alphanumeric item with no JUSTIFIED clause or editing characters in its description. Thus, the data movement of the STRING statement always fills the receiving item with the sending item from left to right and with no editing insertions.

7.1.1 Multiple Sending Items

The STRING statement can concatenate a series of sending items into one receiving item. Consider the following example:

```
STRING FIELD1A FIELD1B FIELD1C DELIMITED BY SIZE
                               INTO FIELD2.
```

In this sample STRING statement, FIELD1A, FIELD1B, and FIELD1C are all sending items. The compiler moves them to the receiving item (FIELD2) in the order in which they appear in the statement, from left to right, resulting in the concatenation of their values.

If FIELD2 is not large enough to hold all three items, the operation stops when it is full. If the operation stops while moving one of the sending items, the compiler ignores the remaining characters of that item and any other sending items not yet processed. For example, if FIELD2 is filled while it is receiving FIELD1B, the compiler ignores the rest of FIELD1B and all of FIELD1C.

If the sending items do not fill the receiving item, the operation stops when the last character of the last sending item (FIELD1C) is moved. It does not alter the contents nor space-fill the remaining character positions of the receiving item.

The sending items can be nonnumeric literals and figurative constants (except for ALL literal). Example 7–1 sets up an address label by stringing the data items CITY, STATE, and ZIP into ADDRESS-LINE. The figurative constant SPACE and the literal period (.) are used to separate the information.

Example 7–1: Using the STRING Statement and Literals

```
01 ADDRESS-GROUP.
   03 CITY           PIC X(20).
   03 STATE          PIC XX.
   03 ZIP            PIC X(5).
01 ADDRESS-LINE      PIC X(31).
      .
      .
      .
   STRING CITY SPACE STATE ". " SPACE ZIP
        DELIMITED BY SIZE INTO ADDRESS-LINE.
```

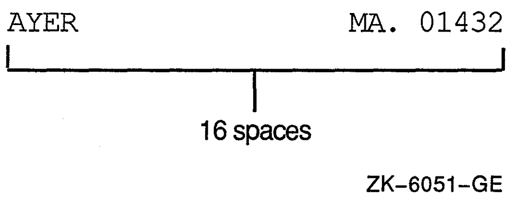
7.1.2 Using the DELIMITED BY Phrase

Although the sending items of the STRING statement are fixed in size at compile time, they are frequently filled with spaces. For example, if a 20-character city item contains the text MAYNARD followed by 13 spaces, the STRING statement using the DELIMITED BY SIZE phrase would move the text (MAYNARD) and the unwanted 13 spaces (assuming the receiving item is at least 20 characters long). The DELIMITED BY phrase, written with a data name or literal, eliminates this problem.

The delimiter can be a literal, a data item, a figurative constant, or the word SIZE. It cannot, however, be ALL literal, since ALL literal has an indefinite length. When the phrase contains the word SIZE, the compiler moves each sending item in total, until it either exhausts the characters in the sending item or fills the receiving item.

If you use the code in Example 7–1, and CITY is a 20-character item, the result of the STRING operation might look like Figure 7–1.

Figure 7–1: Results of the STRING Operation



A more attractive and readable report can be produced by having the STRING operation produce this line:

```
AYER, MA. 01432
```

To accomplish this, use the figurative constant SPACE as a delimiter on the sending item:

```
MOVE 1 TO P.
STRING CITY DELIMITED BY SPACE
        INTO ADDRESS-LINE WITH POINTER P.
STRING ", " STATE ". " ZIP
        DELIMITED BY SIZE
        INTO ADDRESS-LINE WITH POINTER P.
```

This example makes use of the POINTER phrase (see Section 7.1.3). The first STRING statement moves data characters until it encounters a space character—a match of the delimiter SPACE. The second STRING statement supplies the literal, the 2-character STATE item, another literal, and the 5-character ZIP item.

The delimiter can be varied for each item within a single STRING statement by repeating the DELIMITED BY phrase after each of the sending item names to which it applies. Thus, the shorter STRING statement in the following example has the same effect as the two STRING statements in the preceding example. (Placing the operands on separate source lines has no effect on the operation of the statement, but it improves program readability and simplifies debugging.)

```
STRING CITY DELIMITED BY SPACE
        ", " STATE ". "
        ZIP DELIMITED BY SIZE
        INTO ADDRESS-LINE.
```

The sample STRING statement cannot handle 2-word city names, such as San Francisco, since the compiler considers the space between the two words as a match for the delimiter SPACE. A longer delimiter, such as two or three spaces (nonnumeric literal), can solve this problem. Only when a sequence of characters matches the delimiter does the movement stop for that data item. With a 2-character delimiter, the same statement can be rewritten in a simpler form:

```
STRING CITY ", " STATE ". " ZIP
        DELIMITED BY "  " INTO ADDRESS-LINE.
```

Since only the CITY item contains two consecutive spaces, the delimiter's search of the other items will always be unsuccessful, and the effect is the same as moving the full item (delimiting by SIZE).

Data movement under control of a data name or literal generally executes more slowly than data movement delimited by SIZE.

Remember, the remainder of the receiving item is not space-filled, as with a MOVE statement. If ADDRESS-LINE is to be printed on a mailing label, for example, the STRING statement should be preceded by the statement:

```
MOVE SPACES TO ADDRESS-LINE.
```

This statement guarantees a space-fill to the right of the concatenated result. Alternatively, the last item concatenated by the STRING statement can be an item previously set to SPACES. This sending item must either be moved under control of a delimiter other than SPACE or use the value of POINTER and reference modification.

7.1.3 Using the POINTER Phrase

Although the STRING statement normally starts scanning at the leftmost position of the receiving item, the POINTER phrase makes it possible to start scanning at another point within the item. The scanning, however, continues left to right. Consider the following example:

```
MOVE 5 TO P.
STRING FIELD1A FIELD1B DELIMITED BY SIZE
        INTO FIELD2 WITH POINTER P.
```

The value of P determines the starting character position in the receiving item. In this example, the 5 in P causes the program to move the first character of FIELD1A into character position 5 of FIELD2 (the leftmost character position of the receiving item is character position 1), and leave positions 1 to 4 unchanged.

When the STRING operation is complete, P points to one character position beyond the last character replaced in the receiving item. If FIELD1A and FIELD1B are both four characters long, P contains a value of 13 (5+4+4) when the operation is complete (assuming that FIELD2 is at least 13 characters long).

7.1.4 Using the OVERFLOW Phrase

When the SIZE option of the DELIMITED BY phrase controls the STRING operation, and the pointer value is either known or the POINTER phrase is not used, you can add the PICTURE sizes of sending items together at program development time to see if the receiving item is large enough to hold the sending items. However, if the DELIMITED BY phrase contains a literal or an identifier, or if the pointer value is not predictable, it can be difficult to tell whether or not the size of the receiving item will be large enough at run time. If the size of the receiving item is not large enough, an overflow can occur.

An overflow occurs when the receiving item is full and the program is either about to move a character from a sending item or is considering a new sending item. Overflow can also occur if, during the initialization of the statement, the pointer contains a value that is either less than 1 or greater than the length of the receiving item. In this case, the program moves no data to the receiving item and terminates the operation immediately.

The ON OVERFLOW phrase at the end of the STRING statement tests for an overflow condition:

```
STRING FIELD1A FIELD1B DELIMITED BY "C"
        INTO FIELD2 WITH POINTER PNTR
        ON OVERFLOW GO TO 200-STRING-OVERFLOW.
```

The ON OVERFLOW phrase cannot distinguish the overflow caused by a bad initial value in the pointer from the overflow caused by a receiving item that is too short. Only a separate test preceding the STRING statement can distinguish between the two.

Additionally, even if an overflow condition does not exist, you can use the NOT ON OVERFLOW phrase to branch to or execute other sections of code.

Example 7–2 illustrates the overflow condition.

Example 7–2: Sample Overflow Condition

```
DATA DIVISION.
      .
      .
      .
01 FIELD1 PIC XXX VALUE "ABC".
01 FIELD2 PIC XXXX.
PROCEDURE DIVISION.
            .
            .
            .
1.     STRING FIELD1 QUOTE DELIMITED BY SIZE INTO FIELD2
                                             ON OVERFLOW....
2.     STRING FIELD1 FIELD1 DELIMITED BY SIZE INTO FIELD2
                                             ON OVERFLOW....
3.     STRING FIELD1 FIELD1 DELIMITED BY "C" INTO FIELD2
                                             ON OVERFLOW....
4.     STRING FIELD1 FIELD1 FIELD1 FIELD1
            DELIMITED BY "B" INTO FIELD2 ON OVERFLOW....
5.     STRING FIELD1 FIELD1 "D" DELIMITED BY "C"
            INTO FIELD2 ON OVERFLOW....
6.     MOVE 2 TO P.

       MOVE ALL QUOTES TO FIELD2.

       STRING FIELD1 "AC" DELIMITED BY "C"
            INTO FIELD2 WITH POINTER P ON OVERFLOW....
```

The STRING statement numbers in Example 7–2 point to the line number results shown in Table 7–1.

Table 7–1: Results of Sample Overflow Statements

Value of FIELD2 After the STRING Operation	Overflow?
1. ABC"	NO
2. ABCA	YES
3. ABAB	NO
4. AAAA	NO
5. ABAB	YES
6. "ABA	NO

7.1.5 Common STRING Statement Errors

The following are common errors made when writing STRING statements:

- Using the word TO instead of INTO
- Failing to include the DELIMITED BY SIZE phrase
- Failing to initialize the pointer
- Initializing the pointer to 0 instead of 1
- Permitting the pointer to get out of range (negative or larger than the size of the receiving field)
- Failing to provide for space-filling of the receiving item when it is desirable

- Using the pointer as a subscript without fully understanding subscript evaluation

7.2 Separating Data Using the UNSTRING Statement

The UNSTRING statement disperses the contents of a single sending item into one or more receiving items.

The statement has many forms; the simplest is equivalent in function to a nonnumeric MOVE statement. Consider the following example:

```
UNSTRING FIELD1 INTO FIELD2.
```

Regardless of the relative sizes of the two items, the sample statement is equivalent to the following MOVE statement:

```
MOVE FIELD1 TO FIELD2.
```

The sending item (FIELD1) can be either (1) a group item, or (2) an alphanumeric or alphanumeric edited elementary item. The receiving item (FIELD2) can be alphabetic, alphanumeric, or numeric, but it cannot specify any type of editing.

If the receiving item is numeric, it must be DISPLAY usage. The PICTURE character-string of a numeric receiving item can contain any of the legal numeric description characters except P and the editing characters. The UNSTRING statement moves the sending item to the numeric receiving item as if the sending item had been described as an unsigned integer. It automatically truncates or zero-fills as required.

If the receiving item is not numeric, the statement follows the rules for elementary nonnumeric MOVE statements. It left-justifies the data in the receiving item, truncating or space-filling as required. If the data description of the receiving item contains a JUSTIFIED clause, the compiler right-justifies the data, truncating or space-filling to the left as required.

7.2.1 Multiple Receiving Items

The UNSTRING statement can disperse one sending item into several receiving items. Consider the following example of the UNSTRING statement written with multiple receiving items:

```
UNSTRING FIELD1 INTO FIELD2A FIELD2B FIELD2C.
```

The compiler-generated code performs the UNSTRING operation by scanning across FIELD1, the sending item, from left to right. When the number of characters scanned equals the number of characters in the receiving item, the scanned characters are moved into that item and the next group of characters is scanned for the next receiving item.

If each of the receiving items in the preceding example (FIELD2A, FIELD2B, and FIELD2C) is 5 characters long, and FIELD1 is 15 characters long, FIELD1 is scanned until the number of characters scanned equals the size of FIELD2A (5). Those first five characters are moved to FIELD2A, and scanning is resumed at the sixth character position in FIELD1. Next, FIELD1 is scanned from character position 6, until the number of scanned characters equals the size of FIELD2B (five). The sixth through the tenth characters are then moved to FIELD2B, and the scanner is set to the next (eleventh) character position in FIELD1. For the last move in this example, characters 11 to 15 of FIELD1 are moved into FIELD2C.

Each data movement acts as an individual MOVE statement, the sending item of which is an alphanumeric item equal in size to the receiving item. If the receiving item is numeric, the move operation converts the data to numeric form. For example, consider what would happen if the items under discussion had the data descriptions and were manipulating the values shown in Table 7–2.

Table 7–2: Values Moved into the Receiving Items Based on the Sending Item Value

		FIELD2B	
FIELD1		**PIC S9(5)**	
PIC X(15)	**FIELD2A**	**LEADING**	**FIELD2C**
VALUE IS:	**PIC X(5)**	**SEPARATE**	**PIC S999V99**
ABCDE1234512345	ABCDE	+12345	3450{
XXXXX0000100123	XXXXX	+00001	1230{

FIELD2A is an alphanumeric item. Therefore, the statement simply conducts an elementary nonnumeric move with the first five characters.

FIELD2B, however, has a leading separate sign that is not included in its size. Thus, the compiler moves only five numeric characters and generates a positive sign (+) in the separate sign position.

FIELD2C has an implied decimal point with two character positions to the right of it, plus an overpunched sign on the low-order digit. The sending item should supply five numeric digits. However, since the sending item is alphanumeric, the compiler treats it as an unsigned integer; it truncates the two high-order digits and supplies two zero digits for the decimal positions. Furthermore, it supplies a positive overpunch sign, making the low-order digit a +0 (ASCII {). There is no way to have the UNSTRING statement recognize a sign character or a decimal point in the sending item in a single statement.

If the sending item is shorter than the sum of the sizes of the receiving items, the compiler ignores the remaining receiving items. If the compiler reaches the end of the sending item before it reaches the end of one of the receiving items, it moves the scanned characters into that receiving item. It either left-justifies and fills the remaining character positions with spaces for alphanumeric data, or else it decimal point-aligns and zero-fills the remaining character positions for numeric data.

Consider the following statement with reference to the corresponding PICTURE character-strings and values in Table 7–3:

```
UNSTRING FIELD1 INTO FIELD2A FIELD2B.
```

FIELD2A is a 3-character alphanumeric item. It receives the first three characters of FIELD1 (ABC) in every operation. FIELD2B, however, runs out of characters every time before filling, as Table 7–3 illustrates.

Table 7–3: Handling a Short Sending Item

FIELD1 PIC X(6) VALUE IS:	FIELD2B PICTURE IS:	FIELD2B Value After UNSTRING Operation
ABCDEF	XXXXX	DEF
	S99999	0024F
ABC246	S9V999	600{
	S9999 LEADING SEPARATE	+0246

7.2.2 Controlling Moved Data Using the DELIMITED BY Phrase

The size of the data to be moved can be controlled by a delimiter, rather than by the size of the receiving item. The DELIMITED BY phrase supplies the delimiter characters.

UNSTRING delimiters can be literals, figurative constants (including ALL literal), or identifiers (identifiers can even be subscripted data names). This section discusses the use of these three types of delimiters. Subsequent sections cover multiple delimiters, the COUNT phrase, and the DELIMITER phrase.

Consider the following sample UNSTRING statement with the figurative constant SPACE as a delimiter:

```
UNSTRING FIELD1 DELIMITED BY SPACE
         INTO FIELD2.
```

In this example, the compiler scans the sending item (FIELD1), searching for a space character. If it encounters a space, it moves all of the scanned (nonspace) characters that precede that space to the receiving item (FIELD2). If it finds no space character, it moves the entire sending item. When the compiler has determined the size of the sending item, it moves the contents of that item following the rules for the MOVE statement, truncating or zero-filling as required.

Table 7–4 shows the results of the following UNSTRING operation that uses a literal asterisk delimiter:

```
UNSTRING FIELD1 DELIMITED BY "*"
         INTO FIELD2.
```

Table 7–4: Results of Delimiting with an Asterisk

FIELD1 PIC X(6) VALUE IS:	FIELD2 PICTURE IS:	FIELD2 Value After UNSTRING
	XXX	ABC
ABCDEF	X(7)	ABCDEF
	XXX JUSTIFIED	DEF
******	XXX	sss
*ABCDE	XXX	sss

Legend: s = space

(continued on next page)

Table 7–4 (Cont.): Results of Delimiting with an Asterisk

FIELD1 PIC X(6) VALUE IS:	FIELD2 PICTURE IS:	FIELD2 Value After UNSTRING
A*****	XXX JUSTIFIED	ssA
246***	S9999	024F
12345*	S9999 TRAILING SEPARATE	2345+
2468**	S999V9 LEADING SEPARATE	+4680
*246**	9999	0000

Legend: s = space

If the delimiter matches the first character in the sending item, the compiler considers the size of the sending item to be zero. The operation still takes place, however, and fills the receiving item with spaces (if it is nonnumeric) or zeros (if it is numeric).

A delimiter can also be applied to an UNSTRING statement that has multiple receiving items:

```
UNSTRING FIELD1 DELIMITED BY SPACE
        INTO FIELD2A FIELD2B.
```

The compiler generates code that scans FIELD1 searching for a character that matches the delimiter. If it finds a match, it moves the scanned characters to FIELD2A and sets the scanner to the next character position to the right of the character that matched. The compiler then resumes scanning FIELD1 for a character that matches the delimiter. If it finds a match, it moves all of the characters between the character that first matched the delimiter and the character that matched on the second scan, and sets the scanner to the next character position to the right of the character that matched.

The DELIMITED BY phrase handles additional items in the same manner as it handled FIELD2B.

Table 7–5 illustrates the results of the following delimited UNSTRING operation into multiple receiving items:

```
UNSTRING FIELD1 DELIMITED BY "*"
        INTO FIELD2A FIELD2B.
```

Table 7–5: Results of Delimiting Multiple Receiving Items

	Values After UNSTRING Operation	
FIELD1 PIC X(8) VALUE IS:	FIELD2A PIC X(3)	FIELD2B PIC X(3)
ABC*DEF*	ABC	DEF
ABCDE*FG	ABC	FGs

Legend: s = space

(continued on next page)

Table 7–5 (Cont.): Results of Delimiting Multiple Receiving Items

	Values After UNSTRING Operation	
FIELD1 **PIC X(8)** **VALUE IS:**	**FIELD2A** **PIC X(3)**	**FIELD2B** **PIC X(3)**
A*B****	Ass	Bss
*AB*CD**	sss	ABs
**ABCDEF	sss	sss
A*BCDEFG	Ass	BCD
ABC**DEF	ABC	sss
A******B	Ass	sss

Legend: s = space

The previous examples illustrate the limitations of a single-character delimiter. To overcome these limitations, a delimiter of more than one character or a delimiter preceded by the word ALL may be used.

Table 7–6 shows the results of the following UNSTRING operation using a 2-character delimiter:

```
UNSTRING FIELD1 DELIMITED BY "**"
         INTO FIELD2A FIELD2B.
```

Table 7–6: Results of Delimiting with Two Asterisks

	Values After UNSTRING Operation	
FIELD1 **PIC X(8)** **VALUE IS:**	**FIELD2A** **PIC XXX**	**FIELD2B** **PIC XXX** **JUSTIFIED**
ABC**DEF	ABC	DEF
A*B*C*D*	A*B	sss
AB***C*D	ABs	C*D
AB**C*D*	ABs	*D*
AB**CD**	ABs	sCD
AB***CD*	ABs	CD*
AB*****CD	ABs	sss

Legend: s = space

Unlike the STRING statement, the UNSTRING statement accepts the ALL literal as a delimiter. When the word ALL precedes the delimiter, the action of the UNSTRING statement remains essentially the same as with one delimiter until the scanning operation finds a match. At this point, the compiler scans farther, looking for additional consecutive strings of characters that also match the delimiter item. It considers the ALL delimiter to be one, two, three, or more adjacent repetitions of the delimiter item. Table 7–7 shows the results of the following UNSTRING operation using an ALL delimiter:

```
UNSTRING FIELD1 DELIMITED BY ALL "*"
         INTO FIELD2A FIELD2B.
```

Table 7–7: Results of Delimiting with ALL Asterisks

	Values After UNSTRING Operation	
FIELD1 PIC X(8) VALUE IS:	**FIELD2A PIC XXX**	**FIELD2B PIC XXX JUSTIFIED**
ABC*DEF*	ABC	DEF
ABC**DEF	ABC	DEF
A******F	Ass	ssF
A*F*****	Ass	ssF
A*CDEFG	Ass	EFG

Legend: s = space

Table 7–8 shows the results of the following UNSTRING operation that combines ALL with a 2-character delimiter:

```
UNSTRING FIELD1 DELIMITED BY ALL "**"
         INTO FIELD2A FIELD2B.
```

Table 7–8: Results of Delimiting with ALL Double Asterisks

	Values After UNSTRING Operation	
FIELD1 PIC X(8) VALUE IS:	**PIC XX**	**PIC XXX JUSTIFIED**
ABC**DEF	ABC	DEF
AB**DE**	ABs	sDE
A***D***	Ass	s*D
A*******	Ass	ss*

Legend: s = space

In addition to unchangeable delimiters, such as literals and figurative constants, delimiters can be designated by identifiers. Identifiers permit variable delimiting. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY DEL1
         INTO FIELD2A FIELD2B.
```

The data name DEL1 must be alphanumeric; it can be either a group or an elementary item. If the delimiter contains a subscript, the subscript may vary as a side effect of the UNSTRING operation.

7.2.2.1 Multiple Delimiters

The UNSTRING statement scans a sending item, searching for a match from a list of delimiters. This list can contain ALL delimiters and delimiters of various sizes. Delimiters in the list must be connected by the word OR.

The following sample statement unstrings a sending item into three receiving items. The sending item consists of three strings separated by one of the following: (1) any number of spaces, (2) a comma followed by a single space, (3) a single comma, (4) a tab character, or (5) a carriage-return character. The comma and space must precede the single comma in the list if the comma and space are to be recognized.

```
UNSTRING FIELD1 DELIMITED BY ALL SPACE
         OR ", "
         OR ","
         OR TAB
         OR CR
         INTO FIELD2A FIELD2B FIELD2C.
```

Table 7–9 shows the potential of this statement. The tab and carriage-return characters represent single-character items containing the ASCII horizontal tab and carriage-return characters.

Table 7–9: Results of Multiple Delimiters

FIELD1 **PIC X(12)**	**FIELD2A** **PIC XXX**	**FIELD2B** **PIC 9999**	**FIELD2C** **PIC XXX**
A,0,C RET	Ass	0000	Css
A TAB456, E	Ass	0456	Ess
A 3 9	Ass	0003	9ss
A TAB TAB B RET	Ass	0000	Bss
A,,C	Ass	0000	Css
ABCD, 4321,Z	ABC	4321	Zss

Legend: s = space

7.2.3 Using the COUNT Phrase

The COUNT phrase keeps track of the size of the sending string and stores the length in a user-supplied data area.

The length of a delimited sending item can vary from zero to the full length of the item. Some programs require knowledge of this length. For example, some data is truncated if it exceeds the size of the receiving item, so the program's logic requires this information.

The COUNT phrase follows the receiving item. Consider the following example:

```
UNSTRING FIELD1 DELIMITED BY ALL "*"
         INTO FIELD2A COUNT IN COUNT2A
         FIELD2B COUNT IN COUNT2B
         FIELD2C.
```

The compiler generates code that counts the number of characters between the leftmost position of FIELD1 and the first asterisk in FIELD1 and places the count into COUNT2A. The delimiter is not included in the count because it is not a part of the string. The data preceding the first asterisk is then moved into FIELD2A.

The compiler then counts the number of characters between the last contiguous asterisk in the first scan and the next asterisk in the second scan, and places the count in COUNT2B. The data between the delimiters of the second scan is moved into FIELD2B.

The third scan begins at the first character after the last contiguous asterisk in the second scan. Any data between the delimiters of this scan is moved to FIELD2C.

The COUNT phrase should be used only where it is needed. In this example, the length of the string moved to FIELD2C is not needed, so no COUNT phrase follows it.

If the receiving item is shorter than the value placed in the count item, the code truncates the sending string. If the number of integer positions in a numeric item is smaller than the value placed into the count item, high-order numeric digits have been lost. If a delimiter match is found on the first character examined, a zero is placed in the count item.

The COUNT phrase can be used only in conjunction with the DELIMITED BY phrase.

7.2.4 Saving UNSTRING Delimiters Using the DELIMITER Phrase

The DELIMITER phrase causes the actual character or characters that delimited the sending item to be stored in a user-supplied data area. This phrase is most useful when:

- The UNSTRING statement contains a delimiter list.
- Any one of the delimiters in the list might have delimited the item.
- Program logic flow depends on the delimiter match found.

By using the DELIMITER and COUNT phrases, you can make the flow of program logic dependent on both the size of the sending string and the delimiter terminating the string.

To use the DELIMITER phrase, follow the receiving item name with the words DELIMITER IN and an identifier. The compiler generates code that places the delimiter character in the area named by the identifier. Consider the following sample UNSTRING statement:

```
UNSTRING FIELD1 DELIMITED BY ","
         OR TAB
         OR ALL SPACE
         OR CR
         INTO FIELD2A DELIMITER IN DELIMA
         FIELD2B DELIMITER IN DELIMB
         FIELD2C.
```

After moving the first sending string to FIELD2A, the character (or characters) that delimited that string is placed in DELIMA. In this example, DELIMA contains either a comma, a tab, a carriage return, or any number of spaces. Because the delimiter string is moved under the rules of the elementary nonnumeric MOVE statement, the compiler truncates or space-fills with left or right justification.

The second sending string is then moved to FIELD2B and its delimiting character is placed into DELIMB.

When a sending string is delimited by the end of the sending item rather than by a match on a delimiter, the delimiter string is of zero length and the DELIMITER item is space-filled. The phrase should be used only where needed. In this example, the character that delimits the last sending string is not needed, so no DELIMITER phrase follows FIELD2C.

The data item named in the DELIMITER phrase must be described as an alphanumeric item. It can contain editing characters, and it can also be a group item.

When you use both DELIMITER and COUNT phrases, the DELIMITER phrase must precede the COUNT phrase. Both of the data items named in these phrases can be subscripted or indexed. If they are subscripted, the subscript can be varied as a side effect of the UNSTRING operation.

7.2.5 Controlling UNSTRING Scanning Using the POINTER Phrase

Although the UNSTRING statement scan usually starts at the leftmost position of the sending item, the POINTER phrase lets you control the character position where the scan starts. Scanning, however, remains left to right.

When a sending item is to be unstrung into multiple receiving items, the choice of delimiters and the size of subsequent receiving items depends on the size of the first sending string and the character that delimited that string. Thus, the program needs to move the first sending item, hold its scanning position in the sending item, and examine the results of the operation to determine how to handle the sending items that follow.

This is done by using an UNSTRING statement with a POINTER phrase that fills only the first receiving item. When the first string has been moved to a receiving item, the compiler begins the next scanning operation one character beyond the delimiter that caused the interruption. The program examines the new position, the receiving item, the delimiter value, and the sending string size. It resumes the scanning operation by executing another UNSTRING statement with the same sending item and pointer data item. In this way, the UNSTRING statement moves one sending string at a time, with the form of each succeeding move depending on the context of the preceding string of data.

The POINTER phrase must follow the last receiving item in the UNSTRING statement. You are responsible for initializing the pointer before the UNSTRING statement executes. Consider the following two UNSTRING statements with their accompanying POINTER phrases and tests:

```
MOVE 1 TO PNTR.
UNSTRING FIELD1 DELIMITED BY ":"
         OR TAB
         OR CR
         OR ALL SPACE
         INTO FIELD2A DELIMITER IN DELIMA COUNT IN LSIZEA
         WITH POINTER PNTR.
IF LSIZEA = 0 GO TO NO-LABEL-PROCESS.
IF DELIMA =  ":"
         IF PNTR > 8 GO TO BIG-LABEL-PROCESS
         ELSE GO TO LABEL-PROCESS.
IF DELIMA = TAB GO TO BAD-LABEL PROCESS.
         .
         .
         .
UNSTRING FIELD1 DELIMITED BY ... WITH POINTER PNTR.
```

PNTR contains the current position of the scanner in the sending item. The second UNSTRING statement uses PNTR to begin scanning the additional sending strings in FIELD1.

Because the compiler considers the leftmost character to be character position 1, the value of PNTR can be used to examine the next character. To do this, describe the sending item as a table of characters and use PNTR as a sending item subscript. This is shown in the following example:

```
01 FIELD1.
   02 FIELD1-CHAR OCCURS 40 TIMES.
   .
   .
   .
   UNSTRING FIELD1
            .
            .
            .
            WITH POINTER PNTR.
   IF FIELD1-CHAR(PNTR) = "X" ...
```

Another way to examine the next character of the sending item is to use the UNSTRING statement to move the character to a 1-character receiving item:

```
UNSTRING FIELD1
         .
         .
         .
         WITH POINTER PNTR.
UNSTRING FIELD1 INTO CHAR1 WITH POINTER PNTR.
SUBTRACT 1 FROM PNTR.
IF CHAR1 = "X" ...
```

The program must decrement PNTR by 1 in order to work, because the second UNSTRING statement increments the pointer by 1.

The program must initialize the POINTER phrase data item before the UNSTRING statement uses it. The compiler will terminate the UNSTRING operation if the initial value of the pointer is less than one or greater than the length of the sending item. Such a pointer value causes an overflow condition. Overflow conditions are discussed in Section 7.2.7.

7.2.6 Counting UNSTRING Receiving Items Using the TALLYING Phrase

The TALLYING phrase counts the number of receiving items that received data from the sending item.

When an UNSTRING statement contains several receiving items, there are not always as many sending strings as there are receiving items. The TALLYING phrase provides a convenient method for keeping a count of how many receiving items actually received strings. The following example shows how to use the TALLYING phrase.

```
MOVE 0 TO RCOUNT.
UNSTRING FIELD1 DELIMITED BY ","
         OR ALL SPACE
         INTO FIELD2A
              FIELD2B
              FIELD2C
              FIELD2D
              FIELD2E
              TALLYING IN RCOUNT.
```

If the compiler has moved only three sending strings when it reaches the end of FIELD1, it adds 3 to RCOUNT. The first three receiving items (FIELD2A, FIELD2B, and FIELD2C) contain data from the UNSTRING operation, but the last two (FIELD2D and FIELD2E) do not.

The UNSTRING statement does not initialize the TALLYING data item. The TALLYING data item always contains the sum of its initial contents plus the number of receiving items receiving data. Thus, you might want to initialize the tally count before each use.

You can use the POINTER and TALLYING phrases together in the same UNSTRING statement, but the POINTER phrase must precede the TALLYING phrase. Both phrases must follow all of the item names, the DELIMITER phrase, and the COUNT phrase. The data items for both phrases must contain numeric integers without editing characters or the symbol P in their PICTURE character-strings; both data items can be either COMP or DISPLAY usage. They can be signed or unsigned and, if they are DISPLAY usage, they can contain any desired sign option.

7.2.7 Exiting an UNSTRING Statement Using the OVERFLOW Phrase

The OVERFLOW phrase detects the overflow condition and causes an imperative statement to be executed when it detects the condition. An overflow condition exists when:

- The UNSTRING statement is about to execute and its pointer data item contains a value less than one or greater than the size of the sending item. The compiler generates code that executes the OVERFLOW phrase before it moves any data, and the values of all the receiving items remain unchanged.
- Data still remains in the sending item after the UNSTRING statement has filled all the receiving items. The compiler executes the OVERFLOW phrase after it has executed the UNSTRING statement. The value of each receiving item is updated, but some data is still unmoved.

If the UNSTRING operation causes the scan to move past the rightmost position of the sending item (thus exhausting it), the compiler does not execute the OVERFLOW phrase.

The following set of instructions causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending item. The TALLYING data item is a subscript that indexes the receiving item. Compare this loop with the previous loop, which accomplishes the same thing:

```
      MOVE 1 TO TLY PNTR.
PAR1. UNSTRING FIELD1 DELIMITED BY ","
               OR CR
               INTO FIELD2(TLY) WITH POINTER PNTR
               TALLYING IN TLY
               ON OVERFLOW GO TO PAR1.
```

7.2.8 Common UNSTRING Statement Errors

The most common errors made when writing UNSTRING statements are as follows:

- Leaving the OR connector out of a delimiter list
- Misspelling or interchanging the words DELIMITED and DELIMITER

- Writing the DELIMITER and COUNT phrases in the wrong order when both are present (DELIMITER must precede COUNT)
- Omitting the word INTO (or writing it as TO) before the receiving item list
- Repeating the word INTO in the receiving item list as shown in this example:

```
UNSTRING FIELD1 DELIMITED BY SPACE
         OR TAB
         INTO FIELD2A DELIMITER IN DELIMA
         INTO FIELD2B DELIMITER IN DELIMB
         INTO FIELD2C DELIMITER IN DELIMC.
```

- Writing the POINTER and TALLYING phrases in the wrong order (POINTER must precede TALLYING)
- Failing to understand the rules concerning subscript evaluation

7.3 Examining and Replacing Characters Using the INSPECT Statement

The INSPECT statement examines the character positions in an item and counts or replaces certain characters (or groups of characters) in that item.

Like the STRING and UNSTRING operations, INSPECT operations scan across the item from left to right. Included in the INSPECT statement is an optional phrase that allows scanning to begin or terminate upon detection of a delimiter match. This feature allows scanning to begin within the item, as well as at the leftmost position.

7.3.1 Using the TALLYING and REPLACING Options of the INSPECT Statement

The TALLYING operation, which counts certain characters in the item, and the REPLACING operation, which replaces certain characters in the item, can be applied either to the characters in the delimited area of the item being inspected, or to only those characters that match a given character string or strings under stated conditions. Consider the following sample statements, both of which cause a scan of the complete item:

```
INSPECT FIELD1 TALLYING TLY FOR ALL "B".
INSPECT FIELD1 REPLACING ALL SPACE BY ZERO.
```

The first statement causes the compiler to scan FIELD1 looking for the character B. Each time a B is found, TLY is incremented by 1.

The second statement causes the compiler to scan FIELD1 looking for spaces. Each space found is replaced with a zero.

The TALLYING and REPLACING phrases support both single and multiple arguments. For example, both of the following statements are valid:

```
INSPECT FIELD1 TALLYING TLY FOR ALL "A" "B" "C".
INSPECT FIELD1 REPLACING ALL "A" "B" "C" BY "D".
```

You can use both the TALLYING and REPLACING phrases in the same INSPECT statement. However, when used together, the TALLYING phrase must precede the REPLACING phrase. An INSPECT statement with both phrases is equivalent to two separate INSPECT statements. In fact, the compiler compiles such a statement into two distinct INSPECT statements. To simplify debugging, write the two phrases in separate INSPECT statements.

7.3.2 Restricting Data Inspection Using the BEFORE/AFTER Phrase

The BEFORE/AFTER phrase acts as a delimiter and can restrict the area of the item being inspected.

The following sample statement counts only the zeros that precede the percent sign (%) in FIELD1:

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL ZEROS BEFORE "%".
```

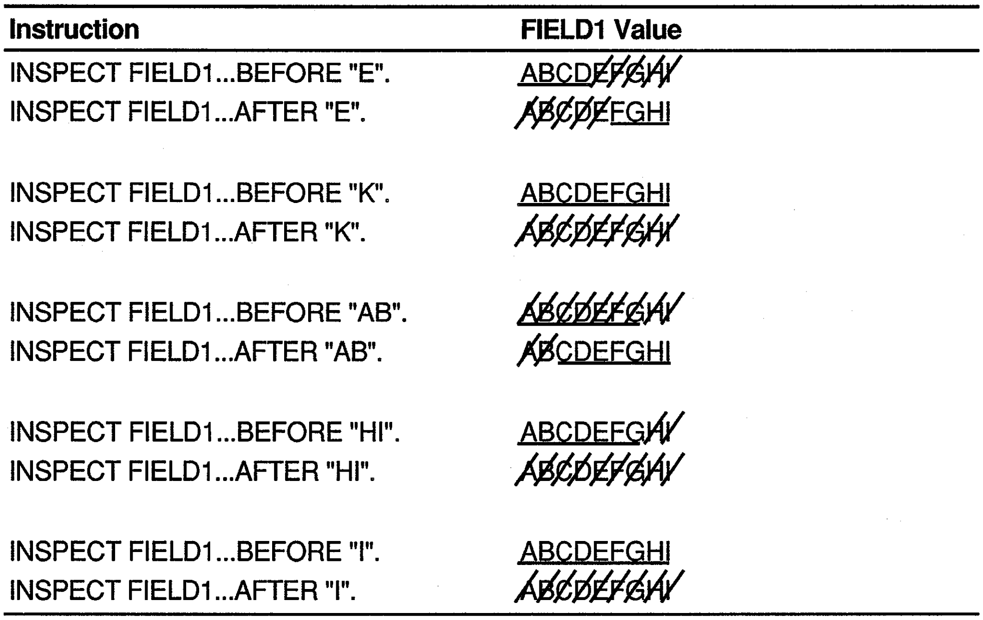
The delimiter (the percent sign in the preceding sample statement) can be a single character, a string of characters, or any figurative constant. Furthermore, it can be either an identifier or a literal.

- If the delimiter is an identifier, it must be an elementary data item of DISPLAY usage. It can be alphabetic, alphanumeric, or numeric, and it can contain editing characters. The compiler always treats the item as if it had been described as an alphanumeric string. It does this by implicit redefinition of the item, as described in Section 7.3.3.
- If the delimiter is a literal, it must be nonnumeric.

The compiler repeatedly compares the delimiter characters against an equal number of characters in the item being inspected. If none of the characters matches the delimiter, or if too few characters remain in the rightmost position of the item for a full comparison, the compiler considers the comparison to be unequal.

The examples of the INSPECT statement in Figure 7–2 illustrate the way the delimiter character finds a match in the item being inspected. The underlined characters indicate the portion of the item the statement inspects as a result of the delimiters of the BEFORE and AFTER phrases. The remaining portion of the item is ignored by the INSPECT statement.

Figure 7–2: Matching Delimiter Characters to Characters in a Field



Instruction	FIELD1 Value
INSPECT FIELD1...BEFORE "E".	ABCD~~EFGHI~~
INSPECT FIELD1...AFTER "E".	~~ABCDE~~FGHI
INSPECT FIELD1...BEFORE "K".	ABCDEFGHI
INSPECT FIELD1...AFTER "K".	~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "AB".	~~ABCDEFGHI~~
INSPECT FIELD1...AFTER "AB".	~~AB~~CDEFGHI
INSPECT FIELD1...BEFORE "HI".	ABCDEFG~~HI~~
INSPECT FIELD1...AFTER "HI".	~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "I".	ABCDEFGHI
INSPECT FIELD1...AFTER "I".	~~ABCDEFGHI~~

ZK–1426A–GE

The ellipses represent the position of the TALLYING or REPLACING phrase. The compiler generates code that scans the item for a delimiter match before it scans for the inspection operation (TALLYING or REPLACING), thus establishing the limits of the operation before beginning the actual inspection. Section 7.3.4.1 further discusses the separate scan.

7.3.3 Implicit Redefinition

The compiler requires that certain items referred to by the INSPECT statement be alphanumeric items. If one of these items is described as another data class, the compiler implicitly redefines that item so the INSPECT statement can handle it as an alphanumeric string as follows:

- If the item is alphabetic, alphanumeric edited, or unsigned numeric, the item is redefined as alphanumeric. This is a compile-time operation; no data movement occurs at run time.
- If the item is signed numeric, the compiler generates code that first removes the sign and then redefines the item as alphanumeric. If the sign is a separate character, that character is ignored, essentially shortening the item, and that character does not participate in the implicit redefinition. If the sign is an overpunch on the leading or trailing digit, the sign value is removed and the character is left with only the numeric value that was stored in it.

The compiler alters the digit position containing the sign before beginning the INSPECT operation and restores it to its former value after the operation. If the sign's digit position does not contain a valid ASCII signed numeric digit, redefinition causes the value to change.

Table 7–10 shows these original, altered, and restored values.

The compiler never moves an implicitly redefined item from its storage position. All redefinition occurs in place.

The position of an implied decimal point on numeric quantities does not affect implicit redefinition.

Table 7–10: Values Resulting from Implicit Redefinition

Original Value	Altered Value	Restored Value
} (173)	0 (60)	} (173)
A (101)	1 (61)	A (101)
B (102)	2 (62)	B (102)
C (103)	3 (63)	C (103)
D (104)	4 (64)	D (104)
E (105)	5 (65)	E (105)
F (106)	6 (66)	F (106)
G (107)	7 (67)	G (107)
H (110)	8 (70)	H (110)
I (111)	9 (71)	I (111)
{ (175)	0 (60)	{ (175)
J (112)	1 (61)	J (112)
K (113)	2 (62)	K (113)
L (114)	3 (63)	L (114)
M (115)	4 (64)	M (115)
N (116)	5 (65)	N (116)
O (117)	6 (66)	O (117)
P (120)	7 (67)	P (120)
Q (121)	8 (70)	Q (121)
R (122)	9 (71)	R (122)

(continued on next page)

Table 7–10 (Cont.): Values Resulting from Implicit Redefinition

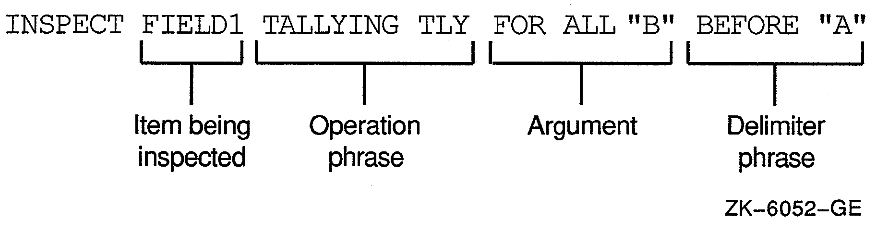
Original Value	Altered Value	Restored Value
0 (60)	0 (60)	} (173)
1 (61)	1 (61)	A (101)
2 (62)	2 (62)	B (102)
3 (63)	3 (63)	C (103)
4 (64)	4 (64)	D (104)
5 (65)	5 (65)	E (105)
6 (66)	6 (66)	F (106)
7 (67)	7 (67)	G (107)
8 (70)	8 (70)	H (110)
9 (71)	9 (71)	I (111)
All other values	0 (60)	} (173)

7.3.4 Examining the INSPECT Operation

Regardless of the type of inspection (TALLYING or REPLACING), the INSPECT statement has only one method for inspecting the characters in the item. This section analyzes the INSPECT statement and describes this inspection method.

Figure 7–3 shows an example of the INSPECT statement. The item to be inspected must be named (FIELD1 in our example), and the item name must be followed by a TALLYING phrase (TALLYING TLY). The TALLY phrase must be followed by one or more identifiers or literals (B). These identifiers or literals comprise the arguments. More than one argument makes up the argument list.

Figure 7–3: Sample INSPECT Statement

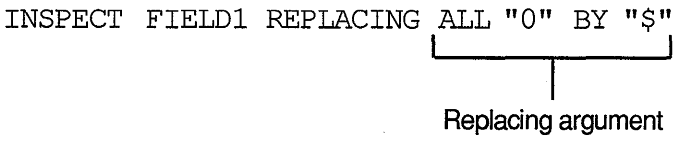


```
INSPECT FIELD1 TALLYING TLY FOR ALL "B" BEFORE "A"
```

Each argument in an argument list can have other items associated with it. Thus, each argument that is used in a TALLYING operation must have a tally counter (such as TLY in the example) associated with it. The tally counter is incremented each time it matches the argument with a character or group of characters in the item being inspected.

Each argument in an argument list used in a REPLACING operation must have a replacement item associated with it. The compiler generates code that uses the replacement item to replace each string of characters in the item that matches the argument. Figure 7–4 shows a typical REPLACING phrase (with $ as the replacement item).

Figure 7–4: Typical REPLACING Phrase



```
INSPECT FIELD1 REPLACING ALL "0" BY "$"
```

ZK-6053-GE

Each argument in an argument list used with either a TALLYING or REPLACING operation can have a delimiter item (BEFORE/AFTER phrase) associated with it. If the delimiter item is not present, the argument is applied to the entire item. If the delimiter item is present, the argument is applied only to that portion of the item specified by the BEFORE/AFTER phrase.

7.3.4.1 Setting the Scanner

The INSPECT operation begins by setting the scanner to the leftmost character position of the item being inspected. It remains on this character until an argument has been matched with a character (or characters) or until all arguments have failed to find a match at that position.

7.3.4.2 Active/Inactive Arguments

When an argument has a BEFORE/AFTER phrase associated with it, that argument has a delimiter and may not be eligible to participate in a comparison at every position of the scanner. Thus, each argument in the argument list has an active/inactive status at any given setting of the scanner.

For example, an argument that has an AFTER phrase associated with it starts the INSPECT operation in an inactive state. The delimiter of the AFTER phrase must find a match before the argument can participate in the comparison. When the delimiter finds a match, the compiler generates code that retains the character position beyond the matched character string; then, when the scanner reaches or passes this position, the argument becomes active. This is shown in the following example:

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL "B" AFTER "X".
```

If FIELD1 has a value of ABABXZBA, the argument B remains inactive until the scanner finds a match for delimiter X. Thus, argument B remains inactive while the compiler generates code that scans character positions 1 to 5. At character position 5, delimiter X finds a match, and since the character position beyond the matched delimiter character is the point at which the argument becomes active, argument B is compared for the first time at character position 6. It finds a successful match at character position 7, causing TLY to be incremented by 1.

Table 7–11 shows an INSPECT...TALLYING statement that is scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters listed in the left column. Assume that TLY is initialized to 0.

Table 7–11: Relationship Among INSPECT Argument, Delimiter, Item Value, and Argument Active Position

Argument and Delimiter	FIELD1 Value	Argument Active at Position	Contents of TLY After Scan
ALL	BXBXXXXBB	6	2
"B" AFTER "XX"	XXXXXXXX	3	0
	BXBXBBBBXX	never	0
	BXBXXBXXB	6	2
"X" AFTER "XX"	XXXXXXXX	3	6
	BBBBBBXX	never	0
	BXYBXBXX	7	0
"B" AFTER "XB"	XBXBXBXB	3	3
	BBBBBBXB	never	0
	XXXXBXXXX	6	0
"BX" AFTER "XB"	XXXXBBXXX	6	1
	XXBXXXXBX	4	1

When an argument has an associated BEFORE delimiter, the inactive/active states reverse roles: the argument is in an active state when the scanning begins and becomes inactive at the character position that matches the delimiter. Regardless of the presence of the BEFORE delimiter, an argument becomes inactive when the scanner approaches the rightmost position of the item and the remaining characters are fewer in number than the characters in the argument. In such a case, the argument cannot possibly find a match in the item, so it becomes inactive.

Since the BEFORE/AFTER delimiters are found on a separate scan of the item, the compiler generates code that recognizes and sets up the delimiter boundaries before it scans for an argument match; therefore, the same characters can be used as arguments and delimiters in the same phrase.

7.3.4.3 Finding an Argument Match

The compiler generates code that selects arguments from the argument list in the order in which they appear in the list. If the first one it selects is an active argument, and the conditions stated in the INSPECT statement allow a comparison, the compiler generates code that compares it to the character at the scanner's position. If the active argument does not find a match, the compiler generates code that takes the next active argument from the list and compares that to the same character. If none of the active arguments finds a match, the scanner moves one position to the right and begins the inspection operation again with the first active argument in the list. The inspection operation terminates at the rightmost position of the item.

When an active argument finds a match, the compiler ignores any remaining arguments in the list and conducts the TALLYING or REPLACING operation on the character. The scanner moves to a new position and the next inspection operation begins with the first argument in the list. The INSPECT statement can contain additional conditions, which are described later in this section; without them, however, the argument match is allowed to take place, and inspection continues following the match.

The compiler updates the scanner by adding the size of the matching argument to it. This moves the scanner to the next character beyond the string of characters that matched the argument. Thus, once an active argument matches a string of characters, the statement does not inspect those character positions again unless program control executes the entire statement again.

7.3.5 The TALLYING Phrase

An INSPECT statement that contains a TALLYING phrase counts the occurrences of various character strings under certain stated conditions. It keeps the count in a user-designated item called a tally counter.

7.3.5.1 The Tally Counter

The identifier following the word TALLYING designates the tally counter. The identifier can be subscripted or indexed. The data item must be a numeric integer without any editing or P characters; it can be COMP or DISPLAY usage, and it can be signed (separate or overpunched).

Each time the tally argument matches the delimited string being inspected, the compiler adds 1 to the tally counter.

You can initialize the tally counter to any numeric value. The INSPECT statement does not initialize it.

7.3.5.2 The Tally Argument

The tally argument specifies a character-string (or strings) and a condition under which that string should be compared to the delimited string being inspected.

The CHARACTERS form of the tally argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the tally argument. This increments the tally counter by a value that equals the size of the delimited string. For example, the following statement causes TLY to be incremented by the number of characters that precede the first comma, regardless of what those characters are:

```
INSPECT FIELD1 TALLYING TLY FOR
        CHARACTERS BEFORE ",".
```

The ALL and LEADING forms of the tally argument specify a particular character-string (or strings), which can be represented by either a literal or an identifier. The tally argument character-string can be any length; however, each character of the argument must match a character in the delimited string before the compiler considers the argument matched.

- A literal character-string must be either nonnumeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE or ZERO, represents a single character and can be written as " " " " or "0" with the same effect.

- An identifier must be an elementary item of DISPLAY usage. It can be any data class. However, if it is not alphanumeric, the compiler performs an implicit redefinition of the item. This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 7.3.2.

The words ALL and LEADING supply conditions that further delimit the inspection operation:

- ALL specifies that every match that the search argument finds in the delimited character string be counted in the tally counter. When a literal follows the word ALL, it does not have the same meaning as the figurative constant, ALL literal. The ALL literal meaning of ALL "," is a string of consecutive commas (as many as the context of the statement requires). ALL "," used as a tally argument means "count each comma without regard to adjacent characters."
- LEADING specifies that only adjacent matches of the TALLY argument at the leftmost position of the delimited character string be counted. At the first failure to match the tally argument, the compiler terminates counting and causes the argument to become inactive. The sample statement INSPECT...TALLYING (scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters listed in the left column) gives the results in Table 7–12 (if the program initializes TLY to 0).

Table 7–12: LEADING Delimiter of the Inspection Operation

Argument and Delimiter	FIELD1 Value	Contents of TLY After Scan
LEADING "*" AFTER "0".	F***0**F	2
	F**0F**	0
	F**F**0	0
	0***F**	3
LEADING "**" AFTER "0".	F**0**F***	1
	F**F0***FF	1
	F**F0****F**	2
	F**F**0*	0

7.3.5.3 The Tally Argument List

One INSPECT...TALLYING statement can contain more than one tally argument, and each argument can have a separate BEFORE/AFTER phrase and tally counter associated with it. These tally arguments with their associated tally counters and BEFORE/AFTER phrases form an argument list. The manner in which this list is processed affects the action of any given tally argument.

The following examples show INSPECT statements with argument lists. The text with each example explains how that list is processed.

```
INSPECT FIELD1 TALLYING T FOR
        ALL ","
        ALL "."
        ALL ";".
```

These three tally arguments have the same tally counter, T, and are active over the entire item being inspected. Thus, the preceding statement adds the total number of commas, periods, and semicolons in FIELD1 to the initial value of T. Since the TALLYING phrase supports multiple arguments and only one counter is used, the previous statement could have been written as follows:

```
INSPECT FIELD1 TALLYING T FOR ALL "," "." ";".
```

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "."
        T3 FOR ALL ";".
```

Each tally argument in this statement has its own tally counter and is active over the entire item being inspected. Thus, the preceding statement adds the total number of commas in FIELD1 to the initial value of T1, the total number of periods to the initial value of T2, and the number of semicolons to T3.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "," AFTER "A"
        T2 FOR ALL "." BEFORE "B"
        T3 FOR ALL ";".
```

Each tally argument in the preceding statement has its own tally counter; the first two arguments have delimiter phrases, and the last one is active over the entire item being inspected. Thus, the first argument is initially inactive and becomes active only after the scanner encounters an A; the second argument begins the scan in the active state but becomes inactive after a B has been encountered; and the third argument is active during the entire scan of FIELD1.

Table 7–13 shows various values of FIELD1 and the contents of the three tally counters after the scan of the previous statements. Assume that the counters are initialized to 0 before the INSPECT statement.

Table 7–13: Results of the Scan with Separate Tallies

	Contents of Tally Counters After Scan		
FIELD1 Value	**T1**	**T2**	**T3**
A.C;D.E,F	1	2	1
A.B.C.D	0	1	0
A,B,C,D	3	0	0
A;B;C;D	0	0	3
*,B,C,D	0	0	0

The BEFORE/AFTER phrase applies only to the argument that precedes it and delimits the item for that argument only. Each BEFORE/AFTER phrase causes a separate scan of the item to determine the limits of the item for its corresponding argument.

7.3.5.4 Interference in Tally Argument Lists

When several tally arguments contain one or more identical characters active at the same time, they may interfere with each other, so that when one of the arguments finds a match, the scanner steps past any other matching characters, preventing those characters from being considered for a match.

The following two identical tally arguments do not interfere with each other since they are not active at the same time. The first A in FIELD1 causes the first argument to become inactive and the second argument to become active:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR ALL "," BEFORE "A"
        T2 FOR ALL "," AFTER "A".
```

However, the next identical tally arguments interfere with each other since both are active at the same time:

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "," AFTER "A".
```

For any given position of the scanner, the arguments are applied to FIELD1 in the order in which they appear in the statement. When one of them finds a match, the scanner moves to the next position and ignores the remaining arguments in the argument list. Each comma in FIELD1 causes T1 to be incremented by 1 and the second argument to be ignored. Thus, T1 always contains an accurate count of all the commas in FIELD1, and T2 is always unchanged.

The following INSPECT statement arguments only partially interfere with each other:

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "," AFTER "A"
        T1 FOR ALL ",".
```

The first argument does not become active until the scanner encounters an A. The second argument tallies all commas that precede the A. After the A, the first argument counts all commas and causes the second argument to be ignored. Thus, T1 contains the number of commas that precede the first A, and T2 contains the number of commas that follow the first A. This statement works well as written, but it could be difficult to debug.

The following three examples show that one INSPECT statement cannot count any character more than once. Thus, when you use the same character in more than one argument of an argument list, consider the possibility of interference and choose the order of the arguments carefully. The solution may require two or more INSPECT statements. Consider the following problem:

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "AB"
        T2 FOR ALL "BC".
```

If FIELD1 contains ABCABC after the scan, T1 is incremented by 2, and T2 is unaltered. The successful matching of the argument includes each B in the item. Each match resets the scanner to the character position to the right of the B, so that the second argument is never successfully matched. The results remain the same even if the order of the arguments is reversed. Only separate INSPECT statements can develop the desired counts.

Sometimes you can use the interference characteristics of the INSPECT statement to your advantage. Consider the following sample argument list:

```
MOVE 0 TO T4 T3 T2 T1.
INSPECT FIELD1 TALLYING
        T4 FOR ALL "****"
        T3 FOR ALL "***"
        T2 FOR ALL "**"
        T1 FOR ALL "*".
```

The argument list counts all of the asterisks in FIELD1 in four different tally counters. T4 counts the number of times that four asterisks occur together; T3 counts the number of times three asterisks appear together; T2 counts double asterisks; and T1 counts singles.

If FIELD1 contains a string of more than four consecutive asterisks, the argument list breaks the string into groups of four and counts them in T4. It then counts the less-than-four remainder in T3, T2, or T1.

Reversing the order of the arguments in this list causes T1 to count all of the asterisks, and T2, T3, and T4 to remain unchanged.

When the LEADING condition is used with an argument in the argument list, that argument becomes inactive as soon as it fails to be matched in the item being inspected. Therefore, when two arguments in an argument list contain one or more identical characters and one of the arguments has a LEADING condition, the argument with the LEADING condition should appear first. Consider the following sample statement:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING "*"
        T2 FOR ALL "*".
```

T1 counts only leading asterisks in FIELD1; the occurrence of any other character causes the first tally argument to become inactive. T2 keeps a count of any remaining asterisks in FIELD1.

Reversing the order of the arguments in the following statement results in an argument list that can never increment T1:

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "*"
        T1 FOR LEADING "*".
```

If the first character in FIELD1 is not an asterisk, neither argument can match it, and the second argument becomes inactive. If the first character in FIELD1 is an asterisk, the first argument matches it and causes the second argument to be ignored. The first character in FIELD1 that is not an asterisk fails to match the first argument, and the second argument becomes inactive because it has not found a match in any of the preceding characters.

An argument with both a LEADING condition and a BEFORE phrase can sometimes successfully delimit the item being inspected, as in the following example:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T2 FOR ALL "   " BEFORE "."
        T2 FOR ALL "  " BEFORE "."
        T2 FOR ALL " " BEFORE ".".
IF T2 > 0 ADD 1 TO T2.
```

These statements count the number of words in the English statement in FIELD1, assuming that no more than three spaces separate the words in the sentence, that the sentence ends with a period, and that the period immediately follows the last word. When FIELD1 has been scanned, T2 contains the number of spaces between the words. Since a count of the spaces renders a number that is one less than the number of words, the conditional statement adds 1 to the count.

The first argument removes any leading spaces, counting them in a different tally counter. This shortens FIELD1 by preventing the application of the second to the fourth arguments until the scanner finds a nonspace character. The BEFORE phrase on each of the other arguments causes them to become inactive when the scanner reaches the period at the end of the sentence. Thus, the BEFORE phrases shorten FIELD1 by making the second to the fourth arguments inactive before the scanner reaches the rightmost position of FIELD1. If the sentence in FIELD1 is indented with tab characters instead of spaces, a second LEADING argument can count the tab characters. For example:

```
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T1 FOR LEADING TAB
        T2 FOR ALL "    "
        .
        .
        .
```

When an argument list contains a CHARACTERS argument, it should be the last argument in the list. Since the CHARACTERS argument always matches the item, it prevents the application of any arguments that follow in the list. However, as the last argument in an argument list, it can count the remaining characters in the item being inspected. Consider the following example.

```
MOVE 0 TO T1 T2 T3 T4 T5.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T2 FOR ALL "." BEFORE ","
        T3 FOR ALL "+" BEFORE ","
        T4 FOR ALL "-" BEFORE ","
        T5 FOR CHARACTERS BEFORE ",".
```

If FIELD1 is known to contain a number in the form frequently used to input data, it can contain a plus or minus sign, and a decimal point; furthermore, the number can be preceded by spaces and terminated by a comma. When this statement is compiled and executed, it delivers the following results:

- T1 contains the number of leading spaces.
- T2 contains the number of periods.
- T3 contains the number of plus signs.
- T4 contains the number of minus signs.
- T5 contains the number of remaining characters (assumed to be numeric).

The sum of T1 to T5, plus 1, gives the character position occupied by the terminating comma.

7.3.6 Using the REPLACING Phrase

When an INSPECT statement contains a REPLACING phrase, that statement selectively replaces characters or groups of characters in the designated item.

The REPLACING phrase names a search argument of one or more characters and a condition under which the string can be applied to the item being inspected. Associated with the search argument is the replacement value, which must be the same length as the search argument. Each time the search argument finds a match in the item being inspected, under the condition stated, the replacement value replaces the matched characters.

A BEFORE/AFTER phrase can be used to delimit the area of the item being inspected. A search argument applies only to the delimited area of the item.

7.3.6.1 The Search Argument

The search argument of the REPLACING phrase names a character string and a condition under which the character string should be compared to the delimited string being inspected.

The CHARACTERS form of the search argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the search argument. Thus, the replacement value replaces each character in the delimited string. For example:

```
INSPECT ITEMA REPLACING CHARACTERS ...
```

The ALL, LEADING, and FIRST forms of the search argument specify a particular character string, which can be represented by a literal or an identifier. The search argument character string can be any length. However, each character of the argument must match a character in the delimited string before the compiler considers the argument matched. For example:

```
INSPECT ITEMA REPLACING ALL ...
```

The necessary literal and identifier characteristics are as follows:

- A literal character string must be either nonnumeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE or ZERO, represents a single character and can be written as " " or "0" with the same effect. Because a figurative constant represents a single character, the replacement value must be one character long.
- An identifier must represent an elementary item of DISPLAY usage. It can be any class. However, if it is not alphabetic, the compiler performs an implicit redefinition of the item. This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 7.3.2.

The words ALL, LEADING, and FIRST supply conditions that further delimit the inspection operation:

- ALL specifies that each match the search argument finds in the delimited character string is replaced by the replacement value. When a literal follows the word ALL, it does not have the same meaning as the figurative constant, ALL literal. The figurative constant meaning of ALL "," is a string of consecutive commas, as many as the context of the statement requires. ALL "," as a search argument of the REPLACING phrase means "replace each comma without regard to adjacent characters."
- LEADING specifies that only adjacent matches of the search argument at the leftmost position of the delimited character-string be replaced. At the first failure to match the search argument, the compiler terminates the replacement operation and causes the argument to become inactive.
- FIRST specifies that only the leftmost character string that matches the search argument be replaced. After the replacement operation, the search argument containing this condition becomes inactive.

7.3.6.2 The Replacement Value

Whenever the search argument finds a match in the item being inspected, the matched characters are replaced by the replacement value. The word BY followed by an identifier or literal specifies the replacement value. For example:

```
INSPECT ITEMA REPLACING ALL "A" BY "X" ALL "D" BY "X".
```

The replacement value must always be the same size as its associated search argument.

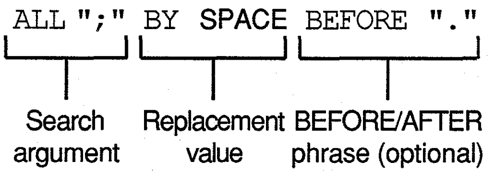
If the replacement value is a literal character-string, it must be either a nonnumeric literal or a figurative constant (other than ALL literal). A figurative constant represents as many characters as the length of the search argument requires.

If the replacement value is an identifier, it must be an elementary item of DISPLAY usage. It can be any class. However, if it is not alphanumeric, the compiler conducts an implicit redefinition of the item. This redefinition is the same as the BEFORE/AFTER redefinition discussed in Section 7.3.2.

7.3.6.3 The Replacement Argument

The replacement argument consists of the search argument (with its condition and character-string), the replacement value, and an optional BEFORE/AFTER phrase, as shown in Figure 7–5.

Figure 7–5: The Replacement Argument



ZK–6054–GE

7.3.6.4 The Replacement Argument List

One INSPECT...REPLACING statement can contain more than one replacement argument. Several replacement arguments form an argument list, and the manner in which the list is processed affects the action of any given replacement argument.

The following examples show INSPECT statements with replacement argument lists. The text following each one tells how that list will be processed.

```
INSPECT FIELD1 REPLACING
        ALL "," BY SPACE
        ALL "." BY SPACE
        ALL ";" BY SPACE.
```

The previous three replacement arguments all have the same replacement value, SPACE, and are active over the entire item being inspected. The statement replaces all commas, periods, and semicolons with space characters and leaves all other characters unchanged.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1"
        ALL "1" BY "0".
```

Each of these two replacement arguments has its own replacement value and is active over the entire item being inspected. The statement exchanges zeros for 1s and 1s for zeros. It leaves all other characters unchanged.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE.
```

NOTE

> When a search argument finds a match in the item being inspected, the code replaces that character-string and scans to the next position beyond the replaced characters. It ignores the remaining arguments and applies the first argument in the list to the character-string in the new position. Thus, it never inspects the new value that was supplied by the replacement operation. Because of this, the search arguments can have the same values as the replacement arguments with no chance of interference.

The statement also exchanges zeros and 1s. Here, however, the first space in FIELD1 causes both arguments to become inactive.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE
        CHARACTERS BY "*" BEFORE SPACE.
```

The first space causes the three replacement arguments to become inactive. This argument list exchanges zeros for 1s, 1s for zeros, and asterisks for all other characters in the delimited area. If the BEFORE phrase is removed from the third argument, that argument will remain active across all of FIELD1. Within the area delimited by the first space character, the third argument replaces all characters except 1s and zeros with asterisks. Beyond this area, it replaces all characters (including the space that delimited FIELD1 for the first two arguments, and any zeros and 1s) with asterisks.

7.3.6.5 Interference in Replacement Argument Lists

When several search arguments, all active at the same time, contain one or more identical characters, they can interfere with each other—and consequently affect the replacement operation. This interference is similar to the interference that occurs between tally arguments.

The action of a search argument is never affected by the BEFORE/AFTER delimiters of other arguments, since the compiler scans for delimiter matches before it scans for replacement operations.

The action of a search argument is never affected by the characters of any replacement value, since the scanner does not inspect the replaced characters again during execution of the INSPECT statement. Interference between search arguments, therefore, depends on the order of the arguments, the values of the arguments, and the active/inactive status of the arguments. The discussion in Section 7.3.5.4 about interference in tally argument lists generally applies to replacement arguments as well.

The following rules help minimize interference in replacement argument lists:

1. Place search arguments with LEADING or FIRST conditions at the start of the list.
2. Place any arguments with the CHARACTERS condition at the end of the list.

3. Consider the order of appearance of any search arguments that contain identical characters.

7.3.7 Using the CONVERTING Option

When an INSPECT statement contains a CONVERTING phrase, that statement selectively replaces characters or groups of characters in the designated item; it executes as if it were a Format 2 INSPECT statement with a series of ALL phrases. (See the INSPECT statement formats in the *VAX COBOL Reference Manual.*)

An example of the use of the CONVERTING phrase follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROGX.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 X PIC X(28).
PROCEDURE DIVISION.
A.
    MOVE "ABC*ABC*ABC ABC@ABCABC" TO X.
    INSPECT X CONVERTING "ABC" TO "XYZ"
            AFTER "*" BEFORE "@".
    DISPLAY X.
    STOP RUN.
```

```
    X before INSPECT executes      X after INSPECT executes

     ABC*ABC*ABC ABC@ABCABC         ABC*XYZ*XYZ XYZ@ABCABC
```

7.3.8 Common INSPECT Statement Errors

Programmers most commonly make the following errors when writing INSPECT statements:

- Leaving the FOR out of an INSPECT...TALLYING statement
- Using the word WITH instead of BY in the REPLACING phrase
- Failing to initialize the tally counter
- Omitting the word ALL before the comparison character-string

Chapter 8

The Basics of Handling VAX COBOL Files and Records

Input and output services require a complex management system; otherwise, the programmer is left with the task of producing detailed input/output control for each program. The VMS operating system provides complete I/O services for handling, controlling, and spooling I/O needs or requests. VAX Record Management Services (RMS) gives a wide range of record management techniques while remaining transparent to you. This chapter introduces you to:

- VAX Record Management Services
- VAX COBOL file organizations
- VAX COBOL file attributes
- VAX COBOL record attributes
- VAX COBOL record access modes
- VAX COBOL OPEN and CLOSE statements
- VAX COBOL with the VAX Vertical Forms Printing software utility

8.1 VAX Record Management Services

VAX COBOL provides extensive capabilities for data storage, retrieval, and modification for the VAX COBOL programmer through VAX Record Management Services (RMS). You can select from one of several file organizations and access techniques—each of which is suited to a particular application—from the simplest sequential search of a sequentially organized file to a sophisticated dynamic access of an indexed file based on one of several alternate key fields.

The three file organizations available through VAX COBOL and RMS—sequential, relative, and indexed—are available to three different access modes: sequential, random, and dynamic. Dynamic access, or access mode switching, is a useful feature that allows your program to switch from sequential to random access and back during file processing.

The following topics are contained in the RMS documentation set. These topics can help you choose the correct RMS defaults for your applications:

- Application design
- File design
- Task design
- Common optimization techniques

- RMS utilities
- Magnetic tape handling

Understanding the preceding RMS topics allows you to achieve the best performance from your application, as many RMS defaults are not the optimal choice for all your applications. See the VMS documentation on VAX Record Management Services for further information.

8.2 File Attributes

A file is a collection of related records. File attributes let you specify the following:

- File organization
- Record format
- Physical record size
- File size

The system uses these attributes to create a file and stores them with the file. When a program accesses a file, it must specify the same attributes stored when the file was created. For example, a program cannot read a sequential file as an indexed file, because no index keys exist.

In VAX COBOL programs, yo'ı specify a file's attributes in the Environment and Data Divisions:

- The APPLY clause specifies file characteristics such as lock-holding, file extension factors, and preallocation factors. The SELECT statement specifies the file organization.
- File description entries specify record format and record blocking.
- Record description entries specify physical record size or sizes.

Chapter 9, Chapter 10, and Chapter 11 all present and discuss examples of each type of file organization supported by VAX COBOL. Chapter 20 explains the use of the APPLY clause.

8.3 Record Attributes

A record is a group of related data elements. The space a record needs on a physical device depends on:

- The file organization
- The record format
- The number of bytes the record contains

If a file has more than one record description, the different record descriptions automatically share the same record area in memory. The Object or Run-Time System does not clear this area before it executes the READ statement. Therefore, if the record read by the latest READ statement does not fill the entire record area, the area not overlaid by the incoming record remains unchanged.

8.3.1 Record Format

You can use fixed, variable, or variable with fixed control record format types.

The compiler determines record format from a combination of record description entries and the RECORD CONTAINS clause. You specify the record format as follows:

- For fixed—Use the RECORD CONTAINS clause or the VAX COBOL default.
- For variable—Use the RECORD CONTAINS TO clause or RECORD VARYING.
- For variable fixed control (VFC)—Use the ADVANCING, APPLY, or LINAGE clause, or use Report Writer statements and phrases.

In Example 8–1, a file contains a company's stock inventory information (part number, supplier, quantity, price). Within this file, the information is divided into records. All information for a single piece of stock constitutes a single record.

Each record in the stock file is itself divided into discrete pieces of information known as elementary items. You give the item a specific location in the record, give it a name, and define its size. The part number is an item in the part record, as are supplier, quantity, and price. In this example PART-RECORD contains four elementary items: PART-NUMBER, PART-SUPPLIER, PART-QUANTITY, and PART-PRICE.

Example 8–1: Sample Record Description

```
01  PART-RECORD.
    02  PART-NUMBER                 PIC 9999.
    02  PART-SUPPLIER               PIC X(20).
    02  PART-QUANTITY               PIC 99999.
    02  PART-PRICE                  PIC S9(5)V99.
```

You can completely control the grouping of elementary items into records and records into files. VAX COBOL programs either build records and pass them to RMS for storage in a file, or they issue requests for records while RMS performs the necessary operations to retrieve the records from a file.

The maximum size of a record depends on its format:

- For fixed-length records, the maximum size is the record size.
- For variable-length records, the maximum size is the size of the largest record plus the number of overhead bytes needed by RMS.
- For variable-fixed control records, the maximum size is the size of the largest record plus header overhead.

In all cases, the length of any record in a file description entry cannot exceed 32,767 bytes for a sequential file, 32,234 bytes for an indexed file, or 32,255 bytes for a relative file.

8.3.1.1 Fixed-Length Records

Files with a fixed-length record format contain the same size records. The compiler generates the fixed-length format when either of the following conditions is true:

- The RECORD CONTAINS clause specifies a fixed number of characters.
- The RECORD CONTAINS clause is omitted.

The compiler does not generate fixed-length format when either of the following conditions exist:

- The file description contains a RECORD CONTAINS TO clause or a RECORD VARYING clause.
- The program specifies a print-controlled file by referring to the file with:
  - The ADVANCING phrase in a WRITE statement
  - An APPLY PRINT-CONTROL clause in the Environment Division
  - A LINAGE clause in the file description
  - Report Writer statements and phrases

Fixed-length record size is determined by either the largest record description or the record size specified by the RECORD CONTAINS clause, whichever is larger. Example 8–2 shows how fixed-length record size is determined.

Example 8–2: Determining Fixed-Length Record Size

```
FD  FIXED-FILE
    RECORD CONTAINS 100 CHARACTERS.
01  FIXED-REC      PIC X(75).
```

For the file, FIXED-FILE, the RECORD CONTAINS clause specifies a record size larger than the record description; therefore, the record size is 100 characters.

However, if the multiple record descriptions are associated with the file, the size of the largest record description is used as the size. Thus, in Example 8–3, for the file REC-FILE, the FIXED-REC2 record specifies the largest record size; therefore, the record size is 90 characters.

Example 8–3: Determining Fixed-Length Record Size for Files with Multiple Record Descriptions

```
FD  REC-FILE
    RECORD CONTAINS 80 CHARACTERS.
01  FIXED-REC1     PIC X(75).
01  FIXED-REC2     PIC X(90).
```

In Example 8–2, the following warning message is generated when the file FIXED-FILE is used:

```
"Record contains value is greater than length of longest record."
```

And when the file REC-FILE is used, the following warning message is generated:

```
"Longest record is longer than RECORD CONTAINS value -
   longest record size used."
```

8.3.1.2 Variable-Length Records

Files with a variable-length record format can contain different length records. The compiler generates the variable-length attribute for a file when the file description contains a RECORD VARYING clause or a RECORD CONTAINS TO clause. (See also Section 8.3.2.)

The system stores the record's size in bytes in a record-length field that precedes each record.

- For disk files, the record-length field is a 2-byte value specifying record length in bytes. Note that a record's length does not include this 2-byte field.
- For ANSI magnetic tape files, the record-length field is a 4-byte decimal value specifying record length in bytes. Note that a record's length includes this 4-byte field.

Example 8–4, Example 8–5, and Example 8–6 show you the three ways VAX COBOL lets you create a variable-length record file.

In Example 8–4, the DEPENDING ON phrase sets the OUT-REC record length. The IN-TYPE data field determines the OUT-LENGTH field's contents.

Example 8–4: Creating Variable-Length Records with the DEPENDING ON Phrase

```
    .
    .
    .
FILE SECTION.

FD  INFILE
    RECORD LABELS ARE STANDARD.
01  IN-REC.
    03  IN-TYPE        PIC X.
    03  REST-OF-REC    PIC X(499).

FD  OUTFILE
    RECORD VARYING FROM 200 TO 500 CHARACTERS
    DEPENDING ON OUT-LENGTH.
01  OUT-REC            PIC X(500).
WORKING-STORAGE SECTION.
01  OUT-LENGTH         PIC 999 COMP VALUE ZEROES.
    .
    .
    .
```

Example 8–5: Creating Variable-Length Records with the RECORD VARYING Phrase

```
FILE SECTION.
FD  OUTFILE
    RECORD VARYING FROM 200 TO 500 CHARACTERS.
01  OUT-REC-1          PIC X(200).
01  OUT-REC-2          PIC X(500).
```

Example 8–5 shows how to create variable-length records using the RECORD VARYING phrase.

Example 8–6 creates variable-length records by using the OCCURS clause with the DEPENDING ON phrase in the record description. VAX COBOL determines record length by adding the sum of the variable record's fixed portion to the size of the table described by the number of table occurrences at execution time.

In this example, the variable record's fixed portion size is 113 characters. (This is the sum of P-PART-NUM, P-PART-INFO, and P-BIN-INDEX.) If P-BIN-INDEX contains a 7 at execution time, P-BIN-NUMBER will be 35 characters long. Therefore, PARTS-REC's length will be 148 characters; the fixed portion's length is 113 characters, and the table entry's length at execution time is 35 characters.

Example 8–6: Creating Variable-Length Records and Using the OCCURS Clause with the DEPENDING ON Phrase

```
      .
      .
      .
FILE SECTION.
FD  PARTS-MASTER
    RECORD VARYING 118 TO 163 CHARACTERS.
01  PARTS-REC.
    03  P-PART-NUM     PIC X(10).
    03  P-PART-INFO    PIC X(100).
    03  P-BIN-INDEX    PIC 999.
    03  P-BIN-NUMBER   PIC X(5)
        OCCURS 1 TO 10 TIMES DEPENDING ON P-BIN-INDEX.
      .
      .
      .
```

If you describe a record with both the RECORD VARYING...DEPENDING ON phrase on data-name-1 and the OCCURS clause with the DEPENDING ON phrase on data-name-2, VAX COBOL specifies record length as the value of data-name-1.

If you have multiple record-length descriptions for a file and omit either the RECORD VARYING clause or the RECORD CONTAINS integer-1 TO integer-2 clause, all records written to the file will have a fixed length equal to the length of the longest record described for the file, as in Example 8–7.

Example 8–7: Defining Fixed-Length Records with Multiple Record Descriptions

```
    .
    .
    .
FD PARTS-MASTER.
01  PARTS-REC-1    PIC X(200).
01  PARTS-REC-2    PIC X(300).
01  PARTS-REC-3    PIC X(400).
01  PARTS-REC-4    PIC X(500).
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
100-WRITE-REC-1.
    MOVE IN-REC TO PARTS-REC-1.
    WRITE PARTS-REC-1.
    GO TO ...
200-WRITE-REC-2.
    MOVE IN-REC TO PARTS-REC-2.
    WRITE PARTS-REC-2
    GO TO ...
    .
    .
    .
```

Writing PARTS-REC-1, PARTS-REC-2, PARTS-REC-3 or PARTS-REC-4 produces records equal in length to the longest record, PARTS-REC-4. Note that this is not variable-length I/O.

8.3.2 Print-Controlled Files

Print-controlled files contain form-advancing information with each record. VAX COBOL places explicit form-control bytes directly into the file. Therefore, any VAX COBOL program trying to read a print-control file can read it successfully as variable files (RMS strips the VFC header).

If you use the WRITE AFTER ADVANCING, the LINAGE, or the APPLY PRINT-CONTROL statement, or if you create a Report Writer file, the compiler generates variable-length print-controlled records. You must use the /NOFEED option on the DCL PRINT command when you print a print-controlled file.

8.4 File Design Considerations

The difficulty of design is proportional to the complexity of the file organization; design is least important for applications using sequential organization, more important for relative organization, and most important for indexed organization. Chapter 9, Chapter 10, and Chapter 11 all discuss file design for sequential, relative, and indexed files, respectively.

8.5 File Handling

Before your program can perform I/O on a file, it must identify the file to the operating system, specify the file's organization and access modes, and make the file available by opening it. A program must follow these steps whenever creating a file or processing one that has already been created.

8.5.1 Identifying a File from Your VAX COBOL Program

A file description entry defines a file's logical structure and associates the file with a file name that is unique within the program. The program uses this file name in the OPEN, READ, START, UNLOCK, DELETE, REWRITE, and CLOSE statements. (The record name is used for WRITE, UNLOCK, and REWRITE.)

You must establish a link between the file name your program uses and the file specification that RMS uses. The SELECT and ASSIGN and VALUE OF ID clauses do this. Together these clauses define a file connector. A file connector is a data structure used by VAX COBOL that contains information about a file. It links the following:

- A file name and a physical file
- A file name and its associated record area

The program must include a SELECT statement, including an ASSIGN clause, for every file description entry (FD) it contains. The file name you specify in the SELECT statement must match the file name in the file description entry. In the ASSIGN clause, you specify a literal or a COBOL word that associates the file name with a file specification. This literal or word can be a complete file specification or one that relies on operating system defaults.

To understand the relationships between the SELECT statement, the ASSIGN clause, and the FD entry, consider two examples. In Example 8–8, because the file name specified in the FD entry is DAT-FILE, all I/O statements in the program referring to that file must use the name DAT-FILE. RMS uses the ASSIGN clause to interpret DAT-FILE as REPORT.DAT and refers to the default directory.

Example 8–8: Defining a Disk File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DAT-FILE
           ASSIGN TO "REPORT.DAT"
           .
           .
           .
DATA DIVISION.
FD  DAT-FILE
           .
           .
           .
```

The I/O statements in Example 8–9 refer to MYFILE-PRO, which the ASSIGN clause identifies to the operating system as MARCH.311. Additionally, the operating system looks for the file in the current directory on the magnetic tape mounted on MTA0:.

Example 8–9: Defining a Magnetic Tape File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MYFILE-PRO
           ASSIGN TO "MTA0:MARCH.311"
           .
           .
           .
DATA DIVISION.
FD  MYFILE-PRO
           .
           .
           .
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT MYFILE-PRO.
           .
           .
           .
    READ MYFILE-PRO.
           .
           .
           .
    CLOSE MYFILE-PRO.
```

8.5.1.1 Using the VALUE OF ID Clause for Device Independence

If the file specification is subject to change, it is inconvenient to edit the ASSIGN clause and recompile and relink the program every time you run it. To avoid this problem, you can use a partial file specification in the ASSIGN clause and complete it by using the optional VALUE OF ID clause of the FD entry. In the VALUE OF ID clause, you may specify a nonnumeric literal or an alphanumeric WORKING-STORAGE item to supplement the file specification.

The VALUE OF ID clause completes or overrides the file specification in the ASSIGN clause. This lets you keep the file specification a variable until run time.

Example 8–10 illustrates how to use the VALUE OF ID clause to complete a partial file specification, MARCH, with operator input. Notice how the Procedure Division statements prompt the operator for a file specification. This technique provides the following advantages:

- Maximum flexibility for file access. The operator can override any part of the file specification in the ASSIGN clause.
- Maximum use of system hardware. The operator can mount a tape (or any other volume) on any available tape drive and direct the program to it.
- Maximum use of computer operator and operating system. The operator and operating system no longer have to wait for one job to finish using a specific tape drive before the next job can be started.

Example 8–10: How to Override or Supplement a File Specification at Run Time

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MYFILE-PRO
           ASSIGN TO "MARCH"
           .
           .
           .
DATA DIVISION.
FILE SECTION.
FD  MYFILE-PRO
***********************************************
*
    VALUE OF ID IS USER-EXTENSION.
*
***********************************************
           .
           .
           .
WORKING-STORAGE SECTION.
***********************************************
*
01  USER-EXTENSION     PIC X(20).
*
***********************************************
PROCEDURE DIVISION.
A000-BEGIN.
***********************************************
*
    DISPLAY "Enter file specification".
    ACCEPT USER-EXTENSION.
*
***********************************************
    OPEN INPUT MYFILE-PRO.
           .
           .
           .
    READ MYFILE-PRO.
           .
           .
           .
    CLOSE MYFILE-PRO.
```

NOTE

If no file type is supplied, VAX COBOL supplies a DAT file type.

8.5.1.2 Using Logical Names

Logical names let you write programs that are device and file independent and provide a brief way to refer to frequently used files.

You can assign logical names with the ASSIGN command. When you assign a logical name, the logical name and its equivalence name (the name of the actual file or device) are placed in one of three logical name tables; the choice depends on whether they are assigned for the current process, on the group level, or on a system-wide basis. See the VMS documentation for more information on DCL and a description of logical name tables.

To translate a logical name, the system searches the three tables in this order: (1) process, (2) group, (3) system. Therefore, you can override a system-wide logical name by defining it for your group or process.

Logical name translation is a recursive procedure: when the system translates a logical name, it uses the equivalence name as the argument for another logical name translation. It continues in this way until it cannot translate the equivalence name.

Assume that your program updates monthly sales files (for example, JAN.DAT, FEB.DAT, MAR.DAT, and so forth). Your SELECT statement could look like either of these:

```
SELECT SALES-FILE ASSIGN TO "MOSLS"
```

```
SELECT SALES-FILE ASSIGN TO MOSLS
```

To update the January sales file, you can use this ASSIGN command to equate the equivalence name JAN.DAT with the logical name MOSLS:

```
$  ASSIGN JAN.DAT MOSLS
```

To update the February sales file, you can use this ASSIGN command:

```
$  ASSIGN FEB.DAT MOSLS
```

In the same way, all programs that access the monthly sales file can use the logical name MOSLS.

To disassociate the relationship between the file and the logical name, you can use this DEASSIGN command:

```
$  DEASSIGN FEB.DAT MOSLS
```

If MOSLS is not set as a logical name, the system uses it as a file specification and looks for a file named MOSLS.DAT.

8.5.2 Choosing File Organization and Record Access Mode

Your program always states—either explicitly or implicitly—a file's organization and access mode before the program opens the file. The ORGANIZATION and ACCESS clauses of the FILE-CONTROL paragraph, if present, specify these two attributes.

8.5.2.1 File Organizations

VAX COBOL supports three types of file organization:

- ORGANIZATION IS SEQUENTIAL—This organization requires that records be referenced in the same sequence in which they were written. This organization is useful for programs that normally access each record serially, as in a payroll or mailing list file.
- ORGANIZATION IS RELATIVE—This organization lets you access records randomly, according to their key values (relative record numbers). This organization is less flexible than indexed organization because you cannot insert a record in the middle of your file unless you have an empty cell to contain it.
- ORGANIZATION IS INDEXED—This organization lets you access records randomly, according to their key values. This is a useful way to organize a file in which records will be added, changed, or deleted upon demand.

Table 8–1 lists the three file organizations available to you and summarizes their advantages and disadvantages. Chapter 9, Chapter 10, and Chapter 11 further discuss each of these file organizations.

Table 8–1: VAX COBOL File Organizations—Advantages and Disadvantages

File Organizations		Advantages and Disadvantages
Sequential	Advantages	Uses disk and memory efficiently Provides optimal usage if the application accesses all records sequentially on each run Provides the most flexible record format Allows READ/WRITE sharing Allows data to be stored on many types of media, in a device-independent manner Allows easy file extension
	Disadvantages	Allows sequential access only Allows records to be added only to the end of a file
Relative	Advantages	Allows sequential, random, and dynamic access Provides random record deletion and insertion Allows records to be READ/WRITE sharing
	Disadvantages	Allows data to be stored on disk only Requires that record cells be the same size
Indexed	Advantages	Allows sequential, random, and dynamic access modes Allows random record deletion and insertion Allows READ/WRITE sharing Allows variable-length records to change length on update Allows easy file extension
	Disadvantages	Allows data to be stored on disk only Requires more disk space Uses more memory to process records Generally requires multiple disk accesses to randomly process a record

If you do not use the ORGANIZATION clause, VAX COBOL assumes the file organization is sequential.

8.5.2.2 Record Access Modes

The methods for retrieving and storing records in a file are called record access modes. VAX COBOL supports three types of record access modes:

- ACCESS MODE IS SEQUENTIAL
  - With sequential files, sequential access retrieves the records in the same sequence established by the WRITE statements that created the file.
  - With relative files, sequential access retrieves the records in the order of ascending record key values (or relative record numbers).
  - With indexed files, sequential access retrieves records in the order of ascending record key values.
- ACCESS MODE IS RANDOM—The value of the record key your program specifies indicates the record to be accessed.

- ACCESS MODE IS DYNAMIC—This access mode allows you to switch from sequential access mode to random access mode and back to sequential access mode while processing a file, by using the NEXT phrase on the READ statement. You can switch back and forth as much as you like; the only limitation is that there must be RELATIVE or INDEXED ORGANIZATION.

If you do not use the ACCESS clause, VAX COBOL assumes sequential access.

A different access mode can be used to process records within the file each time it is opened. A program can also change access modes during the processing of its file. Chapter 9, Chapter 10, and Chapter 11 discuss the access modes applicable to sequential, relative, and indexed file organization, respectively.

Example 8–11 shows sample SELECT statements for sequential files with sequential access modes.

Example 8–11: Sequential File SELECT Statements

```
                  (1)                                      (2)
FILE-CONTROL.                              FILE-CONTROL.
    SELECT LIST-FILE                           SELECT PAYROLL
           ASSIGN TO "MAIL.LIS"                       ASSIGN TO "PAYROL.DAT".
           ORGANIZATION IS SEQUENTIAL
           ACCESS IS SEQUENTIAL.
```

VAX COBOL assumes sequential organization and sequential access unless you specify otherwise.

Sample SELECT statements for relative files are shown in Example 8–12.

Example 8–12: Relative File SELECT Statements

```
                  (1)                                      (2)
FILE-CONTROL.                             FILE-CONTROL.
    SELECT MODEL                              SELECT PARTS
           ASSIGN TO "ACTOR.DAT"                     ASSIGN TO "PART.DAT"
           ORGANIZATION IS RELATIVE                  ORGANIZATION IS RELATIVE
           ACCESS MODE IS SEQUENTIAL.                ACCESS MODE IS DYNAMIC
                                                     RELATIVE KEY IS PART-NO.
```

Sample SELECT statements for indexed files are shown in Example 8–13.

Because the default organization is sequential, both the relative and indexed examples require the ORGANIZATION clause.

Example 8–13: Indexed File SELECT Statements

```
                   (1)                                        (2)
FILE-CONTROL.                                 FILE-CONTROL.
    SELECT A-GROUP                                SELECT TEAS
           ASSIGN TO "RFCBA.PRO"                         ASSIGN TO "TETLY"
           ORGANIZATION IS INDEXED                       ORGANIZATION IS INDEXED
           ACCESS MODE IS DYNAMIC                        RECORD KEY IS LEAVES.
           RECORD KEY IS WRITER
           ALTERNATE RECORD KEY IS EDITOR.
```

8.6 Opening and Closing Files

A VAX COBOL program must open a file with the OPEN statement before any other I/O or Report Writer statement can reference it. Files can be opened more than once in the same program as long as they are closed before the second and subsequent opens.

Opening a file allocates the buffers, creates or checks the file labels, and initializes the data structures to the start of the file. Closing a file writes out any remaining records in the output buffers, writes an end-of-file label on magnetic output files, and optionally rewinds and/or locks magnetic tape files. Files that remain open at program termination are closed by the Run-Time System. The following example shows OPEN and CLOSE statements:

```
OPEN INPUT MASTER-FILE.
OPEN OUTPUT REPORT-FILE.
OPEN I-O   MASTER-FILE
           TRANS-FILE
     OUTPUT REPORT-FILE.
CLOSE MASTER-FILE.
CLOSE TRANS-FILE
      REPRT-FILE.
```

The OPEN statement must specify one of four open modes: INPUT, OUTPUT, I-O, or EXTEND. Your choice, along with the file's organization and access mode, determines which I/O statements you can use. Section 9.3, Section 10.3, and Section 11.3 discuss the I/O statements for sequential, relative, and indexed files, respectively.

When your program performs an OPEN statement, the following events take place:

1. RMS builds a file specification by using the contents of the VALUE OF ID clause, if any, to alter or complete the file specification in the ASSIGN clause. Logicals are translated and the default file type is DAT.
2. The Run-Time System checks the file's current status. If the file is open, or if it was closed WITH LOCK, the OPEN statement fails.
3. If the file specification names an invalid device, or contains any other errors, the Run-Time System generates an error message and the OPEN statement fails.
4. The Run-Time System takes one of the following actions if it cannot find the file:
    a. If the file's OPEN mode is OUTPUT, the file is created.
    b. If the file's OPEN mode is EXTEND, or I-O, the OPEN statement fails, unless the file's SELECT clause includes the OPTIONAL phrase. If the file's SELECT clause includes the OPTIONAL phrase, the file is created.

   c. If the file's OPEN mode is INPUT, and its SELECT clause includes the OPTIONAL phrase, the OPEN statement is successful. The first read on that file causes the AT END condition.

   d. If none of the previous conditions is met, the OPEN fails and the USE procedure (if any) gains control. If no USE procedure exists, the Run-Time System aborts the program.

5. If the file's OPEN mode is OUTPUT, and a file by the same name already exists, a new version is created.
6. If the file attributes specified by the program attempting an OPEN operation differ from the attributes specified when the file was created, the OPEN statement fails.

If the file is on magnetic tape, RMS rewinds it. To close a file on tape without rewinding the tape, use the NO REWIND phrase. This speeds processing when another file is to be written beyond the end of the first file. For example:

```
CLOSE MASTER-FILE NO REWIND.
```

You can also close a file and prevent it from being opened again by the program in the same run. For example:

```
CLOSE MASTER-FILE WITH LOCK.
```

8.7 File Compatibility

Files created by different programming languages may require special processing because of language and character set incompatibilities. The most common incompatibilities are data types and data record formats.

8.7.1 Data Type Differences

Data types vary by programming language and by utilities. For example, VAX FORTRAN does not support the packed-decimal data type and, therefore, cannot easily use PACKED-DECIMAL data in COBOL files.

You can use the following techniques to overcome data type incompatibilities:

- Use the NATIVE character set, which uses ASCII representation, for all data in files intended for use across languages.
- If your requirements include processing non-ASCII data, you can specify a character set in: (1) the SPECIAL-NAMES paragraph of the Environment Division, along with (2) the CODE-SET clause in the SELECT statement. Except for NATIVE, you must specify all character sets in the SPECIAL-NAMES paragraph.
- Use common numeric data types (numeric data types that remain constant across the application).

In the following example, the input file is written in EBCDIC. This creates a file that would be difficult to handle in most other languages.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.   ALPHABET FOREIGN-CODE IS EBCDIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT INPUT-FILE ASSIGN TO "INPFIL"
            CODE-SET IS FOREIGN-CODE.
            .
            .
            .
```

8.7.2 Data Record Formatting Differences

Programming languages and system utilities differ in the conventions used to format their data records. For example, FORTRAN programs place a carriage control character before the first data character in a formatted file record. Similarly, other languages format print-controlled (and other) records differently from COBOL.

In some cases, you can avoid this incompatibility by not using print-controlled files. In FORTRAN, for example, you can open a file with the CARRIAGE CONTROL= 'NONE' specification. Alternately, you can still read it by defining record descriptions that match the actual format; that is, by defining a 1-character data item before the first real data item in each record operation. The 1-character field can be interpreted or ignored during subsequent read operations.

8.8 Backing Up Your Files

If your disk file becomes corrupted by a hardware error, or with bad data, or if your program abnormally terminates when the file is opened for output, the file can become unusable. Proper backup procedures are the key to successful recovery.

You should back up your disk file at some reasonable point (daily, weekly, or monthly, depending on file activity and value of data), and save all transactions until you create a new backup. In this way, you can easily re-create your disk file from your last backup file and transaction files whenever the need arises.

8.9 Low-Volume I/O (ACCEPT and DISPLAY)

The COBOL language provides two statements, ACCEPT and DISPLAY, for low-volume I/O operations. In most system configurations, these statements transfer data to and from a terminal device. In COBOL however, the ACCEPT and DISPLAY statements refer to VMS logical names.

This section discusses the use of mnemonic names, logical name devices, and the ACCEPT and DISPLAY statements. For further information on ACCEPT and DISPLAY screen handling options, refer to Chapter 17.

8.9.1 Mnemonic Names (SPECIAL-NAMES Paragraph)

The ACCEPT and DISPLAY statements transfer data between your program and logical names. If you do not use the FROM or UPON clauses, the default logical names are SYS$INPUT for the ACCEPT statement and SYS$OUTPUT for the DISPLAY statement.

The FROM or UPON clauses refer to mnemonic names that you can define in the SPECIAL-NAMES paragraph in the Environment Division. You define a mnemonic name by equating it to a COBOL implementor name; for example, the following clause equates STATUS-REPORT to the device LINE-PRINTER:

```
LINE-PRINTER IS STATUS-REPORT
```

You can then use the mnemonic name in a DISPLAY statement:

```
DISPLAY "File contains " REC-COUNT UPON STATUS-REPORT.
```

8.9.2 Logical Name Devices

The COBOL implementor names in the SPECIAL-NAMES paragraph represent VMS logical names.

COBOL Implementor Names	Logical Name
CARD-READER	COB$CARDREADER
PAPER-TAPE-READER	COB$PAPERTAPEREADER
CONSOLE	COB$CONSOLE
LINE-PRINTER	COB$LINEPRINTER
PAPER-TAPE-PUNCH	COB$PAPERTAPEPUNCH

The logical names do not always represent physical devices. You can, for example, assign a logical name to a file specification with a VMS ASSIGN command:

```
ASSIGN [ALLSTATUS]STATUS.LIS COB$LINEPRINTER
```

Because a logical name does not imply a device, it carries no implication of open mode. Therefore, a program can display upon a mnemonic name that refers to CARD-READER or accept from a mnemonic name that refers to LINE-PRINTER.

Although the ACCEPT and DISPLAY statements do not refer to file names, the system implicitly opens a logical name used in either of these statements.

NOTE

When the system opens a logical name for a DISPLAY statement, it specifies the variable with fixed-length control (VFC) format to allow carriage control. Therefore, if your program contains both ACCEPT and DISPLAY statements that refer to the same logical name, it should execute a DISPLAY before the first ACCEPT. Otherwise, DISPLAY statement carriage control is lost and all DISPLAY statements execute as if they contained the WITH NO ADVANCING phrase.

Carriage control characters are not lost when you use ACCEPT and DISPLAY statements without the FROM or UPON clause, since these statements refer to different logical names (SYS$INPUT and SYS$OUTPUT).

8.9.3 ACCEPT Statement

In the *VAX COBOL Reference Manual*, Formats 1, 3, and 4 of the ACCEPT statement transfer data from the object of a VMS logical name to a data item. If you do not use the FROM clause, the system uses the logical name SYS$INPUT. Otherwise, it uses the logical name described in the SPECIAL-NAMES paragraph and referenced in the ACCEPT statement. In the following example, the system uses COB$CONSOLE:

```
SPECIAL-NAMES.
    CONSOLE IS WHATS-HIS-NAME
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
    ACCEPT PARAMETER-AREA FROM WHATS-HIS-NAME.
```

8.9.4 DISPLAY Statement

The DISPLAY statement transfers the contents of low-volume data items and literals to the object of a VMS logical name. If you do not use the UPON clause, the system uses the logical name SYS$OUTPUT. Otherwise, it uses the logical name described in the SPECIAL-NAMES paragraph and referenced by the DISPLAY statement. In the following example, the system uses COB$LINEPRINTER:

```
SPECIAL-NAMES.
    LINE-PRINTER IS ERROR-LOG
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
    DISPLAY ERROR-COUNT, " phase 2 errors, ",
            ERROR-MSG UPON ERROR-LOG.
```

8.10 Printing with VAX VFP

VAX COBOL, in conjunction with the VAX Vertical Forms Printing (VFP) software utility, provides direct support of Vertical Form Unit (VFU)-supported printers. These VFU-supported printers offer the following useful features:

- Rapid vertical line positioning
- The ability to customize the vertical spacing requirements of your application, independent of the application itself. When you use a VFU-supported printer, you can configure the vertical spacing requirements of your application by associating these requirements with a particular VFU printer channel.

To take advantage of these features, follow these steps:

1. Configure the vertical spacing requirements you want in the printer's VFU.

2. Reference this specific VFU channel number in the WRITE statement of your VAX COBOL application. The VAX COBOL compiler makes an association with that VFU channel when the record is written. (For detailed information about using VFU channel numbers in WRITE statements, see the *VAX COBOL Reference Manual.*)
3. Convert the file that contains the record you want to print using the VAX VFP software utility. If you do not do this conversion, the VFU will have no effect on the vertical spacing of that record. Refer to the VAX VFP software utility documentation for information about converting VAX COBOL data files with records containing VFU channel commands to printer-specific VFU channel commands.
4. Print the converted record on a VFU-supported printer. The record is printed according to the vertical spacing requirements that you specified in the printer's VFU.

You can change the vertical spacing requirements of your application without having to change the application itself. You can make these independent changes in vertical spacing because you specify the vertical spacing requirements in the printer's VFU and not in the COBOL application.

NOTE

> You cannot use files that contain VFU spacing on versions of VMS earlier than 5.2. If you try to print such files on pre-5.2 versions, you may encounter undefined print symbiont behavior.

Chapter 9

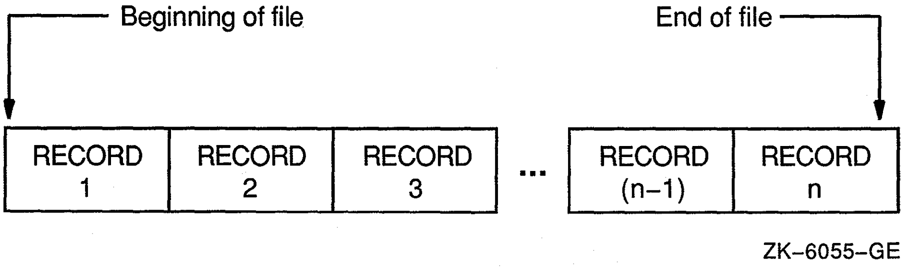
Processing Sequential Files

Sequential input/output, in which records are written and read in sequence, is the simplest and most common form of I/O. It can be performed on all I/O devices, including magnetic tape, disk, terminals, and line printers.

9.1 Sequential File Organization

In sequential file organization, records are arranged in the order in which they were written to the file. Figure 9–1 illustrates sequential file organization.

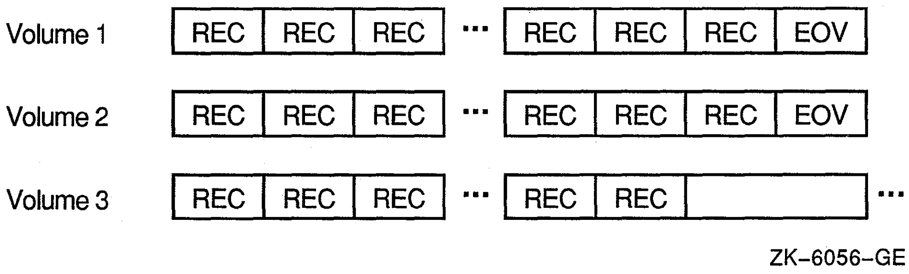
Figure 9–1: Sequential File Organization



Sequential files always contain an end-of-file (EOF) indication. On magnetic tapes, it is the EOF mark; on disk, it is a counter in the file header that designates the end of the file. VAX COBOL statements can write over the EOF mark and, thus, extend the length of the file. Because the EOF indicates the end of useful data, VAX COBOL provides no method for reading beyond it, even though the amount of space reserved for the file exceeds the amount actually used.

Occasionally a file with sequential organization, for example, a multiple-reel magnetic tape file, is so large that it requires more than one volume. An end-of-volume (EOV) label marks the end of recorded information on each volume and signals the file system to switch to a new volume. On multiple-volume files, the EOF mark appears only once, at the end of the last record on the last volume. See Figure 9–2.

Figure 9–2: A Multiple-Volume Sequential File


Volume 1
REC REC REC ... REC REC REC EOV
Volume 2
REC REC REC ... REC REC REC EOV
Volume 3
REC REC REC ... REC REC ...
ZK-6056-GE

See the VMS documentation on magnetic tapes for more information on tape formats.

9.2 Design Considerations

Before you create your sequential file applications, you should design your files based on these design considerations:

1. Record format (see Chapter 8).
   - Fixed-length
   - Variable-length
2. Medium—Sequential files can be accessed on disk, magnetic tape, and unit record devices (for example, printers and card readers). When you select the medium for your file, consider the following:
   - Speed of access—Tape is significantly slower than disk.
   - Frequency of use—Use tape to store files and save your disk space for more immediate purposes.
   - Cost of medium—Disk is generally more expensive than tape. The more frequently the data will be accessed, the more justification there is to use a more costly medium.
   - Transportability—Use tape files if you need to use the file across systems that have no common disk devices.
3. Allocation—At time of file creation and file extension.
4. Compiler limitations—You want to consider the logical and physical limits imposed by the VAX COBOL compiler.

For more information on sequential file design, see Chapter 20, and the VAX documentation on RMS tuning.

9.3 Statements for Sequential File Processing

Processing a sequential file involves the following:

1. Opening the file with the OPEN statement
2. Processing the file with valid I/O statements

3. Closing the file with the CLOSE statement

Table 9–1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 9–1: Valid I/O Statements for Sequential Files

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I/O**	**EXTEND**
SEQUENTIAL	SEQUENTIAL	READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	No	Yes
		UNLOCK	Yes	Yes	Yes	Yes

9.4 Defining a Sequential File

Each sequential file in a VAX COBOL program is given a file name in a separate SELECT clause in the Environment Division. Refer to Example 9–1 for these file names: MASTER-FILE, TRANS-FILE, and REPRT-FILE. These names are referred to by statements in the VAX COBOL program.

The ASSIGN clause associates the file name with a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device. For example:

- MASTER-FILE is located on disk unit DB1:, directory [DOE.LRM], and is called MASTER.DAT.
- TRANS-FILE is located on magnetic tape unit 1, directory [DOE.LRM], and is called TRANS.DAT.
- REPRT-FILE is assigned to the line printer.

Each file is further described in the program with a file description (FD) entry in the File Section of the Data Division (for example, MASTER-FILE, TRANS-FILE, and REPRT-FILE). The FD entry is followed immediately by the file's record description (for example, MASTER-RECORD, TRANSACTION-RECORD, and REPORT-LINE).

You need not specify either the ORGANIZATION IS SEQUENTIAL phrase or the ACCESS MODE IS SEQUENTIAL phrase in the SELECT clause since VAX COBOL assumes sequential organization and sequential access mode unless you specify otherwise.

Example 9–1: Defining a Sequential File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   MASTER-FILE   ASSIGN   TO   "DB1:[DOE.LRM]MASTER.DAT".
    SELECT   TRANS-FILE    ASSIGN   TO   "MTA1:[DOE.LRM]TRANS.DAT".
    SELECT   REPRT-FILE    ASSIGN   TO   "LP0:".
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE...
01  MASTER-RECORD.
    02  MASTER-DATA        PIC X(80).
    02  MASTER-SIZE        PIC 99.
    02  MASTER-TABLE       OCCURS 0 to 50 TIMES
                           DEPENDING ON MASTER-SIZE.
        03  MASTER-YEAR    PIC 99.
        03  MASTER-COUNT   PIC S9(5)V99.
FD  TRANS-FILE...
01  TRANSACTION-RECORD     PIC X(25).
FD  REPRT-FILE...
01  REPORT-LINE            PIC X(132).
```

9.5 Creating a Sequential File

A VAX COBOL program creates a sequential file by:

1. Opening the file as OUTPUT or EXTEND
2. Executing the WRITE statement

Each WRITE statement releases a logical record to the end of an output file, thereby creating an entirely new record in the file. The WRITE statement releases records to files that are OPEN in the following modes:

- OUTPUT—The output mode can create these two kinds of files:
  - Storage files—A storage file remains on tape or disk for future reference or processing.
  - Print files—The LINAGE clause, APPLY PRINT-CONTROL clause, Report Writer statements (via RWCS), or the ADVANCING phrase in the WRITE statement designates a file as a print file. One or more records containing a VFC header, which indicates carriage-control characters, are written to perform line spacing. The WRITE statement does not have to release print files directly to a storage file. It can release them directly to the printer for immediate printing. A storage file can also be a print file.
- EXTEND—The extend mode permits new records to be added in sequence after the last record of an existing file (see Section 9.8).

You can write records in the following two ways:

- WRITE record-name FROM source-area
- WRITE record-name

However, the first way provides easier program readability when working with multiple record types. For example, statements (1) and (2) in this example are logically equivalent:

```
FILE SECTION.
FD  STOCK-FILE.
01  STOCK-RECORD       PIC X(80).
WORKING-STORAGE SECTION.
01  STOCK-WORK         PIC X(80).

----------------(1)----------------     --------------(2)---------------
WRITE STOCK-RECORD FROM STOCK-WORK.     MOVE STOCK-WORK TO STOCK-RECORD.
                                        WRITE STOCK-RECORD.
```

When you omit the FROM phrase, you process the records directly in the record area or buffer (for example, STOCK-RECORD).

The following example writes the record PRINT-LINE to the device assigned to that record's file, then skips three lines. When it reaches the end of the page (as specified by the LINAGE clause), it causes program control to transfer to HEADER-ROUTINE.

```
WRITE PRINT-LINE BEFORE ADVANCING 3 LINES
      AT END-OF-PAGE PERFORM HEADER-ROUTINE.
```

For a WRITE FROM statement, if the destination area is shorter than the file's record length, the destination area is padded on the right with spaces; if longer, the destination area is truncated on the right.

Example 9–2 creates a sequential file.

Example 9–2: Creating a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TRANS-FILE ASSIGN TO "TRANS.DAT".
DATA DIVISION.
FILE SECTION.
FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT TRANS-FILE.
    PERFORM A010-PROCESS-TRANS
       UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A010-PROCESS-TRANS.
    DISPLAY "Enter next record  - X(25)".
    DISPLAY "enter END to terminate the session".
    DISPLAY "-------------------------".
    ACCEPT TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END"
       WRITE TRANSACTION-RECORD.
```

9.6 Reading a Sequential File

To read a sequential file you must do the following:

1. Open the file for INPUT or I/O.
2. Execute the READ statement.

Each READ statement reads a single logical record and makes its contents available to the program in the record area. There are two ways of reading records:

- READ file-name INTO destination-area
- READ file-name

In the following example, statements (1) and (2) are logically equivalent:

```
FILE SECTION.
FD  STOCK-FILE.
01  STOCK-RECORD      PIC X(80).
WORKING-STORAGE SECTION.
01  STOCK-WORK        PIC X(80).

-------------(1)---------------    -------------(2)---------------
READ STOCK-FILE INTO STOCK-WORK.   READ STOCK-FILE.
                                   MOVE STOCK-RECORD TO STOCK-WORK.
```

When you omit the INTO phrase, you process the records directly in the record area or buffer (for example, STOCK-RECORD). The record is also available in the record area if you use the INTO phrase.

In a READ INTO clause, if the destination area is shorter than the length of the record area being read, the record is truncated on the right and a warning is issued; if longer, the destination area is filled on the right with blanks.

If the data in the record being read is shorter than the length of the record (for example, a variable-length record), the contents of the record beyond that data are undefined.

Example 9–3 reads a sequential file and displays its contents on the terminal.

Example 9–3: Reading a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TRANS-FILE ASSIGN TO "TRANS.DAT".
DATA DIVISION.
FILE SECTION.
FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT TRANS-FILE.
    PERFORM A100-READ-TRANS-FILE
       UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A100-READ-TRANS-FILE.
    READ TRANS-FILE
       AT END MOVE "END" TO TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END"
       DISPLAY TRANSACTION-RECORD.
```

9.7 Updating Records in a Sequential File

To update a record in a sequential file you must do the following:

1. Open the file for I/O.
2. Read the target record.
3. Rewrite the target record.

The REWRITE statement places the record just read back into the file. The REWRITE statement completely replaces the contents of the target record with new data. You can use the REWRITE statement for files on mass storage devices only (for example, disk units). There are two ways of rewriting records:

- REWRITE record-name FROM source-area
- REWRITE record-name

In the following example, statements (1) and (2) are logically equivalent:

```
FILE SECTION.
FD  STOCK-FILE.
01  STOCK-RECORD      PIC X(80).
WORKING-STORAGE SECTION.
01  STOCK-WORK        PIC X(80).

---------------(1)------------------    --------------(2)--------------
REWRITE STOCK-RECORD FROM STOCK-WORK.   MOVE STOCK-WORK TO STOCK-RECORD.
                                        REWRITE STOCK-RECORD.
```

When you omit the FROM phrase, you process the records directly in the record area or buffer (for example, STOCK-RECORD).

For a REWRITE statement, the record being rewritten must be the same length as the record being replaced.

Example 9–4 reads a sequential file and rewrites as many records as the operator wants.

Example 9–4: Rewriting a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TRANS-FILE ASSIGN TO "TRANS.DAT".
DATA DIVISION.
FILE SECTION.
FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).
WORKING-STORAGE SECTION.
01  ANSWER                PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O TRANS-FILE.
    PERFORM A100-READ-TRANS-FILE
       UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A100-READ-TRANS-FILE.
    READ TRANS-FILE AT END
       MOVE "END" TO TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END"
       PERFORM A300-GET-ANSWER UNTIL ANSWER = "Y" OR "N"
       PERFORM A200-REWRITE-RECORD.
A200-REWRITE-RECORD.
    IF ANSWER = "Y" DISPLAY "Please enter new record content"
       ACCEPT TRANSACTION-RECORD
       REWRITE TRANSACTION-RECORD.
A300-GET-ANSWER.
    DISPLAY "Do you want to replace this record? -- "
             TRANSACTION-RECORD.
    DISPLAY "Please answer Y or N".
    ACCEPT ANSWER.
```

9.8 Extending a Sequential File

To position a file to its current end, and to allow the program to write new records beyond the last record in the file, use both:

- The EXTEND phrase of the OPEN statement
- The WRITE statement

Example 9–5 shows how to extend a sequential file.

Example 9–5: Extending a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TRANS-FILE ASSIGN TO "TRANS.DAT".
DATA DIVISION.
FILE SECTION.
FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN EXTEND TRANS-FILE.
    PERFORM A100-WRITE-RECORD
       UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A100-WRITE-RECORD.
    DISPLAY "Enter next record  - X(25)".
    DISPLAY "Enter END to terminate the session".
    DISPLAY "-------------------------".
    ACCEPT TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END"
       WRITE TRANSACTION-RECORD.
```

Without the EXTEND phrase, a VAX COBOL program would have to do the following tasks:

1. Open the input file.
2. Copy it to an output file.
3. Add records to the output file.

Chapter 10

Processing Relative Files

A relative file consists of fixed-size record cells and uses a key to retrieve its records. The key, or record key, is an integer that specifies the record's storage cell within the file. It is analogous to the subscript of a table.

Unlike sequential files, where retrieving the twentieth record involves reading the previous 19 records first, relative files can directly access the twentieth record with one READ operation. In addition, relative files allow the program to read forward or backward with respect to the current record key.

Another significant fact of relative file processing is that not every cell must contain a record. Although each cell occupies one record space, a field preceding the record on the storage medium indicates whether or not that cell contains a valid record. Thus, a file can contain fewer records than it has cells, and the empty cells can be anywhere in the file.

The numerical order of the cells remains the same during all operations on a relative file; however, accessing statements can move a record from one cell to another, delete a record from a cell, insert new records into empty cells, or rewrite existing cells.

Relative file processing is available only on disk devices.

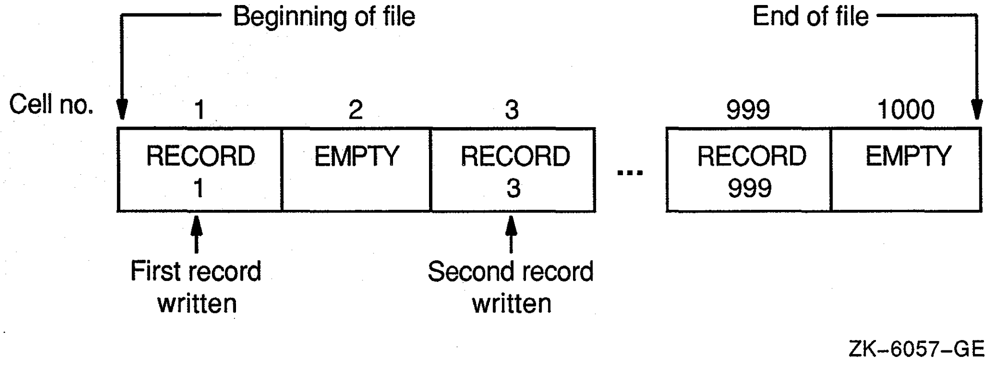
10.1 Relative File Organization

With relative file processing, RMS structures a file as a series of fixed-sized record cells. Cell size is based on the size specified as the maximum permitted length for a record in the file. RMS considers these cells as successively numbered from 1 (the first) to n (the last). A cell's relative record number (RRN) represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. Empty cells can be interspersed among cells containing records.

Since cell numbers in a relative file are unique, they can be used to identify both the cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 21 occupies the twenty-first cell, and so forth. Figure 10–1 depicts the structure of relative file organization.

Figure 10–1: Relative File Organization



Relative files have three capabilities not available with sequential files:

- Random access by record key
- Record deletion by record key
- Record updating by record key

Relative files are used primarily when records must be accessed in random order and the records can easily be associated with a sequential number. When a program creates a relative file, RMS allocates disk space for each cell. No additional space in the cell can be added thereafter unless you recreate the file. However, since records can be replaced, you can insert empty records at first, then replace them later with real records, which gives the effect of adding records. After a program creates a relative file, it can be updated by replacing or deleting records. Records are replaced by rewriting the new record over (on top of) the old one.

Relative files are used like tables. Their advantage over tables is that their size is limited to disk space rather than memory space. Also, their information can be saved from run to run. Relative files are best for records that are easily associated with ascending, consecutive numbers (so that the program conversion from data to cell number is easy), such as years (the years 71 to 90 could be stored with record keys 1 to 20), months (record keys 1 to 12), or the 50 U.S. states (record keys 1 to 50).

10.2 Design Considerations

Before you create your relative file applications, you should design your file based on these design considerations:

1. Record format (see Chapter 8).
    - Fixed-length
    - Variable-length

    Relative files can contain either fixed-length records or variable-length records; however, RMS calculates a cell size equal to the maximum record size plus overhead bytes, resulting in fixed-length storage. Once created, relative records can be accessed sequentially, randomly, or dynamically.

2. Medium—Relative files can be accessed on disk only. Make sure the disk pack is large enough to meet your current and future needs.
3. Allocation at time of file creation and file extension.
4. Bucket size—To optimize the packing of cells into buckets, cell size should be evenly divisible into bucket size.
5. Maximum number of records.
6. Compiler limitations—Consider the logical and physical limits imposed by the VAX COBOL compiler.
7. Key scheme.

For more information on relative file design, see Chapter 20, and the VMS documentation on RMS tuning.

10.3 Statements for Relative File Processing

Processing a relative file involves the following:

1. Opening the file with the OPEN statement
2. Setting the relative record number
3. Processing the file with valid I/O statements
4. Closing the file with the CLOSE statement

Table 10–1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 10–1: Valid I/O Statements for Relative Files

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
RELATIVE	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	Yes
		UNLOCK	Yes	Yes	Yes	Yes

(continued on next page)

Table 10–1 (Cont.): Valid I/O Statements for Relative Files

File			Open Mode			
Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No
		READ NEXT	Yes	No	Yes	No

10.4 Defining a Relative File

Each relative file in a VAX COBOL program is given a file name in a SELECT clause in the Environment Division.

The ASSIGN clause associates the file name with a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device (see HINZ.DAT in Example 10–1). Each file is further described in the program with a file description (FD) entry in the File Section of the Data Division (see FLAVORS in Example 10–1). The FD entry is followed immediately by the file's record description (see KETCHUP-MASTER in Example 10–1).

You must specify the ORGANIZATION IS RELATIVE phrase in the SELECT clause; otherwise, VAX COBOL assumes sequential organization. You must also specify the RELATIVE KEY IS phrase and assign a relative key data name for random or dynamic access.

Example 10–1: Defining a Relative File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.
```

10.5 Creating a Relative File

A VAX COBOL program creates a relative file by performing the following tasks:

1. Specifying ORGANIZATION IS RELATIVE in the SELECT clause
2. Specifying either of the following access modes in the SELECT clause:
   - Sequential access
   - Random access

     Each of these two access mode choices requires a different processing technique. The next two sections discuss those techniques.
3. Opening the file as:
   - OUTPUT—The only function of a WRITE statement with output files is to place entirely new records into the file. If a file requires more space, RMS automatically extends the file size, regardless of the access mode.
   - I-O—With input/output files, the WRITE statement places records into cells that already exist and contain no valid record.
4. Initializing the relative key data name for each record to be written
5. Executing the WRITE statement for each new relative record
6. Closing the file

10.5.1 Sequential Access Mode Creation

When a program creates a relative file in sequential access mode, RMS does not use the relative key. Instead, it writes the first record in the file at relative record number 1, the second record at relative record number 2, and so on, until the program closes the file. If you use the RELATIVE KEY IS clause, the compiler moves the relative record number of the record being written to the relative key data item. Example 10–2 writes 10 records with relative record numbers 1 to 10.

Example 10–2: Creating a Relative File in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER.
    02  FILLER          PIC X(14).
    02  REC-NUM         PIC 9(05).
    02  FILLER          PIC X(31).
WORKING-STORAGE SECTION.
01  REC-COUNT           PIC S9(5) VALUE 0.
```

(continued on next page)

Example 10–2 (Cont.): Creating a Relative File in Sequential Access Mode

```
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT FLAVORS.
    PERFORM A010-WRITE 10 TIMES.
    CLOSE FLAVORS.
    STOP RUN.
A010-WRITE.
    MOVE "Record number" TO KETCHUP-MASTER.
    ADD 1 TO REC-COUNT.
    MOVE REC-COUNT TO REC-NUM.
    WRITE KETCHUP-MASTER
          INVALID KEY DISPLAY "BAD WRITE"
                      STOP RUN.
```

10.5.2 Random Access Mode Creation

When a program creates a relative file using random access mode, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. Example 10–3 shows how to supply the relative key. It writes 10 records in the cells numbered: 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20. Record cells 1, 3, 5, 7, 9, 11, 13, 15, 17, and 19 are also created, but contain no valid record.

Example 10–3: Creating a Relative File in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER.
    02  FILLER            PIC X(14).
    02  REC-NUM           PIC 9(05).
    02  FILLER            PIC X(31).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY    PIC 99.
01  REC-COUNT             PIC S9(5) VALUE 0.
```

(continued on next page)

Example 10–3 (Cont.): Creating a Relative File in Random Access Mode

```
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT FLAVORS.
    MOVE 0 TO KETCHUP-MASTER-KEY.
    PERFORM A010-CREATE-RELATIVE-FILE 10 TIMES.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-CREATE-RELATIVE-FILE.
    ADD 2 TO KETCHUP-MASTER-KEY.
    MOVE "Record number" TO KETCHUP-MASTER.
    ADD 2 TO REC-COUNT.
    MOVE REC-COUNT TO REC-NUM.
    WRITE KETCHUP-MASTER
          INVALID KEY DISPLAY "BAD WRITE"
                      STOP RUN.
```

10.6 Reading a Relative File

Your program can read a relative file three ways:

- Sequentially
- Randomly
- Dynamically

10.6.1 Sequential Reading

To read relative records sequentially, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS SEQUENTIAL clause.
3. Open the file for INPUT or I-O.
4. Read records as you would a sequential file, or use the START statement.

The READ statement makes the next logical record of an open file available to the program. The system reads the file sequentially from either: (1) cell 1 or (2) wherever you START the file, up to cell n. It skips the empty cells and retrieves only valid records. Each READ statement updates the contents of the file's RELATIVE KEY data item, if specified. The data item contains the relative number of the available record. When the At End condition occurs, execution of the READ statement is unsuccessful (see Chapter 12).

Sequential processing need not begin at the first record of a relative file. The START statement specifies the next record to be read and positions the file position indicator for subsequent I/O operations.

Example 10–4 reads a relative file sequentially, displaying every record on the terminal.

Example 10–4: Reading a Relative File Sequentially

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.
01  END-OF-FILE             PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.
    PERFORM A010-DISPLAY-RECORDS UNTIL END-OF-FILE = "Y".
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DISPLAY-RECORDS.
    READ FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY KETCHUP-MASTER.
```

10.6.2 Random Reading

To read relative records randomly, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS RANDOM or DYNAMIC clause.
3. Open the file for INPUT or I-O.
4. Move the relative record number value to the RELATIVE KEY data name.
5. Read the record from the cell identified by the relative record number.

Example 10–5 reads a relative file randomly, displaying every record on the terminal.

The READ statement selects a specific record from an open file and makes it available to the program. The value of the relative key identifies the specific record. The system reads the record identified by the RELATIVE KEY data name clause. If the cell does not contain a valid record, the invalid key condition occurs, and the READ operation fails (see Chapter 12).

Example 10–5: Reading a Relative File Randomly

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL05.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99 VALUE 99.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.
    PERFORM A100-DISPLAY-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A100-DISPLAY-RECORD.
    DISPLAY "TO DISPLAY A RECORD ENTER ITS RECORD NUMBER (ZERO to END)".
    ACCEPT KETCHUP-MASTER-KEY WITH CONVERSION.
    IF KETCHUP-MASTER-KEY > 00
       READ FLAVORS
            INVALID KEY DISPLAY "BAD KEY"
                        CLOSE FLAVORS
                        STOP RUN
       END-READ
       DISPLAY KETCHUP-MASTER.
```

10.6.3 Dynamic Reading

The READ statement has two formats so that it can select the next logical record (sequentially) or select a specific record (randomly) and make it available to the program. In dynamic mode, the program can switch from random access I/O statements to sequential access I/O statements in any order, without closing and reopening files. However, you must use the READ NEXT statement to sequentially read a relative file open in dynamic mode.

Sequential processing need not begin at the first record of a relative file. The START statement positions the file position indicator for subsequent I/O operations.

A sequential read of a dynamic file is indicated by the NEXT phrase of the READ statement. A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record indicated by the current record pointer. Subsequent READ NEXT statements sequentially retrieve records until another START statement or random READ statement executes.

Example 10–6 processes a relative file containing 10 records. If the previous program examples in this chapter have been run, each record has a unique even number from 2 to 20 as its key. The program positions the record pointer (using the START statement) to the cell corresponding to the value in INPUT-RECORD-KEY. The program's READ...NEXT statement retrieves the remaining valid records in the file for display on the terminal.

Example 10–6: Reading a Relative File Dynamically

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL06.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS DYNAMIC
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99.
01  END-OF-FILE               PIC X   VALUE "N".
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    DISPLAY "Enter number".
    ACCEPT KETCHUP-MASTER-KEY.
    START FLAVORS KEY = KETCHUP-MASTER-KEY
          INVALID KEY DISPLAY "Bad START statement"
          GO TO A005-END-OF-JOB.
    PERFORM A010-DISPLAY-RECORDS UNTIL END-OF-FILE = "Y".
A005-END-OF-JOB.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DISPLAY-RECORDS.
    READ FLAVORS NEXT RECORD AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY KETCHUP-MASTER.
```

10.7 Updating a Relative File

A program updates a relative file with the WRITE, REWRITE, and DELETE statements. The WRITE statement adds a record to the file. Only the REWRITE and DELETE statements change the contents of records already existing in the file. In either case, adequate backup must be available in the event of error. The next two sections explain how to rewrite and delete relative records.

10.7.1 Rewriting Relative Records

Two options are available for rewriting relative records:

- Sequential access mode rewriting
- Random access mode rewriting

The REWRITE statement logically replaces a record in a relative file. After successfully rewriting a record into the file, the program can access that record at any time. However, the program cannot access the record that occupied the cell previous to the rewrite operation.

10.7.1.1 Sequential Access Mode Rewriting

To rewrite relative records in sequential access mode, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS SEQUENTIAL clause.
3. Open the file for I-O.
4. Read the target record, or use the START statement and then the READ statement to sequentially read the file up to the target record.
5. Update the target record.
6. Rewrite the target record into its cell.

The REWRITE statement places the successfully read record back into its cell in the file.

Example 10–7 reads a relative record sequentially and displays the record on the terminal. The program then passes the record to an update routine that is not included in the example. The update routine updates the record, and passes the updated record back to the program illustrated in Example 10–7, which displays the updated record on the terminal and rewrites the record in the same cell.

Example 10–7: Rewriting Relative Records in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL07.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99 VALUE 99.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A100-UPDATE-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
```

(continued on next page)

Example 10–7 (Cont.): Rewriting Relative Records in Sequential Access Mode

```
A100-UPDATE-RECORD.
    DISPLAY "TO UPDATE A RECORD ENTER ITS RECORD NUMBER (ZERO to END)".
    ACCEPT KETCHUP-MASTER-KEY WITH CONVERSION.
    IF KETCHUP-MASTER-KEY IS NOT EQUAL TO 00
       START FLAVORS KEY IS EQUAL TO KETCHUP-MASTER-KEY
             INVALID KEY DISPLAY "BAD START"
                         STOP RUN
       END-START
       PERFORM A200-READ-FLAVORS
       DISPLAY  "*********BEFORE UPDATE*********"
       DISPLAY KETCHUP-MASTER
************************************************************
*
*      Update routine code here
*
************************************************************
       DISPLAY  "*********AFTER UPDATE*********"
       DISPLAY KETCHUP-MASTER
       REWRITE KETCHUP-MASTER.
A200-READ-FLAVORS.
    READ FLAVORS
         AT END DISPLAY "END OF FILE"
                GO TO A005-EOJ.
```

10.7.1.2 Random Access Mode Rewriting

To rewrite relative records in random access mode, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS RANDOM or DYNAMIC clause.
3. Open the file for I-O.
4. Move the relative record number value of the record you want to read to the RELATIVE KEY data name.
5. Read the record from the cell identified by the relative record number.
6. Update the record.
7. Rewrite the record into the cell identified by the relative record number.

The system randomly reads the record identified by the KEY IS clause. The REWRITE statement places the successfully read record back into its cell in the file.

If the cell does not contain a valid record, or if the REWRITE operation is unsuccessful, the invalid key condition occurs, and the REWRITE operation fails (see Chapter 12).

Example 10–8 reads a relative record randomly, displays its contents on the terminal, updates the record, displays its updated contents on the terminal, and rewrites the record in the same cell.

Example 10–8: Rewriting Relative Records in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL08.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A100-UPDATE-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A100-UPDATE-RECORD.
    DISPLAY "TO UPDATE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    READ FLAVORS INVALID KEY DISPLAY "BAD READ"
                        GO TO A005-EOJ.
    DISPLAY  "*********BEFORE UPDATE*********".
    DISPLAY KETCHUP-MASTER.
********************************************************
*
*               Update routine
*
********************************************************
    DISPLAY  "*********AFTER UPDATE*********".
    DISPLAY KETCHUP-MASTER.
    REWRITE KETCHUP-MASTER INVALID KEY DISPLAY "BAD REWRITE"
                                       GO TO A005-EOJ.
```

10.7.2 Deleting Relative Records

Two options are available for deleting relative records:

- Sequential access mode deletion
- Random access mode deletion

The DELETE statement logically removes an existing record from a relative file. After successfully removing a record from a file, the program cannot later access it.

10.7.2.1 Sequential Access Mode Deletion

To delete a relative record in sequential access mode, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS SEQUENTIAL clause.
3. Open the file for I-O.

4. Either (a) use the START statement to position the record pointer or (b) sequentially read the file up to the target record.
5. Delete the last read record.

Example 10–9 is an example of deleting relative records in sequential access mode.

Example 10–9: Deleting Relative Records in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL09.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER           PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY       PIC 99 VALUE 1.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A010-DELETE-RECORDS UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DELETE-RECORDS.
    DISPLAY "TO DELETE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    IF KETCHUP-MASTER-KEY NOT = 00 PERFORM A200-READ-FLAVORS
                                   DELETE FLAVORS RECORD.
A200-READ-FLAVORS.
    START FLAVORS
        INVALID KEY DISPLAY "INVALID START"
                    STOP RUN.
    READ FLAVORS AT END DISPLAY "FILE AT END"
                        GO TO A005-EOJ.
```

10.7.2.2 Random Access Mode Deletion

To delete a relative record in random access mode, you must do the following:

1. Specify the ORGANIZATION IS RELATIVE clause.
2. Specify the ACCESS MODE IS RANDOM clause.
3. Open the file I-O.
4. Move the relative record number value to the RELATIVE KEY data name.
5. Delete the record identified by the relative record number.

If the file does not contain a valid record, an invalid key condition exists.

Example 10–10 is an example of deleting relative records in random access mode.

Example 10–10: Deleting Relative Records in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL10.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99 VALUE 1.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A010-DELETE-RECORDS UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DELETE-RECORDS.
    DISPLAY "TO DELETE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    IF KETCHUP-MASTER-KEY NOT = 00
       DELETE FLAVORS RECORD
              INVALID KEY DISPLAY "INVALID DELETE"
                          STOP RUN.
```

Chapter 11

Processing Indexed Files

Unlike the sequential ordering of records in a sequential file or the relative positioning of records in a relative file, the location of records in indexed file organization is transparent to the program. It is possible to add new records to an indexed file and logically place them between physically adjacent records without re-creating the file. Not only can records be added, but they can also be deleted, making room for new records.

RMS controls the placement of records in an indexed file based on user-specified primary and alternate keys in the record itself. The presence of keys in the records of the file governs this placement. This is the only file organization where RMS uses the actual contents of the records for record placement within the file.

Indexed file processing is available only on disk devices.

11.1 Indexed File Organization

VAX COBOL allows sequential, random, and dynamic access to records. Each record is accessed by one of its primary or alternate keys.

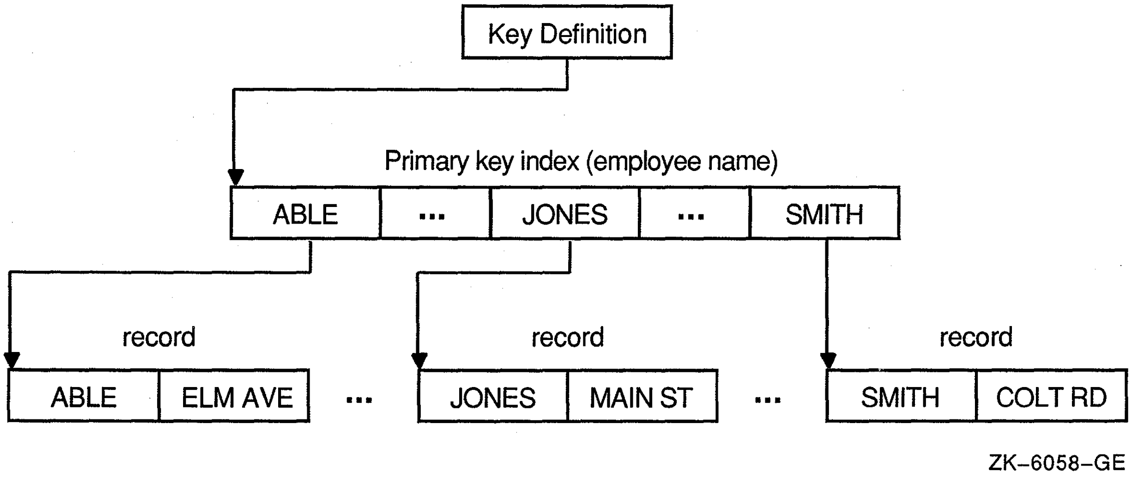
A major feature of indexed file organization is the use of a key to uniquely identify a record within the file. Its location and length are identical in all records. When creating an indexed file, you must select the data items to be the keys. Selecting such a data item indicates to RMS that the contents (key value) of that string in any record written to the file can be used by the program to identify that record for subsequent retrieval. For more information, see the RECORD KEY IS clause and the ALTERNATE RECORD KEY IS clause in the *VAX COBOL Reference Manual.*

You must define at least one main key, called the **primary key**, for an indexed file. VAX COBOL also allows you to optionally define from 1 to 254 additional keys called **alternate keys**. Each alternate key represents an additional character string in each record of the file. The key value in any of these additional strings can also be used as a means of identifying the record for retrieval.

You define primary and alternative key values in the Record Description entry. Primary and alternate key values need not be unique if you specify the WITH DUPLICATES phrase in the file description entry. When duplicate key values are present, you can retrieve the records in the order that they were written. The logical sort order of each key can be either ascending (the default) or descending. The logical sort order controls the order of sequential processing of the record.

As your program writes records into an indexed file, RMS locates the values contained in the primary and alternate keys. RMS builds these values into a tree-structured table known as an *index* (or B-Tree), which consists of a series of entries. Each entry contains a key value copied from a record. With each key value is a pointer to the location in the file of the record from which the value was copied. Figure 11–1 shows the general structure of an indexed file defined with a primary key only.

Figure 11–1: Indexed File Organization



For a more detailed explanation of indexed file structure, see the VMS documentation on RMS tuning.

11.2 Design Considerations

Before you create your indexed file applications, you should design your file based on these design considerations:

1. Record format (see Chapter 8).
   - Fixed-length
   - Variable-length
2. Medium—Indexed files can be accessed on disk only.
3. Allocation at the time of file creation and file extension (see Chapter 20).
4. Speed—You want to maximize the speed with which the program processes data.
5. Space—You want to minimize file size, disk space, and memory requirements to run your program.
6. Shared access—You must consider who is going to use the data and how they will access the data.
7. Ease of design—You want to minimize the time spent writing the application.
8. Compiler limitations—You want to consider the logical and physical limits imposed by the VAX COBOL compiler.

For more information on indexed file design optimization, see Chapter 20 and the VMS documentation on RMS tuning. If you do not carefully design your index file—that is, if you take all the file defaults—your indexed file application may run more slowly than you expect.

11.3 Statements for Indexed File Processing

Processing an indexed file involves the following tasks:

1. Opening the file with the OPEN statement
2. Processing the file with valid I/O statements
3. Closing the file with the CLOSE statement

Table 11–1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 11–1: Valid I/O Statements for Indexed Files

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
INDEXED	SEQUENTIAL	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	No	Yes
		UNLOCK	Yes	Yes	Yes	Yes
	RANDOM	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		WRITE	No	Yes	Yes	No
		UNLOCK	Yes	Yes	Yes	No
	DYNAMIC	DELETE	No	No	Yes	No
		READ	Yes	No	Yes	No
		REWRITE	No	No	Yes	No
		START	Yes	No	Yes	No
		WRITE	No	Yes	Yes	No
		READ NEXT	Yes	No	Yes	No
		UNLOCK	Yes	Yes	Yes	No

11.4 Defining an Indexed File

Each indexed file in a VAX COBOL program is given a file name in a SELECT clause in the Environment Division. The ASSIGN clause associates the file name with a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device (see Example 11–1, DAIRY.DAT). Each file is further described in the program with a file description (FD) entry in the File Section of the Data Division (see Example 11–1, FLAVORS). The FD entry is followed immediately by the file's record description (see Example 11–1, ICE-CREAM-MASTER). Refer to the *VAX COBOL Reference Manual* for information relating to the RECORD KEY and ALTERNATE RECORD KEY clauses.

Example 11–1 defines a dynamic access mode indexed file with one primary key (ICE-CREAM-MASTER-KEY) and two alternate record keys (ICE-CREAM-STORE-CODE, and ICE-CREAM-STORE-STATE). Note that one alternate record key allows duplicates (ICE-CREAM-STORE-STATE). Any program using the identical entries in the SELECT clause as shown in Example 11–1 can reference the DAIRY.DAT file sequentially and randomly.

Example 11–1: Defining an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "DAIRY.DAT"
             ORGANIZATION IS INDEXED
             ACCESS MODE IS DYNAMIC
             RECORD KEY IS ICE-CREAM-MASTER-KEY
             ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                              WITH DUPLICATES
             ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.
PROCEDURE DIVISION.
A00-BEGIN.
```

You must specify the ORGANIZATION IS INDEXED phrase; otherwise, the default is ORGANIZATION IS SEQUENTIAL. You specify ACCESS MODE IS... in the SELECT clause, depending on how you want to access the file (SEQUENTIAL, RANDOM, DYNAMIC).

11.5 Creating and Populating an Indexed File

A VAX COBOL program creates an indexed file by:

1. Specifying ORGANIZATION IS INDEXED in the SELECT clause.
2. Specifying either of the following access modes in the SELECT clause:
   - Sequential access—The program can write records in ascending or descending order by primary key, depending on the sort order.
   - Random or dynamic access—The program can write records in any order.

3. Opening the file for:
    - OUTPUT—To add records only
    - I-O—To add, change, or delete records
4. Initializing the key values.
5. Executing the WRITE statement.

The best way to initially populate an indexed file is to sequentially write the records in ascending order by primary key.

The program can add records to the file until it reaches the physical limitations of its storage device. When this occurs, you should: (1) delete unnecessary records, (2) back up the file, and (3) recreate the file either by using the CONVERT Utility to optimize file space, or by using a VAX COBOL program. For more information, see the VMS documentation on the CONVERT Utility.

Example 11–2 creates and populates an indexed file (DAIRY.DAT). The file (DAIRYI.DAT) has been sorted in ascending sequence. Notice that the primary and alternate keys are initialized in ICE-CREAM-MASTER when the contents of the fields in INPUT-RECORD are read into ICE-CREAM-MASTER before the record is written.

Example 11–2: Creating and Populating an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "DAIRYI.DAT".
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS SEQUENTIAL
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                         WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-RECORD-KEY             PIC 9999.
    02  INPUT-RECORD-DATA            PIC X(47).
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.
WORKING-STORAGE SECTION.
01  END-OF-FILE                      PIC X.
```

(continued on next page)

Example 11–2 (Cont.): Creating and Populating an Indexed File

```
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT INPUT-FILE.
    OPEN OUTPUT FLAVORS.
A010-POPULATE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.
A100-READ-INPUT.
    READ INPUT-FILE INTO ICE-CREAM-MASTER
         AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       WRITE ICE-CREAM-MASTER INVALID KEY DISPLAY "BAD WRITE"
                                          STOP RUN.
```

11.6 Reading an Indexed File

Your program can read an indexed file in the following three ways:

- Sequentially
- Randomly
- Dynamically

However, to read the file randomly, the program must: (1) initialize either the primary key data name or the alternate key data name before reading the target record, and (2) specify that data name in the KEY IS phrase of the READ statement.

Dynamic access permits switching back and forth from sequential access to random access any number of times during one OPEN of the file.

11.6.1 Sequential Reading

To read indexed records in a sequential mode, you must do the following:

1. Specify the ORGANIZATION IS INDEXED in the SELECT clause.
2. Specify the ACCESS MODE IS SEQUENTIAL clause.
3. Open the file for INPUT or I-O.
4. Read records from the beginning of the file as you would a sequential file—use the READ...AT END statement.

The READ statement makes the next logical record of an open file available to the program. It skips deleted records and sequentially reads and retrieves only valid records. When the at end condition occurs, execution of the READ statement is unsuccessful (see Chapter 12).

Example 11–3 reads the entire indexed file sequentially beginning with the first record in the file, displaying every record on the terminal.

Example 11–3: Reading an Indexed File Sequentially

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS SEQUENTIAL
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.
WORKING-STORAGE SECTION.
01  END-OF-FILE                      PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.
A010-SEQUENTIAL-READ.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.
A100-READ-INPUT.
    READ  FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       DISPLAY ICE-CREAM-MASTER
       STOP "Type CONTINUE to display next master".
```

11.6.2 Random Reading

To read indexed records randomly, you must do the following:

1. Specify the ORGANIZATION IS INDEXED clause.
2. Specify the ACCESS MODE IS RANDOM clause.
3. Open the file for INPUT or I-O.
4. Initialize the RECORD KEY or ALTERNATE RECORD KEY data name before reading the record.
5. Read the record using the KEY IS clause.

The READ statement selects a specific record from an open file and makes it available to the program. The value of the primary or alternate key identifies the specific record. The system randomly reads the record identified by the KEY clause. If RMS does not find a valid record, the invalid key condition occurs, and the READ statement fails (see Chapter 12).

Example 11–4 reads an indexed file randomly, displaying its contents on the terminal.

Example 11–4: Reading an Indexed File Randomly

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS DYNAMIC
                      RECORD KEY IS ICE-CREAM-KEY.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-KEY          PIC XXXX.
    02 ICE-CREAM-DATA.
       03  ICE-CREAM-STORE-CODE       PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS    PIC X(20).
       03  ICE-CREAM-STORE-CITY       PIC X(20).
       03  ICE-CREAM-STORE-STATE      PIC XX.
WORKING-STORAGE SECTION.
01  PROGRAM-STAT                      PIC X.
    88  OPERATOR-STOPS-IT             VALUE "1".
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A020-INITIAL-PROMPT.
    IF OPERATOR-STOPS-IT
       PERFORM A005-TERMINATE.
    PERFORM A030-RANDOM-READ.
    PERFORM A025-SUBSEQUENT-PROMPTS UNTIL OPERATOR-STOPS-IT.
    PERFORM A005-TERMINATE.
A005-TERMINATE.
    DISPLAY "END OF JOB".
    STOP RUN.
A020-INITIAL-PROMPT.
    DISPLAY "Do you want to see a store?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT = "Y" OR "y" OR "N" OR "n".
    IF PROGRAM-STAT = "N" OR "n"
       MOVE "1" TO PROGRAM-STAT.
A025-SUBSEQUENT-PROMPTS.
    MOVE SPACE TO PROGRAM-STAT.
    DISPLAY "Do you want to see another store ?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT = "Y" OR "y" OR "N" OR "n".
    IF PROGRAM-STAT = "Y" OR "y"
       PERFORM A030-RANDOM-READ
    ELSE
       MOVE "1" TO PROGRAM-STAT.
A030-RANDOM-READ.
    DISPLAY "Enter key".
    ACCEPT ICE-CREAM-KEY.
    PERFORM A100-READ-INPUT-BY-KEY.
A040-GET-ANSWER.
    DISPLAY "Please answer Y or N"
    ACCEPT PROGRAM-STAT.
```

(continued on next page)

Example 11–4 (Cont.): Reading an Indexed File Randomly

```
A100-READ-INPUT-BY-KEY.
    READ FLAVORS KEY IS ICE-CREAM-KEY
         INVALID KEY DISPLAY "Record does not exist - Try again"
         NOT INVALID KEY DISPLAY "The record is: ", ICE-CREAM-MASTER.
```

11.6.3 Dynamic Reading

The READ statement has two formats, so it can select the next logical record (sequentially) or select a specific record (randomly) and make it available to the program. In dynamic mode, the program can switch from using random access I/O statements to sequential access I/O statements, in any order, without closing and reopening files. However, the program must use the READ NEXT statement to sequentially read an indexed file opened in dynamic mode.

Sequential processing need not begin at the first record of an indexed file. The START statement specifies the next record to be read sequentially and positions the file position indicator for subsequent I/O operations anywhere within the file.

A sequential read of a dynamic file is indicated by the NEXT phrase of the READ statement. A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record indicated by the file position indicator. Subsequent READ NEXT statements sequentially retrieve records until another START statement or random READ statement executes.

Example 11–5 processes an indexed file containing 26 records. Each record has a unique letter of the alphabet as its primary key. The program positions the file to the first record whose INPUT-RECORD-KEY is equal to the specified letter of the alphabet. The program's READ NEXT statement sequentially retrieves the remaining valid records in the file for display on the terminal.

Example 11–5: Reading an Indexed File Dynamically

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX05.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IND-ALPHA  ASSIGN TO "ALPHA.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS DYNAMIC
                      RECORD KEY IS INPUT-RECORD-KEY.
DATA DIVISION.
FILE SECTION.
FD  IND-ALPHA.
01  INPUT-RECORD.
    02  INPUT-RECORD-KEY            PIC X.
    02  INPUT-RECORD-DATA           PIC X(50).
WORKING-STORAGE SECTION.
01  END-OF-FILE                     PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O IND-ALPHA.
    DISPLAY "Enter letter"
```

(continued on next page)

Example 11–5 (Cont.): Reading an Indexed File Dynamically

```
    ACCEPT INPUT-RECORD-KEY.
    START IND-ALPHA KEY = INPUT-RECORD-KEY
          INVALID KEY DISPLAY "BAD START STATEMENT"
          GO TO A010-END-OF-JOB.
    PERFORM A100-GET-RECORDS THROUGH A100-GET-RECORDS-EXIT
            UNTIL END-OF-FILE = "Y".
A010-END-OF-JOB.
    DISPLAY "END OF JOB".
    CLOSE IND-ALPHA.
    STOP RUN.
A100-GET-RECORDS.
    READ IND-ALPHA NEXT RECORD AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY INPUT-RECORD.
A100-GET-RECORDS-EXIT.
    EXIT.
```

11.7 Updating an Indexed File

To update a record in an indexed file, your program must do the following:

- In sequential access mode:
  1. Read the target record.
  2. Verify that this record is indeed the record you want to change.
  3. Change the record.
  4. Rewrite or delete the record.
- In random access mode: rewrite or delete the record.

Your program can update an indexed file three ways:

- Sequentially
- Randomly
- Dynamically

NOTE

A program cannot rewrite an existing record if it changes the contents of the primary key in that record. Instead, the program must delete the record and write a new record. Alternate key values can be changed at any time. However, the value of alternate keys must be unique unless the WITH DUPLICATES phrase is present.

11.7.1 Sequential Updating

To update indexed records in a sequential mode, you must do the following:

1. Specify the ORGANIZATION IS INDEXED clause.
2. Specify the ACCESS MODE IS SEQUENTIAL clause.
3. Open the file for I-O.
4. Read records as you would a sequential file, that is, use the READ statement with the AT END phrase.

5. Rewrite or delete records using the INVALID KEY phrase.

The READ statement makes the next logical record of an open file available to the program. It skips deleted records and sequentially reads and retrieves only valid records. When the at end condition occurs, execution of the READ statement is unsuccessful (see Chapter 12).

The REWRITE statement replaces the record just read, while the DELETE statement logically removes the record just read from the file.

Example 11–6 is an example of a sequential update of an indexed file.

Example 11–6: Updating an Indexed File Sequentially

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX06.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS      ASSIGN TO "DAIRY.DAT"
                        ORGANIZATION IS INDEXED
                        ACCESS MODE IS SEQUENTIAL
                        RECORD KEY IS ICE-CREAM-MASTER-KEY
                        ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                                WITH DUPLICATES
                        ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY           PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE       PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS    PIC X(20).
       03  ICE-CREAM-STORE-CITY       PIC X(20).
       03  ICE-CREAM-STORE-STATE      PIC XX.
WORKING-STORAGE SECTION.
01  END-OF-FILE                       PIC X.
01  REWRITE-KEY                       PIC XXXXX.
01  DELETE-KEY                        PIC XX.
01  NEW-ADDRESS                       PIC X(20).
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    DISPLAY "Which store code do you want to find?".
    ACCEPT REWRITE-KEY.
    DISPLAY "What is its new address?".
    ACCEPT NEW-ADDRESS.
    DISPLAY "Which state do you want to delete?".
    ACCEPT DELETE-KEY.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
```

(continued on next page)

Example 11–6 (Cont.): Updating an Indexed File Sequentially

```
A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.
A100-READ-INPUT.
    READ  FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" AND
       REWRITE-KEY = ICE-CREAM-STORE-CODE
       PERFORM A200-REWRITE-MASTER.
    IF END-OF-FILE NOT = "Y" AND
       DELETE-KEY  = ICE-CREAM-STORE-STATE
       PERFORM A300-DELETE-MASTER.
A200-REWRITE-MASTER.
    MOVE NEW-ADDRESS TO ICE-CREAM-STORE-ADDRESS.
    REWRITE ICE-CREAM-MASTER
            INVALID KEY DISPLAY "Bad rewrite - ABORTED"
                        STOP RUN.
A300-DELETE-MASTER.
    DELETE FLAVORS.
```

11.7.2 Random Updating

To update indexed records in a random mode, you must do the following:

1. Specify the ORGANIZATION IS INDEXED clause.
2. Specify the ACCESS MODE IS RANDOM clause.
3. Open the file for I-O.
4. Initialize the RECORD KEY or ALTERNATE RECORD KEY data name.
5. Write, rewrite, or delete records using the INVALID KEY phrase.

   Note that if the primary or alternate key specified in step 4 allows duplicates, only the first occurrence of a record with a matching value will be updated.

Example 11–7 is an example of a random update of an indexed file.

Example 11–7: Updating an Indexed File Randomly

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX07.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS RANDOM
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
```

(continued on next page)

Example 11–7 (Cont.): Updating an Indexed File Randomly

```
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.
WORKING-STORAGE SECTION.
01  HOLD-ICE-CREAM-MASTER            PIC X(51).
01  PROGRAM-STAT                     PIC X.
    88  OPERATOR-STOPS-IT            VALUE "1".
    88  LETS-SEE-NEXT-STORE          VALUE "2".
    88  NO-MORE-DUPLICATES           VALUE "3".
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A030-RANDOM-READ UNTIL OPERATOR-STOPS-IT.
A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.
A030-RANDOM-READ.
    DISPLAY "Enter key".
    ACCEPT ICE-CREAM-MASTER-KEY.
    PERFORM A100-READ-INPUT-BY-PRIMARY-KEY
            THROUGH A100-READ-INPUT-EXIT.
    DISPLAY " Do you want to terminate the session?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT   = "Y" OR "N".
    IF PROGRAM-STAT   = "Y" MOVE "1" TO PROGRAM-STAT.
A040-GET-ANSWER.
        DISPLAY "Please answer Y or N"
        ACCEPT PROGRAM-STAT.
A100-READ-INPUT-BY-PRIMARY-KEY.
    READ FLAVORS KEY IS ICE-CREAM-MASTER-KEY
         INVALID KEY DISPLAY "Master does not exist - Try again"
         GO TO A100-READ-INPUT-EXIT.
    DISPLAY ICE-CREAM-MASTER.
    PERFORM A200-READ-BY-ALTERNATE-KEY UNTIL NO-MORE-DUPLICATES.
A100-READ-INPUT-EXIT.
    EXIT.
A200-READ-BY-ALTERNATE-KEY.
    DISPLAY "Do you want to see the next store in this state?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT   = "Y" OR "N".
    IF PROGRAM-STAT   = "Y"
       MOVE "2" TO PROGRAM-STAT
       READ FLAVORS KEY IS ICE-CREAM-STORE-STATE
                    INVALID KEY DISPLAY "No more stores in this state"
                                MOVE "3" TO PROGRAM-STAT.
```

(continued on next page)

Example 11–7 (Cont.): Updating an Indexed File Randomly

```
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "NY"
             PERFORM A500-DELETE-RANDOM-RECORD.
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "NJ"
             MOVE "Monmouth" TO ICE-CREAM-STORE-CITY
             PERFORM A400-REWRITE-RANDOM-RECORD.
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "CA"
             MOVE ICE-CREAM-MASTER TO HOLD-ICE-CREAM-MASTER
             PERFORM A500-DELETE-RANDOM-RECORD
             MOVE HOLD-ICE-CREAM-MASTER TO ICE-CREAM-MASTER
             MOVE "AZ" TO ICE-CREAM-STORE-STATE
             PERFORM A300-WRITE-RANDOM-RECORD.
    IF PROGRAM-STAT   = "N"
       MOVE "3" TO PROGRAM-STAT.
A300-WRITE-RANDOM-RECORD.
    WRITE ICE-CREAM-MASTER
          INVALID KEY DISPLAY "Bad write - ABORTED"
                      STOP RUN.
A400-REWRITE-RANDOM-RECORD.
    REWRITE ICE-CREAM-MASTER
            INVALID KEY DISPLAY "Bad rewrite - ABORTED"
                        STOP RUN.
A500-DELETE-RANDOM-RECORD.
    DELETE FLAVORS
           INVALID KEY DISPLAY "Bad delete - ABORTED"
                       STOP RUN.
```

11.7.3 Dynamic Updating

In dynamic mode, the program can switch from using random access I/O statements to sequential access I/O statements in any order without closing and reopening files. To dynamically update indexed records, you must do the following:

1. Specify the ORGANIZATION IS INDEXED clause.
2. Specify the ACCESS MODE IS DYNAMIC clause.
3. Open the file for I-O.
4. Read the records in one of two ways:
    - Sequentially—Use the START statement to position the record pointer, and then use the READ...NEXT statement.
    - Randomly—Initialize the RECORD KEY or ALTERNATE RECORD KEY data name, and then read records in any order you want using the INVALID KEY phrase.
5. Write, rewrite, or delete records using the INVALID KEY phrase.

Chapter 12

Input/Output Exception Conditions Handling

Many types of exception conditions can occur when a program processes a file; not all of them are errors. The three categories of exception conditions are as follows:

- At end condition—This is a normal condition when you access a file sequentially. However, if your program tries to read the file any time after having read the last logical record in the file, and there is no applicable Declarative procedure or AT END phrase, the program abnormally terminates when the next READ statement executes.
- Invalid key condition—When you process relative and indexed files, the invalid key condition is a normal condition if you plan for it with a Declarative procedure or INVALID KEY phrase. It is an abnormal condition that causes your program to terminate if there is no applicable Declarative procedure or INVALID KEY phrase.
- All other conditions—These can also be either normal conditions (if you plan for them) or abnormal conditions that cause your program to terminate.

Planning for exception conditions effectively increases program and programmer efficiency. A program with exception handling routines is more flexible than a program without them. They minimize operator intervention and often reduce or eliminate the time a programmer uses to debug and rerun the program.

This chapter introduces you to the tools you need to execute sequential, relative, and indexed file exception handling routines as a normal part of your program. The tools you need are as follows:

- The AT END phrase
- The INVALID KEY phrase
- File Status values
- Special registers—RMS-CURRENT-STS, RMS-CURRENT-STV, RMS-STS, and RMS-STV
- Declarative procedures

12.1 Planning for the At End Condition

VAX COBOL provides you the option of testing for this condition with the AT END phrase of the READ statement for sequential, relative, and indexed files and the ACCEPT statement.

Programs often read sequential files from beginning to end. They can produce reports from the information in the file or even update it. However, the program must be able to detect the end of the file, so that it can continue normal processing at that point. If the program does not test for this condition when it occurs, and if no applicable Declarative procedure exists (see Section 12.4), the program terminates abnormally. The program must detect when no more data is available from the file so that it can perform its normal end-of-job totaling, balancing, and closing of the file.

Example 12–1 shows the use of the AT END phrase with the READ statement.

Example 12–1: Handling the At End Condition

```
READ SEQUENTIAL-FILE AT END PERFORM A600-TOTAL-ROUTINES
                            PERFORM A610-VERIFY-TOTALS-ROUTINES
                            MOVE "Y" TO END-OF-FILE.
READ RELATIVE-FILE NEXT RECORD AT END PERFORM A700-CLEAN-UP-ROUTINES
                                      CLOSE RELATIVE-FILE
                                      STOP RUN.
READ INDEXED-FILE NEXT RECORD AT END DISPLAY "End of file"
                                     DISPLAY "Do you want to continue?"
                                     ACCEPT REPLY
                                     PERFORM A700-CLEAN-UP-ROUTINES.
```

12.2 Planning for the Invalid Key Condition

An invalid key condition occurs whenever RMS cannot complete a VAX COBOL DELETE, READ, REWRITE, START, or WRITE statement. When the condition occurs, execution of the statement that recognized it is unsuccessful, and the file is not affected.

For example, relative and indexed files use keys to retrieve records. The program specifying random access must initialize a key before executing a DELETE, READ, REWRITE, START, or WRITE statement. If the key does not result in the successful execution of any of these statements, the invalid key condition exists. This condition is fatal to the program, if the program does not check for the condition when it occurs and if no applicable Declarative procedure exists (see Section 12.4).

The invalid key condition, although fatal if not planned for, can be to your advantage when used properly. You can, as in Example 12–2, read through an indexed file for all records with a specific duplicate key and produce a report from the information in those records. However, after you have read the last of the duplicate records, you receive an invalid key condition for subsequent read operations to indicate that no more records with this key exist in the file. Planning for the invalid key condition in this case allows the program to continue its normal processing.

Example 12–2: Handling the Invalid Key Condition

```
          .
          .
          .
          MOVE "SMITH" TO LAST-NAME.
          MOVE "Y" TO ANY-MORE-DUPLICATES.
          PERFORM A500-READ-DUPLICATES-ROUTINE
                  UNTIL ANY-MORE-DUPLICATES = "N".
          .
          .
          .
          STOP RUN.
     A500-READ-DUPLICATES-ROUTINE.
          READ INDEXED-FILE RECORD INTO HOLD-RECORD
               KEY IS LAST-NAME
               INVALID KEY DISPLAY "Name not in file!" STOP RUN.
          PERFORM A510-READ-NEXT-DUPLICATES-ROUTINE
                  UNTIL ANY-MORE-DUPLICATES = "N".
     A510-READ-NEXT-DUPLICATES-ROUTINE.
          READ INDEXED-FILE NEXT RECORD
               AT END MOVE "N" TO ANY-MORE-DUPLICATES.
          IF ANY-MORE-DUPLICATES = "Y" PERFORM A700-PRINT-ROUTINES.
     MOVE "N" TO ANY-MORE-DUPLICATES.
          .
          .
          .
     A700-PRINT-ROUTINES.
          .
          .
          .
```

12.3 Using File Status Values

Your program can check for the specific cause of the failure of a file operation by checking for specific File Status values in its exception handling routines. To obtain File Status values from VAX COBOL, use the FILE STATUS clause in the file description entry. To provide FILE STATUS values from RMS, use the VAX COBOL special registers RMS-STS and RMS-STV or RMS-CURRENT-STS and RMS-CURRENT-STV.

12.3.1 VAX COBOL File Status Values

The run-time execution of any VAX COBOL file processing statement results in a RMS completion code value that reports the success or failure of the COBOL statement. To access this value, you must specify the FILE STATUS clause in the file description entry, as shown in Example 12–3.

Example 12–3: Defining a File Status for a File

```
DATA DIVISION.
FILE SECTION.
FD  INDEXED-FILE
*
    FILE STATUS IS INDEXED-FILE-STATUS.
*
01  INDEXED-RECORD         PIC X(50).
WORKING-STORAGE SECTION.
01  INDEXED-FILE-STATUS    PIC XX.
01  ANSWER                 PIC X.
```

The program can access this File Status variable, INDEXED-FILE-STATUS, anywhere in the Procedure Division, and depending on its value, take a specific course of action without terminating the program. Notice that Example 12–4 uses the File Status defined in Example 12–3. However, not all statements allow you to access the File Status value as part of the statement. Your program has two options:

- Examine the status value as part of an error recovery routine built into the statement. The only relative and indexed file processing statements that allow you to do this within the INVALID KEY phrase are DELETE, READ, REWRITE, START, and WRITE. See Example 12–4.
- Define a Declarative procedure to handle the condition (see Section 12.4). All file organizations and their I/O statements have this option available.

Example 12–4: Using the File Status Value in an Exception Handling Routine

```
PROCEDURE DIVISION.
A000-BEGIN.
    .
    .
    .
    DELETE INDEXED-RECORD
           INVALID KEY MOVE "Bad DELETE" to BAD-VERB-ID
                       PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
```

(continued on next page)

Example 12–4 (Cont.): Using the File Status Value in an Exception Handling Routine

```
    READ INDEXED-FILE NEXT RECORD
         INVALID KEY MOVE "Bad READ" TO BAD-VERB-ID
                     PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    REWRITE INDEXED-RECORD
            INVALID KEY MOVE "Bad REWRITE" TO BAD-VERB-ID
                        PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    START INDEXED-FILE KEY IS EQUAL TO MASTER-KEY
          INVALID KEY MOVE "Bad START" TO BAD-VERB-ID
                      PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    WRITE INDEXED-RECORD
          INVALID KEY MOVE "Bad WRITE" TO BAD-VERB-ID
                      PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
A900-EXCEPTION-HANDLING-ROUTINE.
    DISPLAY BAD-VERB-ID " - File Status Value = " INDEXED-FILE-STATUS.
    PERFORM A905-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.
A905-GET-ANSWER.
    DISPLAY "Do you want to continue?"
    DISPLAY "Please answer Y or N"
    ACCEPT ANSWER.
```

Each file processing statement described in the Procedure Division section of the *VAX COBOL Reference Manual* contains a specific list of File Status values in its Technical Notes section. Additionally, all File Status values are listed in an appendix of the Reference Manual.

12.3.2 RMS File Status Values

VAX COBOL checks for RMS completion codes after each file and record operation. If the code indicates anything other than unconditional success, VAX COBOL maps the RMS error code to a File Status value. However, not all RMS completion codes map to distinct File Status values. Many RMS completion codes map to a File Status value of 30, a COBOL code for errors that have no corresponding File Status value.

VAX COBOL provides four special registers, RMS-STS, RMS-STV, RMS-CURRENT-STS, and RMS-CURRENT-STV. These registers supplement the File Status values already available and allow the VAX COBOL program to directly access RMS completion codes. RMS-CURRENT-STS and RMS-CURRENT-STV contain the file status values from the most recent file or record operation for any file. For more information, refer to the VMS documentation on RMS completion codes.

You need not define these registers in your program. As special registers, they are available whenever and wherever you need to use them in the Procedure Division. However, if you define more than one file in the program and intend to access RMS-STS and RMS-STV, you must qualify your references to them. RMS-CURRENT-STS and RMS-CURRENT-STV contain the file status values for the most recent file or record operation for any file. So when you access RMS-CURRENT-STS and RMS-CURRENT-STV, you must not qualify your reference to them.

Notice the use of the WITH CONVERSION phrase of the DISPLAY statement in Example 12–5. This converts the PIC S9(9) COMP special registers from binary to decimal digits for terminal display.

Example 12–5: Referencing RMS-STS, RMS-STV, RMS-CURRENT-STS, and RMS-CURRENT-STV Values

```
DATA DIVISION.
FILE SECTION.
FD  FILE-1.
01  RECORD-1         PIC X(50).
FD  FILE-2.
01  RECORD-2         PIC X(50).
WORKING-STORAGE SECTION.
01  ANSWER           PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    .
    .
    WRITE RECORD-1 INVALID KEY PERFORM A901-REPORT-FILE1-STATUS.
*
*   The following PERFORM statement displays the file status values
*   resulting from the above WRITE statement for FILE-1.
*
    PERFORM A903-REPORT-RMS-CURRENT-STATUS.
    .
    .
    .
    WRITE RECORD-2 INVALID KEY PERFORM A902-REPORT-FILE2-STATUS.
*
*   The following PERFORM statement displays the file status values
*   resulting from the above WRITE statement for FILE-2.
*
    PERFORM A903-REPORT-RMS-CURRENT-STATUS.
    .
    .
    .
A901-REPORT-FILE1-STATUS.
*******************************************
*
    DISPLAY "RMS-STS = " RMS-STS OF FILE-1 WITH CONVERSION.
    DISPLAY "RMS-STV = " RMS-STV OF FILE-1 WITH CONVERSION.
*
*******************************************
    PERFORM A999-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.
```

(continued on next page)

Example 12–5 (Cont.): Referencing RMS-STS, RMS-STV, RMS-CURRENT-STS, and RMS-CURRENT-STV Values

```
A902-REPORT-FILE2-STATUS.
*******************************************
*
    DISPLAY "RMS-STS = " RMS-STS OF FILE-2 WITH CONVERSION.
    DISPLAY "RMS-STV = " RMS-STV OF FILE-2 WITH CONVERSION.
*
*******************************************
    PERFORM A999-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.
A903-REPORT-CURRENT-STATUS.
*******************************************
*
    DISPLAY "RMS-CURRENT-STS = " RMS-CURRENT-STS WITH CONVERSION.
    DISPLAY "RMS-CURRENT-STV = " RMS-CURRENT-STV WITH CONVERSION.
*
*******************************************
    PERFORM A999-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.
A999-GET-ANSWER.
    DISPLAY "Do you want to continue?"
    DISPLAY "Please answer Y or N"
    ACCEPT ANSWER.
```

12.4 Using Declarative Procedures to Handle Exception Conditions

A Declarative procedure executes whenever an I/O statement results in an exception condition (a File Status value that does not begin with a zero (0)) and the I/O statement does not contain an AT END or INVALID KEY phrase. The AT END and INVALID KEY phrases take precedence over a Declarative procedure, but only for the I/O statement that includes the clause. Therefore you can have specific I/O statement exception condition handling for a file and also include a Declarative procedure for general exception handling.

A Declarative procedure is a set of one or more special-purpose sections at the beginning of the Procedure Division. As shown in Example 12–6, the key word DECLARATIVES precedes the first of these sections, and the key words END DECLARATIVES follow the last.

Example 12–6: The Declarative Skeleton

```
PROCEDURE DIVISION.
DECLARATIVES.
     .
     .
     .
END DECLARATIVES.
```

As shown in Example 12–7, a Declarative procedure consists of a section header, followed, in order, by a USE statement and zero, one, or more paragraphs.

Example 12–7: A Declarative Procedure Skeleton

```
PROCEDURE DIVISION.
DECLARATIVES.
D0-00-FILE-A-PROBLEM SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON FILE-A.
D0-01-FILE-A-PROBLEM.
     .
     .
     .
D0-02-FILE-A-PROBLEM.
     .
     .
     .
D0-03-FILE-A-PROBLEM.
     .
     .
     .
END DECLARATIVES.
```

VAX COBOL Declarative procedures, and the conditions in the USE statement under which they execute, are as follows:

- File name—You can define a file name Declarative procedure for each file name. This procedure overrides the next four procedures. It executes for any unsuccessful exception condition.
- INPUT—You can define only one INPUT Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for input and (2) a file name Declarative does not exist for that file.
- OUTPUT—You can define only one OUTPUT Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for output and (2) a file name Declarative does not exist for that file.
- INPUT-OUTPUT—You can define only one INPUT-OUTPUT Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for input/output and (2) a file name Declarative does not exist for that file.
- EXTEND—You can define only one EXTEND Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for extending and (2) a file name Declarative does not exist for that file.

Note that the USE statement itself does not execute; it defines the condition that causes the Declarative procedure to execute. For more information about Declarative procedures, refer to the USE statement in the *VAX COBOL Reference Manual.*

Example 12–8 shows you how to include each of the conditions in your program and contains explanatory comments for each.

Example 12–8: Five Types of Declarative Procedures

```
PROCEDURE DIVISION.
DECLARATIVES.
********************************************************
D1-00-FILE-A-PROBLEM SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON FILE-A.
*
*
* If any I/O statement for FILE-A results in an
* error, D1-00-FILE-A-PROBLEM executes.
*
*
D1-01-FILE-A-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
********************************************************
D2-00-FILE-INPUT-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON INPUT.
*
*
* If an error occurs because of an I/O statement
* for any file open in the input mode except FILE-A,
* D2-00-FILE-INPUT-PROBLEM executes.
*
*
D2-01-FILE-INPUT-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
********************************************************
D3-00-FILE-OUTPUT-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON OUTPUT.
*
*
* If an error occurs because of an I/O statement
* for any file open in the output mode except FILE-A,
* D3-00-FILE-OUTPUT-PROBLEM executes.
*
*
D3-01-FILE-OUTPUT-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
```

(continued on next page)

Example 12–8 (Cont.): Five Types of Declarative Procedures

```
********************************************************
D4-00-FILE-I-O-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON I-O.
*
*
* If an error occurs because of an I/O statement
* for any file open in the input-output mode except FILE-A,
* D4-00-FILE-I-O-PROBLEM executes.
*
*
*
D4-01-FILE-I-O-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
********************************************************
D5-00-FILE-EXTEND-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON EXTEND.
*
*
* If an error occurs because of an I/O statement
* for any file open in the extend mode except FILE-A,
* D5-00-FILE-EXTEND-PROBLEM executes.
*
*
D5-01-FILE-EXTEND-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
********************************************************
D9-00-REPORT-FILE-STATUS SECTION.
    .
    .
    .
END DECLARATIVES
********************************************************
A000-BEGIN SECTION.
    .
    .
    .
```

Chapter 13

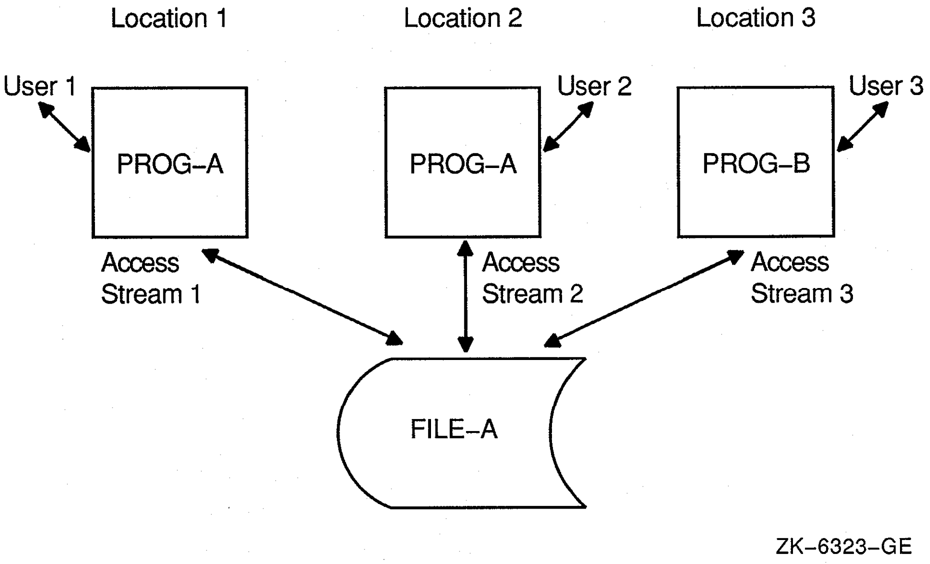
Sharing Files and Protecting Records

This chapter discusses file sharing and record locking for sequential, relative, and indexed files.

13.1 File-Sharing and Record-Locking Concepts

In a data manipulation environment where many users and programs access the same data, file control must be applied to protect files from nonprivileged users, to permit the desired degree of file sharing, and to preserve data integrity in the files. For example, in Figure 13–1 many users and programs want to access data found in FILE-A.

Figure 13–1: Multiple Access to a File



File sharing and record locking allow you to control file and record operations when more than one access stream (the series of file and record operations being performed by a single user) is concurrently accessing a file, as in Figure 13–1.

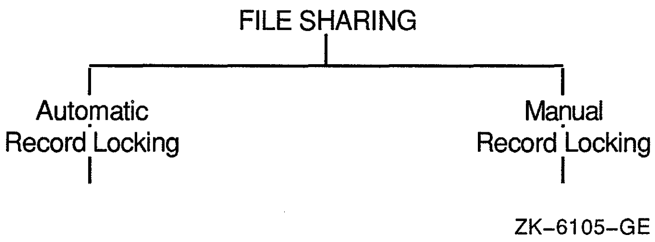
A VAX COBOL program can define one or more RMS access streams. You create one access stream with each OPEN file-name statement. The access stream remains active until you terminate the access stream with the CLOSE file-name statement, or your program terminates.

File sharing allows multiple readers and writers to access a single file concurrently. The protection level of the file, set by the file owner, determines which users can share a file.

Record locking controls simultaneous record operations in files that are accessed concurrently. Record locking ensures that when a program is writing, deleting, or rewriting a record in a given access stream, another access stream is allowed to access the same record in a specified manner.

Figure 13–2 illustrates the relationship of record locking to file sharing.

Figure 13–2: Relationship of Record Locking to File Sharing



File sharing is a function of the file system, while record locking is a function of the VAX Record Management Services (RMS). The file system manages file placement and the file-sharing process, in which multiple access streams simultaneously access a file. RMS manages the record-sharing process and provides access methods to records within a file. This includes managing the record-locking process, in which multiple access streams simultaneously access a record.

You must have successful file sharing before you can consider record locking.

In VAX COBOL, the file operations begin with an OPEN statement and end with a CLOSE statement. The OPEN statement initializes an access stream. The CLOSE statement terminates an access stream and can be either explicit (stated in the program) or implicit (on program termination).

In VAX COBOL, you use the ALLOWING clause (in the OPEN statement and certain record operation statements) to specify file sharing and record locking. This clause describes what operations other access streams can perform on specified files. You use the VAX COBOL open mode specification to provide the specification for the intentions of your access stream.

NOTE

> The first program to open a file determines how other programs can access the file concurrently (if at all).

The record operations for VAX COBOL are as follows:

- READ
- START

- WRITE
- REWRITE
- DELETE
- UNLOCK

You must specify the APPLY clause in the I-O-CONTROL paragraph when you use manual record locking. See Section 13.3.2 for more details on the use of this clause.

13.2 Ensuring Successful File Sharing

Successful file sharing requires that you:

- Provide disk residency for the file.
- Use the VMS system file protection facility, as related to the user identification code (UIC).
- Determine the intended access mode to the file (VAX COBOL open modes).
- Indicate the access allowed by other streams (VAX COBOL ALLOWING clause).

The remainder of this section discusses these four requirements.

13.2.1 Providing Disk Residency

Only files that reside on a disk can be shared. In VAX COBOL you can share sequential, relative, and indexed files.

13.2.2 Using VMS File Protection

By using the appropriate VMS file protection, the owner of a file determines how other users can access the file. An owner can permit up to four types of file access for each of four user categories. The level of file protection the file owner specifies determines the types of open modes that a VAX COBOL program can specify successfully. The four types of file access follow. Note that the following VMS file protection access types are not a part of VAX COBOL syntax:

- READ—Permits the reading of the records in the file.
- WRITE—Permits updating or extending the records in the file.
- EXECUTE—Applies to on-disk volume protection and image execution and is therefore not applicable to a VAX COBOL program.
- DELETE—Permits deletion of the file and is therefore not applicable to a VAX COBOL program (since VAX COBOL has no delete file facility).

NOTE

> Note that the EXECUTE and DELETE categories of the file protection are used by VAX COBOL programmers but not by VAX COBOL programs; however, a VAX COBOL program can perform these actions using system service routines.

In the VMS file protection facility, four different categories of users exist with respect to data structures and devices. A file owner determines which of the following user categories can share the file:

- SYSTEM—Users of the system whose group numbers are in the range 0 to the value of the MAXSYSGROUP parameter, or who have certain I/O-related privileges
- OWNER—Users of the system whose UIC group and member numbers are identical to the UIC of the file owner
- GROUP—Users of the system whose group number is identical to the group number of the file owner
- WORLD—All other users of the system who are not included in the previous categories

The owner of the file has a default protection that the system applies to each newly created file unless the owner specifically requests modified protection.

For more information on file protection, refer to the VMS documentation on DCL.

13.2.3 Determining the Intended Access Mode to a File

Once you establish disk residency and privileges for a file, you can consider the third file-sharing criterion: how the stream intends to access the file. You specify this intention by using the VAX COBOL open and access modes.

The VAX COBOL open modes are INPUT, OUTPUT, EXTEND, and I-O. The VAX COBOL access modes are SEQUENTIAL, RANDOM, and DYNAMIC. The combination of open and access modes determines the operations intended on the file.

You must validate your VAX COBOL intention against the file protection assigned by the file owner. For example, to use an OPEN INPUT clause requires that the file owner has granted read access privileges to the file. To use an OPEN OUTPUT or EXTEND clause requires write access privileges to the file. To use an OPEN I-O clause requires both read and write access privileges.

An OPEN OUTPUT clause creates a new version of the file, which makes it difficult to share the file.

The following chart shows the relationship between open and access modes and intended VAX COBOL operations. The word ANY indicates that all three access methods result in the same intentions.

Open Mode	Access Mode	Intended COBOL Operations
INPUT	ANY	READ, START
OUTPUT	ANY	WRITE
I-O	SEQUENTIAL	READ, START, REWRITE, DELETE
	RANDOM/DYNAMIC	READ, START, REWRITE, DELETE, WRITE
EXTEND	SEQUENTIAL	WRITE

Note that if the file protection does not permit the intended operations, file access is not granted, even if open and access modes are compatible.

File protection and open mode access apply to both the unshared and shared (multiple access stream) file environments. A file protection and intent check is made when the first access stream opens a file (in the unshared file environment), and again when the second and subsequent access streams open the file (in the shared file environment).

After these file-sharing checks pass, you can apply the fourth file-sharing criterion, access allowed to other streams.

13.2.4 Indicating the Access Allowed to Other Streams

You use the VAX COBOL ALLOWING clause of the OPEN statement to specify what other access streams are allowed to access that file.

The OPEN ALLOWING options are as follows:

- OPEN ALLOWING NO OTHERS—Locks the file for exclusive access. Attempts by other access streams to access the file cause a file lock exception.
- OPEN ALLOWING READERS—Locks the file against operations that indicate intended write access (OPEN I-O and OPEN EXTEND). Other streams can use the OPEN INPUT statement to access the file.
- OPEN ALLOWING WRITERS or UPDATERS or ALL—Allows read and write access by other streams. Other access streams can open the file in INPUT, EXTEND, and I-O modes.

VAX COBOL also permits a list of OPEN ALLOWING options, separated by commas. The list results in the following equivalent ALLOWING specifications:

- ALLOWING WRITERS, UPDATERS becomes ALLOWING ALL
- ALLOWING READERS, UPDATERS becomes ALLOWING UPDATERS

The first access stream uses the ALLOWING clause to specify what other access streams can do. When the second and subsequent access streams attempt to open the file, the following checks occur:

1. The allowed options of this access stream are checked against the intended access of the previous streams.
2. The intended access of this access stream is checked against the allowed access of the previous streams.

For example, if the first access stream specifies the ALLOWING READERS clause, then a subsequent access stream that opens the file ALLOWING NO OTHERS would fail. Also, if the first access stream opens the file ALLOWING READERS, the following access stream that opens the file ALLOWING ALL and with I-O mode would fail, because the clause option and the I-O mode declare write intent to the file.

13.2.5 Describing Types of Access Streams

You can establish several types of access streams. For example, two programs opening the same file represent two access streams to that file. Both programs begin with the file open, perform record operations, and then close the file.

In addition, a single program can establish multiple access streams to a file. In this case, you use multiple SELECT clauses to choose the file, while the FDs and all other clauses and statements treat the file independently. Example 13–1 shows two access streams to the same file.

Example 13–1: Two Access Streams to a Single File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ACCESSTRM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILE-1
     ORGANIZATION IS SEQUENTIAL
     ASSIGN TO "SHAREDAT.DAT"
     .
     .
     .
SELECT FILE-2
     ORGANIZATION IS SEQUENTIAL
     ASSIGN TO "SHAREDAT.DAT"
     .
     .
     .
I-O-CONTROL.
     APPLY LOCK-HOLDING ON FILE-1, FILE-2.
DATA DIVISION.
FILE SECTION.
FD FILE-1
     .
     .
     .
FD FILE-2
     .
     .
     .
PROCEDURE DIVISION.
01.
     OPEN INPUT FILE-1 ALLOWING READERS.
     OPEN INPUT FILE-2 ALLOWING READERS.
     READ FILE-1 ALLOWING READERS.
     READ FILE-2 ALLOWING NO OTHERS.
          .
          .
          .
     UNLOCK FILE-1 ALL RECORDS.
     UNLOCK FILE-2 ALL RECORDS.
     CLOSE FILE-1.
     CLOSE FILE-2.
     STOP RUN.
```

13.2.6 Summarizing Related File-Sharing Criteria

This section summarizes the relationships among three of the file-sharing criteria (the first file-sharing requirement, disk residency, is not included).

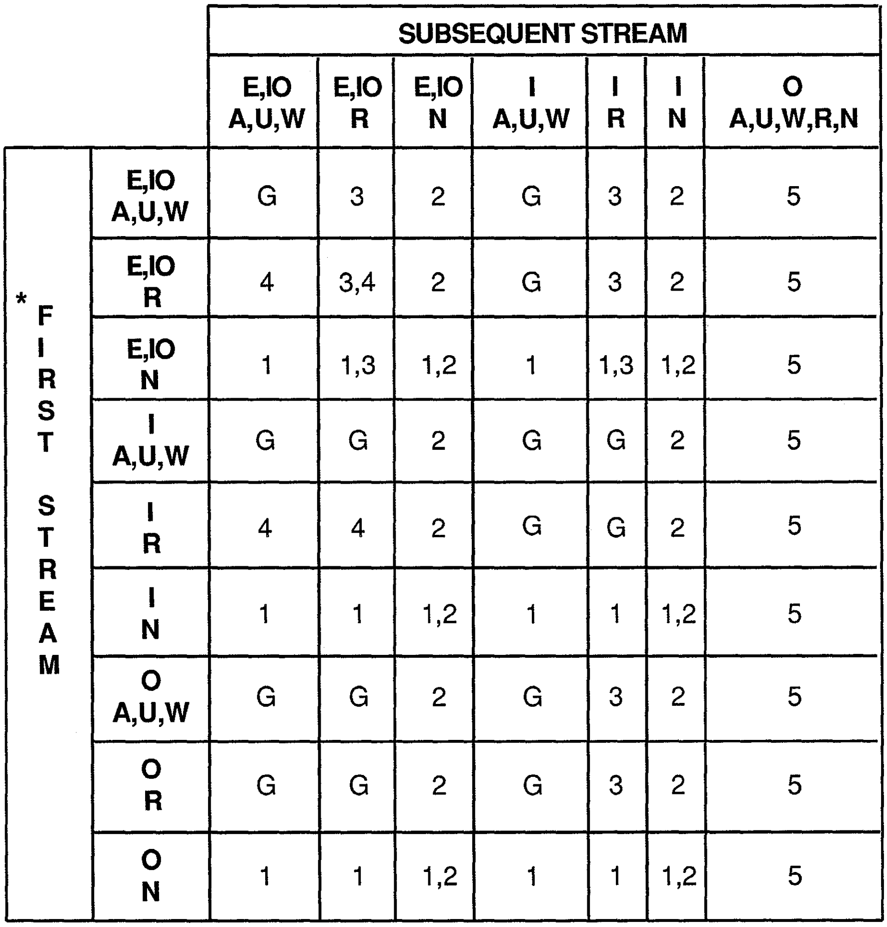
The following chart shows the file protection and open mode requirements. For example, the file protection privilege READ (R) permits OPEN INPUT.

File Protection	Open Mode
R	INPUT
W	OUTPUT, EXTEND
RW	I-O, INPUT, OUTPUT, EXTEND

Remember, you specify intended operations through the first access stream. For the second and subsequent shared access to a file, you use the access intentions (open modes) and the ALLOWING clause to determine if and how a file is shared.

Figure 13–3 shows the valid and invalid OPEN ALLOWING combinations between first and subsequent access streams.

Figure 13–3: File-Sharing Options



		SUBSEQUENT STREAM						
		E,IO A,U,W	E,IO R	E,IO N	I A,U,W	I R	I N	O A,U,W,R,N
* FIRST STREAM	E,IO A,U,W	G	3	2	G	3	2	5
	E,IO R	4	3,4	2	G	3	2	5
	E,IO N	1	1,3	1,2	1	1,3	1,2	5
	I A,U,W	G	G	2	G	G	2	5
	I R	4	4	2	G	G	2	5
	I N	1	1	1,2	1	1	1,2	5
	O A,U,W	G	G	2	G	3	2	5
	O R	G	G	2	G	3	2	5
	O N	1	1	1,2	1	1	1,2	5

Legend:

* Assumes "no" file protection violations on first stream

G Second stream successfully opens and file sharing is granted

1 Second stream denied access to the file because the first stream requires exclusive access (first specified NO OTHERS)

2 Second stream denied access to the file because the second stream requires exclusive access (second specified NO OTHERS)

3 Second stream denied access to the file because first intends write while second specifies read–only sharing

4 Second stream denied access to the file because second intends write while first specifies read–only sharing

5 No sharing; second will create new file with OPEN OUTPUT

ZK–6059–GE

The abbreviations used in Figure 13–3 are as follows:

- OPEN ABBREVIATIONS
  - E,IO—OPEN EXTEND, OPEN I-O
  - I—OPEN INPUT
  - O—OPEN OUTPUT
- ALLOWING ABBREVIATIONS
  - A,U,W—OPEN ALLOWING ALL or OPEN ALLOWING UPDATERS or OPEN ALLOWING WRITERS
  - R—OPEN ALLOWING READERS
  - N—OPEN ALLOWING NO OTHERS

In the following example, three streams illustrate some of the file-sharing rules:

```
STREAM 1           OPEN INPUT ALLOWING ALL
STREAM 2           OPEN INPUT ALLOWING READERS
STREAM 3           OPEN I-O ALLOWING UPDATERS
```

In this example, stream 1 permits ALLOWING ALL; thus stream 2 can read the file. However, the third stream violates the intent of the second stream, because OPEN I-O implies a write intention that stream 2 disallows. Consequently, the third access stream receives a file locked error.

13.2.7 Checking File Operations

You can check the success or failure of a file open operation by using the File Status code or the RMS status variable (a VAX COBOL special register). This VAX COBOL special register normally contains RMS-STS values, which you can obtain by using the VALUE IS EXTERNAL clause.

In addition, if no RMS translation exists for the VAX COBOL specific error condition, the RMS-STS special register may contain an error message value. See the *VAX COBOL Reference Manual* for an explanation of the VAX COBOL error message symbols.

Table 13–1 illustrates the codes you frequently use in a file-sharing environment.

Table 13–1: File-Sharing Environment Codes

File Status	RMS-STS Register	Meaning
00	RMS$_SUC	Successful operation
91	RMS$_FLK	File is locked
38	COB$_FILCLOLOC	File is closed WITH LOCK
30	RMS$_PRV	File protection violation

File Status 00, which corresponds to the RMS-STS symbol RMS$_SUC, results from completion of a successful operation.

File Status 91, which corresponds to the RMS-STS symbol RMS$_FLK, indicates that another accessor of the file has denied access. Other accessors are other programs that have denied file access by opening the file for exclusive access (OPEN ALLOWING NO OTHERS).

File Status 38, which corresponds to the symbol COB$_FILCLOLOC in the RMS-STS special register, indicates that another file accessor has denied access by executing a CLOSE WITH LOCK statement.

File Status 30, when it corresponds to the RMS-STS symbol RMS$_PRV, results from a violation of the file protection codes described in Section 13.2.2. To correct this condition, the file owner must reset the protection on the file or the directory that contains the file.

Example 13–2 includes additional codes you may encounter.

Example 13–2: Program Segment for RMS-STS File-Sharing Exceptions

```
WORKING-STORAGE SECTION.

01 RMS-SUC        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_SUC.
01 RMS-OK-RLK     PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_OK_RLK.
01 RMS-OK-RRL     PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_OK_RRL.
01 RMS-RNL        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_RNL.
01 RMS-DNR        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_DNR.
01 RMS-EOF        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_EOF.
01 RMS-FLK        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_FLK.
01 RMS-FNF        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_FNF.
01 RMS-PRV        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_PRV.
01 RMS-REX        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_REX.
01 RMS-RLK        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_RLK.
01 RMS-RNF        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_RNF.
01 RMS-WLK        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_WLK.
01 RMS-DNF        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_DNF.
01 RMS-DIR        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_DIR.
01 RMS-DUP        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_DUP.
01 RMS-FUL        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_FUL.
01 RMS-KEY        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_KEY.
01 RMS-KRF        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_KRF.
01 RMS-KSZ        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_KSZ.
01 RMS-RAC        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_RAC.
01 RMS-RSZ        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_RSZ.
01 RMS-SNE        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_SNE.
01 RMS-SPE        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_SPE.
01 RMS-ENQ        PIC S9(9) COMP  VALUE IS EXTERNAL RMS$_ENQ.
PROCEDURE DIVISION.
DECLARATIVES.
FILE-1-ERR SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON FILE-1.
FILE-1-USE.
    EVALUATE RMS-STS OF FILE-1
      WHEN RMS-SUC      DISPLAY "successful operation"
      WHEN RMS-OK-RLK   DISPLAY "record locked but read anyway"
      WHEN RMS-OK-RRL   DISPLAY "record locked against read but read anyway"
      WHEN RMS-RNL      DISPLAY "record not locked"
      WHEN RMS-DNR      DISPLAY "device not ready or not mounted"
      WHEN RMS-EOF      DISPLAY "end of file detected"
      WHEN RMS-FLK      DISPLAY "file currently locked by another user"
      WHEN RMS-FNF      DISPLAY "file not found"
      WHEN RMS-PRV      DISPLAY "file protection violation"
      WHEN RMS-REX      DISPLAY "record already exists"
      WHEN RMS-RLK      DISPLAY "record currently locked by another stream"
```

(continued on next page)

Example 13–2 (Cont.): Program Segment for RMS-STS File-Sharing Exceptions

```
      WHEN RMS-RNF      DISPLAY "record not found"
      WHEN RMS-WLK      DISPLAY "device currently write locked"
      WHEN RMS-DNF      DISPLAY "directory not found"
      WHEN RMS-DIR      DISPLAY "error in directory name"
      WHEN RMS-DUP      DISPLAY "duplicate key detected (DUP not set)"
      WHEN RMS-FUL      DISPLAY "device full (insufficient space)"
      WHEN RMS-KEY      DISPLAY "invalid record number key or key value"
      WHEN RMS-KRF      DISPLAY "invalid key-of-reference for $GET/$FIND"
      WHEN RMS-KSZ      DISPLAY "invalid key size for $GET/$FIND"
      WHEN RMS-RAC      DISPLAY "invalid record access mode"
      WHEN RMS-RSZ      DISPLAY "invalid record size"
      WHEN RMS-SNE      DISPLAY "file sharing not enabled"
      WHEN RMS-SPE      DISPLAY "file-sharing page count exceeded"
      WHEN RMS-ENQ      DISPLAY "system service request failed"
      WHEN OTHER STOP RUN
    END-EVALUATE.
END DECLARATIVES.
```

13.2.8 Specifying the OPEN EXTEND in a File-Sharing Environment

If you specify an OPEN EXTEND in a file-sharing environment, be aware that the EXTEND results differ depending upon what file organization you use.

13.2.8.1 OPEN EXTEND with a Shared Sequential File

In a shared sequential file environment, when two concurrent access streams use EXTEND ALLOWING UPDATERS, ALLOWING ALL, or ALLOWING WRITERS, and both streams issue a write to the end of the file (EOF), the additional data will come from both streams, and the data will be inserted into the file in the order it was written to the file.

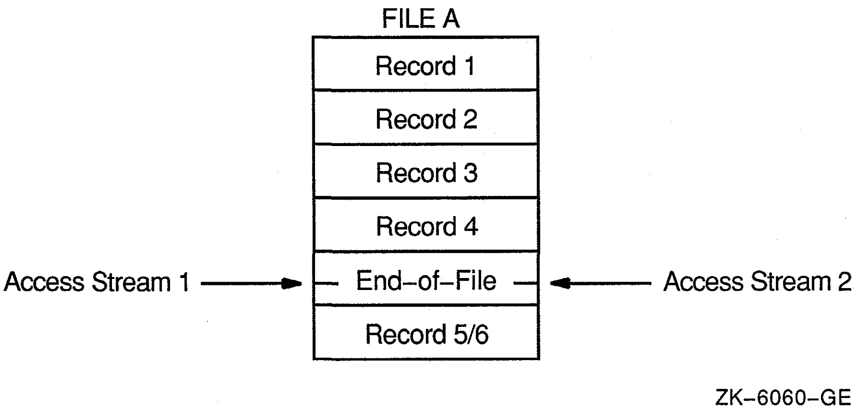
13.2.8.2 OPEN EXTEND with a Shared Relative File

You must use the sequential file access mode when you open a relative file in extend mode. Sequential file access mode for a relative file indicates that the record order is by ascending relative record number.

In sequential access mode for a relative file, the RELATIVE KEY clause of the WRITE statement is not used on record insertion; instead, the RELATIVE KEY clause acts as a receiving field. Consequently, after the completion of a write by the first access stream, the relative key field is set to the actual relative record number.

Figure 13–4 illustrates why this condition occurs.

Figure 13–4: Why a Record-Already-Exists Error Occurs



As the file operations begin, both access streams point to the end of file by setting record 4 as the highest relative record number in the file. When access stream 1 writes to the file, record 5 is created as the next ascending relative record number and 5 is returned as the RELATIVE KEY number.

When access stream 2 writes to the file, it also tries to write the fifth record. Record 5 already exists (inserted by the first stream), and the second access stream gets a record-exists error. Thus, in a file-sharing environment, the second access stream always receives a record-exists error.

13.2.8.3 OPEN EXTEND with a Shared Indexed File

You must use the sequential file access mode when you open an indexed file in extend mode. Sequential access mode requires that the first additional record insertion have a prime record key whose value is greater than the highest prime record key value in the file.

In a file-sharing environment, you should be aware of and prepared for duplicate key errors (by using INVALID KEY and USE procedures), especially on the first write to the file by the second access stream.

Subsequent writes should also allow for duplicate key errors, although subsequent writes are not constrained to use keys whose values are greater than the highest key value that existed at file open time. If you avoid duplicate key errors, you successfully insert all access stream records.

13.3 Using Record Locking

Once you meet all file-sharing criteria and you access a file, you can consider two record-locking modes that control access to records in a file:

- Automatic record locking
- Manual record locking

Automatic record locking is the default. In automatic record locking, if you do not specify an ALLOWING clause on the OPEN statement, the default for files opened for INPUT is ALLOWING READERS, and the default for files opened for I-O, OUTPUT, or EXTEND mode is ALLOWING NO OTHERS.

You specify manual record locking by using the APPLY LOCK-HOLDING clause (in the I-O-CONTROL paragraph), the OPEN ALLOWING statement, and the ALLOWING clauses on the VAX COBOL record operations (except DELETE).

Both automatic record locking and manual record locking use the same form of the OPEN ALLOWING clause.

When you close a file, any existing record lock is released automatically. The UNLOCK RECORD statement releases the lock only on the current record, which is the last record you successfully accessed.

13.3.1 Specifying Automatic Record Locking

Automatic record locking applies the lock when you access the record and releases the lock when you de-access the record. In automatic record locking the access stream can have only one record locked at a time and can apply only one type of lock to the records of the file.

You de-access a record by using the next READ operation, a REWRITE or a DELETE operation on the record, or by closing the file. In addition, you can release locks applied by automatic record locking by using the UNLOCK statement.

In automatic record-locking mode, you can release the current record lock by using either an UNLOCK RECORD statement or an UNLOCK ALL RECORDS statement. However, because in automatic record locking you can only lock one record at a time, the UNLOCK ALL RECORDS statement unnecessarily checks all records for additional locks.

The sample program in Example 13–3 uses automatic record locking. The program opens the file with I-O ALLOWING READERS. Another access stream in another program opens the file with INPUT ALLOWING ALL.

If the first access stream is updating records in random order, a record lock can occur to the second stream from the READ until the REWRITE statement of the first stream. Record locks can also occur to the first stream when the second stream reads a record and the first stream tries to read the same record.

Example 13–3: Automatic Record Locking

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AUTOLOCK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILE-1
     ORGANIZATION IS RELATIVE
     ASSIGN TO "SHAREDAT.DAT"
     .
     .
     .
I-O-CONTROL.
```

(continued on next page)

Example 13–3 (Cont.): Automatic Record Locking

```
DATA DIVISION.
FILE SECTION.
FD FILE-1
     RECORD CONTAINS 100 CHARACTERS
     .
     .
     .
PROCEDURE DIVISION.
     OPEN I-O FILE-1 ALLOWING READERS.
     READ FILE-1-REC.
          .
          .
          .
     REWRITE FILE-1.
     CLOSE FILE-1.
     STOP RUN.
```

13.3.2 Specifying Manual Record Locking

Manual record locking allows greater control of locking options by permitting users to lock multiple records in a file and by permitting different types of locking to apply to different records.

Manual record locking applies the specified lock when you access the record and releases the lock when you unlock the record.

When you specify manual record locking you must use the following clauses: (1) an APPLY LOCK-HOLDING clause in the I-O CONTROL paragraph, (2) an OPEN ALLOWING clause at file open time, and (3) an ALLOWING clause on each VAX COBOL record operation (except DELETE).

The possible ALLOWING clauses for the VAX COBOL record operations are as follows:

- ALLOWING NO OTHERS—Locks records for exclusive access. Others cannot perform READ, WRITE, DELETE, or UPDATE statements. This clause constitutes a lock for write and does not allow readers.

  However, if the file's OPEN mode is INPUT, using this clause on the record operation does not lock the record for exclusive access. The most restrictive record locking you can achieve on a file whose OPEN mode is INPUT is to exclude writers and allow readers. If a file's OPEN mode is INPUT, specifying ALLOWING NO OTHERS is equivalent to specifying ALLOWING READERS.

- ALLOWING READERS—Locks records against WRITE, REWRITE, and DELETE access by all streams including the stream that issues the statement. Others can perform READ statements. This clause constitutes an RMS lock for read, which allows others to read the record, but not to write it.

- ALLOWING UPDATERS—Does not apply any locks to the records. Others can perform READ, REWRITE, and DELETE statements. This clause constitutes a no record lock condition.

Figure 13–5 shows the valid and invalid ALLOWING combinations for manual record locking. The columns represent lock held, and the rows represent lock requested.

Figure 13–5: Valid and Invalid Combinations for Manual Record Locking

		"lock held"		
	ALLOWING	UPDATERS	READERS	NO OTHERS
"lock requested"	UPDATERS	legal	legal	illegal
	READERS	legal	legal	illegal
	NO OTHERS	illegal	illegal	illegal

ZK-6061-GE

Example 13–4 uses manual record locking. The file is opened with the ALLOWING READERS clause. The records are read but do not become available to other access streams because of the lock applied by the READ statement (READ...ALLOWING NO OTHERS). When the UNLOCK is executed, the records can be read by another access stream if that stream opens the file allowing writers.

Example 13–4: Sample Program Using Manual Record Locking

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MANLOCK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILE-1
     ORGANIZATION IS RELATIVE
     ASSIGN "SHAREDAT.DAT"
      .
      .
      .
I-O-CONTROL.
     APPLY LOCK-HOLDING ON FILE-1.
DATA DIVISION.
FILE SECTION.
FD FILE-1
     RECORD CONTAINS 100 CHARACTERS
      .
      .
      .
```

(continued on next page)

Example 13–4 (Cont.): Sample Program Using Manual Record Locking

```
PROCEDURE DIVISION.
01.
     OPEN I-O FILE-1 ALLOWING READERS.
          .
          .
          .
     READ FILE-1 ALLOWING NO OTHERS.
          .
          .
          .
     REWRITE FILE-1-REC ALLOWING NO OTHERS.
          .
          .
          .
     UNLOCK FILE-1 ALL RECORDS.
     CLOSE FILE-1.
     STOP RUN.
```

In manual record locking, you release record locks by the UNLOCK statement or when you close the file (either explicitly or implicitly). The UNLOCK statement provides for either releasing the lock on the current record (UNLOCK RECORD) or releasing all locks currently held by the access stream on the file (UNLOCK ALL RECORDS).

When you access a shared file with ACCESS MODE IS SEQUENTIAL and use manual record locking, the UNLOCK statement can cause you to violate either of the following statements: (1) the REWRITE statement rule that states that the last input-output statement executed before the REWRITE must be a READ or START statement, or (2) the DELETE statement rule that states that the last input/output statement executed before the DELETE statement must be a READ. You must lock the record before it can be rewritten or deleted.

13.3.3 Locking Error Conditions

Two record-locking conditions (hard and soft record lock) indicate if a record was transferred to the record buffer. VAX COBOL provides the success, failure, or informational status of an I/O operation in the File Status variable.

A hard record lock causes the File Status variable to be set to 92, whereas a soft record lock causes the File Status variable to be set to 90.

13.3.3.1 Hard Record Locks

A hard record lock condition indicates that the record operation failed and the record was not transferred to the buffer. A hard record lock results from a situation such as the following, which uses manual record-locking mode:

1. Program A opens the file I-O ALLOWING ALL.
2. Program A reads a record ALLOWING NO OTHERS.
3. Program B opens the file I-O ALLOWING ALL.
4. Program B tries to access the same record as A.
5. Program B receives a hard record lock condition.
6. The record is NOT accessible to Program B.

7. Program B's File Status variable is set to 92.
8. Program B's USE procedure is invoked.
9. Program A continues.

The record was not available to program B.

13.3.3.2 Soft Record Locks

A soft record lock condition indicates that the record is locked, but access to the record is still allowed. A soft record lock occurs when the stream accessing the record has allowed read access by other streams that have opened the file in input mode, or when a READ REGARDLESS or START REGARDLESS statement (see Section 13.3.5) is employed to override a record lock. A soft record lock results from a situation such as the following, which uses automatic record-locking mode:

1. Program A opens the file I-O ALLOWING READERS.
2. Program A reads a record.
3. Program B opens the file INPUT ALLOWING ALL.
4. Program B reads the same record.
5. Program B receives a soft record lock condition. The record is accessible to Program B.
6. Program B's File Status variable is set to 90.
7. Program B's USE procedure is invoked.
8. Programs A and B continue.

The record was available to Program B.

13.3.4 Releasing Locks on Deleted Records

In automatic record locking, the DELETE operation releases the lock on the record. In manual record-locking mode, you can delete a record using the DELETE statement but still retain a record lock. You must use the UNLOCK ALL RECORDS statement to release the lock on a deleted record.

If a second stream attempts to access a deleted record that retains a lock, the second stream will receive either a record not found exception or a hard lock condition.

If another stream attempts to REWRITE to a deleted record that retains a lock, the type of exception that access stream receives depends on its file organization. If the file organization is RELATIVE, the access stream receives the record locked status. If the file organization is INDEXED, the access stream succeeds (receives the success status).

In relative files, the lock is on the relative cell (record) and cannot be rewritten until the lock is released. On indexed files, the lock is on the records file address (RFA) of the deleted record, so a new record (with a new RFA) can be written to the file.

13.3.5 Bypassing a Record Lock

When you use manual record locking, you can apply a READ REGARDLESS or START REGARDLESS statement to bypass any record lock that exists. READ REGARDLESS reads the record and applies no locks to the record. START REGARDLESS positions to the record and applies no locks to the record. If the record is currently locked by another access stream, a soft record lock condition can be detected by a USE procedure.

You use READ REGARDLESS or START REGARDLESS when: (1) a record is locked against readers because the record is about to be written, but (2) your access program needs the existing record regardless of the possible change in its data.

NOTE

You should recognize that READ REGARDLESS and START REGARDLESS are very powerful tools and should be used only in extreme circumstances. You prevent the use of READ REGARDLESS or START REGARDLESS at the file protection level, where you prevent readers from referencing the file.

Chapter 14

Using the COBOL SORT and MERGE Statements

This chapter presents and explains examples of the SORT and MERGE statements.

The SORT statement provides a wide range of sorting capabilities and options. To establish a SORT routine, you do the following (1) declare the sort file with a SELECT statement in the Environment Division; (2) use a Sort Description (SD) entry in the Data Division to define the sort file's characteristics; and (3) use a SORT statement in the Procedure Division.

The following program segments demonstrate SORT program coding:

SELECT Statement

```
SELECT SORT-FILE ASSIGN TO "SRTFIL"
```

An SD File Description Entry

```
SD  SORT-FILE.
01  SORT-RECORD.
     05 SORT-KEY1    PIC X(5).
     05 SOME-DATA    PIC X(25).
     05 SORT-KEY2    PIC XX.
```

Note that you can place the sort file anywhere in the FILE SECTION, but you must use a Sort Description (SD) level indicator, not a File Description (FD) level indicator.

SORT Statement (in the Procedure Division)

```
SORT SORT-FILE
     ASCENDING KEY S-NAME
     USING NAME-FILE
     GIVING NEW-FILE.
```

This SORT statement names a sort file, a key, an input file, and an output file. An explanation of keys follows.

14.1 ASCENDING and DESCENDING KEY Phrases

Use the ASCENDING and DESCENDING KEY phrases to specify your sort parameters. The order of data names determines the sort hierarchy; that is, the major sort key is the first data name entered, while the minor sort key is the last data name entered.

In this example, the hierarchy of the sort is SORT-KEY-1, SORT-KEY-2, SORT-KEY-3.

```
SORT SORT-FILE
    ASCENDING KEY SORT-KEY-1 SORT-KEY-2
    DESCENDING KEY SORT-KEY-3
```

14.1.1 Sorting Concepts

Records are sorted based on the data values in the sort keys. The following example depicts unsorted employee name and address records used for creating mailing labels:

Smith,	Joe	234 Ash St.	New Boston	NH	04356
Jones,	Bill	12 Birch St.	Gardner	MA	01430
Baker,	Tom	78 Oak St.	Ayer	MA	01510
Thomas,	Pete	555 Maple St.	Maynard	MA	01234
Morris,	Dick	21 Harris St.	Acton	ME	05670

If you sort the addresses in the previous example using the zip code as the ascending sort key, the mailing labels are printed in the order shown in the following example:

					SORT KEY
Thomas,	Pete	555 Maple St.	Maynard	MA	01234
Jones,	Bill	12 Birch St.	Gardner	MA	01430
Baker,	Tom	78 Oak St.	Ayer	MA	01510
Smith,	Joe	234 Ash St.	New Boston	NH	04356
Morris,	Dick	21 Harris St.	Acton	ME	05670

Also, records can be sorted on more that one key at a time. If you need an alphabetical listing of all employees within each state, you can sort on the state code first (major sort key) and employee name second (minor sort key).

For example, if you sort the file in ascending order by state (major key) and last name (minor key), your name and address appear in the order shown in the following example:

SORT KEY				SORT KEY	
(minor)				(major)	
Baker,	Tom	78 Oak St.	Ayer	MA	01510
Jones,	Bill	12 Birch St.	Gardner	MA	01430
Thomas,	Pete	555 Maple St.	Maynard	MA	01234
Morris,	Dick	21 Harris St.	Acton	ME	05670
Smith,	Joe	234 Ash St.	New Boston	NH	04356

14.2 USING and GIVING Phrases

If you only need to resequence a file, use the USING and GIVING phrases of the SORT statement. The USING phrase opens the input file, then reads and releases its records to the sort. The GIVING phrase opens and writes sorted records to the output file.

Note that you cannot manipulate data with either the USING or the GIVING phrases.

Consider this SORT statement:

```
SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
     USING INPUT-FILE GIVING OUTPUT-FILE.
```

It does the following:

1. Opens INPUT-FILE
2. Reads all records in INPUT-FILE and releases them to the sort
3. Sorts the records in ascending sequence using the data in SORT-KEY-1
4. Opens the output file and writes the sorted records to OUTPUT-FILE
5. Closes all files used in the SORT statement

14.3 INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

You can manipulate data before and after sorting by using the INPUT PROCEDURE and OUTPUT PROCEDURE phrases, and sort only some of the information in a file. For example, these phrases allow you to use only those input records and/or input data fields you need.

The INPUT PROCEDURE phrase replaces the USING phrase when you want to manipulate data entering the sort. The SORT statement transfers control to the sections or paragraphs named in the INPUT PROCEDURE phrase. You then use COBOL statements to open and read files, and manipulate the data. You use the RELEASE statement to transfer records to the sort. After the last statement of the input procedure executes, control is given to the sort, and the records are subsequently sorted.

After the records are sorted, the SORT statement transfers control to the sections or paragraphs named in the OUTPUT PROCEDURE phrase. This phrase replaces the GIVING phrase when you want to manipulate data in the sort. You can use COBOL statements to open files and manipulate data. You use the RETURN statement to transfer records from the sort. For example, you can use the RETURN statement to print a report from sorted records.

Example 14–1 shows a sort using the INPUT and OUTPUT procedures.

Example 14–1: INPUT and OUTPUT PROCEDURE Phrases

```
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
                   ON DESCENDING KEY SORT-KEY-2
                   INPUT PROCEDURE IS 050-RETRIEVE-INPUT
                                 THRU 100-DONE-INPUT
                   OUTPUT PROCEDURE IS 200-WRITE-OUTPUT
                                  THRU 230-DONE-OUTPUT.
    DISPLAY "END OF SORT".
    STOP RUN.
```

(continued on next page)

Example 14–1 (Cont.): INPUT and OUTPUT PROCEDURE Phrases

```
050-RETRIEVE-INPUT SECTION.
060-OPEN-INPUT.
    OPEN INPUT IN-FILE.
070-READ-INPUT.
    READ IN-FILE AT END
        CLOSE IN-FILE
        GO TO 100-DONE-INPUT.
    MOVE INPUT-RECORD TO SORT-RECORD.
************************************************************
*You can add, change, or delete records before sorting     *
*using COBOL data manipulation techniques.                 *
************************************************************
    RELEASE SORT-RECORD.
    GO TO 070-READ-INPUT.
100-DONE-INPUT SECTION.
110-EXIT-INPUT.
    EXIT.
200-WRITE-OUTPUT SECTION.
210-OPEN-OUTPUT.
    OPEN OUTPUT OUT-FILE.
220-GET-SORTED-RECORDS.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 230-DONE-OUTPUT.
    MOVE SORT-RECORD TO OUTPUT-RECORD.
****************************************************
*You can add, change, or delete sorted records     *
*using COBOL data manipulation techniques.         *
****************************************************
    WRITE OUTPUT-RECORD.
    GO TO 220-GET-SORTED-RECORDS.
230-DONE-OUTPUT SECTION.
240-EXIT-OUTPUT.
    EXIT.
```

You can combine the INPUT PROCEDURE with the GIVING phrases, or the USING with the OUTPUT PROCEDURE phrases. In Example 14–2, the USING phrase replaces the INPUT PROCEDURE phrase used in Example 14–1.

Example 14–2: USING Phrase Replaces INPUT PROCEDURE Phrase

```
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
                   ON DESCENDING KEY SORT-KEY-2
                   USING IN-FILE
                   OUTPUT PROCEDURE IS 200-WRITE-OUTPUT
                                  THRU 230-DONE-OUTPUT.
    DISPLAY "END OF SORT".
    STOP RUN.
200-WRITE-OUTPUT SECTION.
210-OPEN-OUTPUT.
    OPEN OUTPUT OUT-FILE.
220-GET-SORTED-RECORDS.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 230-DONE-OUTPUT.
    MOVE SORTED-RECORD TO OUTPUT-RECORD.
    WRITE OUTPUT-RECORD.
    GO TO 220-GET-SORTED-RECORDS.
230-DONE-OUTPUT SECTION.
240-EXIT-OUTPUT.
    EXIT.
```

NOTE

You cannot access records released to the sort-file after the SORT statement ends.

14.4 WITH DUPLICATES IN ORDER Phrase

The sort orders data in the sequence specified in the ASCENDING KEY and DESCENDING KEY phrases. However, records with duplicate sort keys may not be written to the output file in the same sequence as they were read into it. The WITH DUPLICATES IN ORDER phrase ensures that any records with duplicate sort keys are in the same order in the output file as in the input file.

The following list shows the difference between sorting with the WITH DUPLICATES IN ORDER phrase and sorting without it:

		Sorted Without		**Sorted with**	
Input file		**Duplicates in Order**		**Duplicates in Order**	
Record		**Record**		**Record**	
Name	**Data**	**Name**	**Data**	**Name**	**Data**
JONES	ABCD	DAVIS	LMNO	DAVIS	LMNO
DAVIS	LMNO	JONES	EFGH	JONES	ABCD
WHITE	STUV	JONES	ABCD	JONES	EFGH
JONES	EFGH	SMITH	1234	SMITH	1234
SMITH	1234	WHITE	STUV	WHITE	STUV
WHITE	WXYZ	WHITE	WXYZ	WHITE	WXYZ

If you omit the WITH DUPLICATES IN ORDER phrase, you cannot predict the order of records with duplicate sort keys. The JONES records are not in the same sequence as they were in the input file, but the WHITE records are.

In contrast, the WITH DUPLICATES IN ORDER phrase guarantees that records with duplicate sort keys remain in the same sequence as they were in the input file.

14.5 COLLATING SEQUENCE IS Alphabet-Name Phrase

This phrase lets you specify a collating sequence other than the ASCII default. You must define collating sequences in the SPECIAL-NAMES paragraph of the Environment Division. A sequence specified in the COLLATING SEQUENCE IS phrase of the SORT statement overrides a sequence specified in the PROGRAM COLLATING SEQUENCE IS phrase of the OBJECT-COMPUTER paragraph.

Example 14–3 shows the alphabet name NEWSEQUENCE overriding the EBCDIC-CODE collating sequence.

Example 14–3: Overriding the COLLATING SEQUENCE IS Phrase

```
ENVIRONMENT DIVISION.
OBJECT-COMPUTER.    VAX
                    PROGRAM COLLATING SEQUENCE IS EBCDIC-CODE.
SPECIAL-NAMES.
    ALPHABET NEWSEQUENCE IS "ZYXWVUTSRQPONMLKJIHGFEDCBA"
    ALPHABET EBCDIC-CODE IS EBCDIC.
PROCEDURE DIVISION.
000-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         COLLATING SEQUENCE IS NEWSEQUENCE
         USING INPUT-FILE GIVING OUTPUT-FILE.
```

14.6 File Organization

You can sort any file regardless of its organization; furthermore, the organization of the output file can differ from that of the input file. For example, a sort can have a sequential input file and a relative output file. In this case, the relative key for the first record returned from the sort is 1; the second record's relative key is 2; and so forth. However, if an indexed file is described as output in the GIVING or OUTPUT PROCEDURE phrases, the first sort key associated with the ASCENDING phrase must specify the same character positions specified by the RECORD KEY phrase for that file.

14.7 Multiple Sorts

A program can contain more than one sort file, more than one SORT statement, or both sort files and SORT statements. Example 14–4 uses two sort files to produce two reports with different sort sequences.

Example 14–4: Using Two Sort Files

```
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE1.
01  SORT-REC-1.
    03  S1-KEY-1     PIC X(5).
    03  FILLER       PIC X(40).
    03  S1-KEY-2     PIC X(5).
    03  FILLER       PIC X(50).
SD  SORT-FILE2.
01  SORT-REC-2.
01  SORT-REC-2.
    03  FILLER       PIC X(20).
    03  S2-KEY-1     PIC X(10).
    03  FILLER       PIC X(10).
    03  S2-KEY-2     PIC X(10).
    03  FILLER       PIC X(50).
            .
            .
            .
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-FIRST-SORT.
    SORT SORT-FILE1 ON ASCENDING KEY
                   S1-KEY-1
                   S1-KEY-2
                   WITH DUPLICATES IN ORDER
                   USING INPUT-FILE
                   OUTPUT PROCEDURE IS 050-CREATE-REPORT-1
                                  THRU 300-DONE-REPORT-1.
020-DO-SECOND-REPORT.
    SORT SORT-FILE2 ON ASCENDING KEY
                   S2-KEY-1
                    ON DESCENDING KEY
                   S2-KEY-2
                   USING INPUT-FILE
                   OUTPUT PROCEDURE IS 400-CREATE-REPORT-2
                                  THRU 700-DONE-REPORT-2.
030-END-JOB.
    DISPLAY "PROGRAM ENDED".
    STOP RUN.
050-CREATE-REPORT-1 SECTION.
***********************************************************
*                                                         *
*                                                         *
*   Use the RETURN statement to read the sorted records.  *
*                                                         *
*                                                         *
***********************************************************
300-DONE-REPORT-1 SECTION.
310-EXIT-REPORT-1.
    EXIT.
400-CREATE-REPORT-2 SECTION.
```

(continued on next page)

Example 14–4 (Cont.): Using Two Sort Files

```
***********************************************************
*                                                         *
*                                                         *
*   Use the RETURN statement to read the sorted records. *
*                                                         *
*                                                         *
***********************************************************
700-DONE-REPORT-2 SECTION.
710-EXIT-REPORT.
    EXIT.
```

14.8 Sorting Variable-Length Records

If you specify the USING phrase and the input file contains variable-length records, the sort-file record must not be smaller than the smallest record, nor larger than the largest record, described in the input file.

If you specify the GIVING phrase and the output file contains variable-length records, the sort-file record must not be smaller than the smallest record, nor larger than the largest record, described in the output file.

14.9 Preventing I/O Aborts

All I/O errors detected during a sort can cause abnormal program termination. The USE AFTER STANDARD ERROR PROCEDURE declarative, shown in Example 14–5, specifies error-handling procedures should I/O errors occur.

Example 14–5: Using the AFTER STANDARD ERROR PROCEDURE

```
PROCEDURE DIVISION.
DECLARATIVES.
SORT-FILE SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
SORT-ERROR.
    DISPLAY "I-O TYPE ERROR WHILE SORTING".
    DISPLAY "INPUT-FILE STATUS IS " INPUT-STATUS.
    STOP RUN.
END DECLARATIVES.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON DESCENDING KEY
                         S-KEY-1
                      WITH DUPLICATES IN ORDER
                      USING INPUT-FILE
                      GIVING OUTPUT-FILE.
    DISPLAY "END OF SORT".
    STOP RUN.
```

NOTE

The USE PROCEDURE phrase does not apply to Sort Description (SD) files.

14.10 The MERGE Statement

The MERGE statement combines two or more identically sequenced files and makes their records available, in merged order, to an output procedure or to one or more output files. Use MERGE statement phrases the same way you use their SORT statement phrase equivalents.

In Example 14–6, district sales data is merged into one regional sales file.

Example 14–6: Using the MERGE Statement

```
     .
     .
     .
DATA DIVISION.
FILE SECTION.
SD  MERGE-FILE.
01  MERGE-REC.
    03  FILLER                PIC XX.
    03  M-PRODUCT-CODE        PIC X(10).
    03  FILLER                PIC X(88).
FD  DISTRICT1-SALES.
01  DISTRICT1-REC             PIC X(100).
FD  DISTRICT2-SALES.
01  DISTRICT2-REC             PIC X(100).
FD  REGION1-SALES
01  REGION1-REC               PIC X(100).
PROCEDURE DIVISION.
000-MERGE-FILES.
    MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
          USING DISTRICT1-SALES DISTRICT2-SALES
          GIVING REGION1-SALES.
    STOP RUN.
```

14.11 Sample Programs Using the SORT and MERGE Statements

The programs in Example 14–7, Example 14–8, Example 14–9, Example 14–10, Example 14–11, and Example 14–12 all show how to use the SORT and MERGE statements.

Example 14–7 shows how to use the SORT statement with the USING and GIVING phrases.

Example 14–7: Sorting a File with the USING and GIVING Phrases

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTA.
**************************************************
*   This program shows how to sort               *
*   a file with the USING and GIVING phrases     *
*   of the SORT statement. The fields to be      *
*   sorted are S-KEY-1 and S-KEY-2; they         *
*   contain account numbers and amounts. The     *
*   sort sequence is amount within account       *
*   number.                                      *
*   Notice that OUTPUT-FILE is a relative file. *
**************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       VAX.
OBJECT-COMPUTER.       VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "INPFIL".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
           ORGANIZATION IS RELATIVE.
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM      PIC X(8).
    03  FILLER                 PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
    03  FILLER                 PIC X(53).
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC                     PIC X(100).
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC                    PIC X(100).
PROCEDURE DIVISION.
000-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         WITH DUPLICATES IN ORDER
         USING INPUT-FILE GIVING OUTPUT-FILE.
************************************************************
*   At this point, you could transfer control to another   *
*   section of your program and continue processing.       *
************************************************************
    DISPLAY "END OF PROGRAM SORTA".
    STOP RUN.
```

Example 14–8 shows how to use the USING and OUTPUT PROCEDURE phrases.

Example 14–8: Using the USING and OUTPUT PROCEDURE Phrases

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTB.
**************************************************************
*   This program shows how to sort a file                    *
*   with the USING and OUTPUT PROCEDURE phrases              *
*   of the SORT statement. The program eliminates            *
*   duplicate records by adding their amounts to the         *
*   amount in the first record with the same account         *
*   number. Only records with unique account numbers         *
*   are written to the output file. The fields to be         *
*   sorted are S-KEY-1 and S-KEY-2; they contain account     *
*   numbers and amounts. The sort sequence is amount         *
*   within account number.                                   *
*   Notice that the organization of OUTPUT-FILE is indexed.  *
**************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       VAX.
OBJECT-COMPUTER.       VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "INPFIL".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
           ORGANIZATION IS INDEXED.
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM      PIC X(8).
    03  FILLER                 PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
    03  FILLER                 PIC X(53).
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC                     PIC X(100).
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD
    ACCESS MODE IS SEQUENTIAL
    RECORD KEY IS OUT-KEY.
01  OUT-REC.
    03  OUT-KEY                PIC X(8).
    03  FILLER                 PIC X(92).
WORKING-STORAGE SECTION.
01  INITIAL-SORT-READ          PIC X   VALUE "Y".
01  SAVE-SORT-REC.
    03  SR-ACCOUNT-NUM         PIC X(8).
    03  FILLER                 PIC X(32).
    03  SR-AMOUNT              PIC S9(5)V99.
    03  FILLER                 PIC X(53).
PROCEDURE DIVISION.
000-START SECTION.
005-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
```

(continued on next page)

Example 14–8 (Cont.): Using the USING and OUTPUT PROCEDURE Phrases

```
         USING INPUT-FILE
         OUTPUT PROCEDURE IS 300-CREATE-OUTPUT-FILE
                        THRU 600-DONE-CREATE.
*************************************************************
*    At this point, you could transfer control to another   *
*    section of the program and continue processing.        *
*************************************************************
    DISPLAY "END OF PROGRAM SORTB".
    STOP RUN.

300-CREATE-OUTPUT-FILE SECTION.
350-OPEN-OUTPUT.
    OPEN OUTPUT OUTPUT-FILE.
400-READ-SORT-FILE.
    RETURN SORT-FILE AT END
        PERFORM 500-WRITE-THE-OUTPUT
        CLOSE OUTPUT-FILE
        GO TO 600-DONE-CREATE.
    IF INITIAL-SORT-READ = "Y"
        MOVE SORT-REC TO SAVE-SORT-REC
        MOVE "N" TO INITIAL-SORT-READ
        GO TO 400-READ-SORT-FILE.
450-COMPARE-ACCOUNT-NUM.
    IF S-ACCOUNT-NUM = SR-ACCOUNT-NUM
        ADD S-AMOUNT TO SR-AMOUNT
        GO TO 400-READ-SORT-FILE.
500-WRITE-THE-OUTPUT.
    MOVE SAVE-SORT-REC TO OUT-REC.
    WRITE OUT-REC INVALID KEY
        DISPLAY "INVALID KEY " SR-ACCOUNT-NUM " SORTB ABORTED"
        CLOSE OUTPUT-FILE STOP RUN.
550-GET-A-REC.
    MOVE SORT-REC TO SAVE-SORT-REC.
    GO TO 400-READ-SORT-FILE.
600-DONE-CREATE SECTION.
650-EXIT-PARAGRAPH.
    EXIT.
```

Example 14–9 shows how to use the INPUT PROCEDURE and OUTPUT PROCEDURE phrases.

Example 14–9: Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTC.
*********************************************************
*   This program shows how to use the INPUT             *
*   PROCEDURE and OUTPUT PROCEDURE phrases of the       *
*   SORT statement. Input to the sort is two files      *
*   containing the same type of data. Records with      *
*   a "D" status-code are not released to the sort.     *
*   The program eliminates duplicate records by         *
*   adding their amounts to the amount in the first     *
*   record with the same account number. Only records   *
*   with unique account numbers are written to          *
*   the output file. The fields to be sorted are        *
*   S-KEY-1 and S-KEY-2. The sort sequence is amount    *
*   within account number.                              *
*********************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       VAX.
OBJECT-COMPUTER.       VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FIRST-FILE ASSIGN TO "FILE01".
    SELECT SECOND-FILE ASSIGN TO "FILE02".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL".
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM      PIC X(8).
    03  FILLER                 PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
    03  FILLER                 PIC X(53).
FD  FIRST-FILE
    LABEL RECORDS ARE STANDARD.
01  RECORD1.
    03  FILLER                 PIC X(99).
    03  R1-STATUS-CODE       PIC X.
FD  SECOND-FILE
    LABEL RECORDS ARE STANDARD.
01  RECORD2.
    03  FILLER                 PIC X(99).
    03  R2-STATUS-CODE       PIC X.
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC                    PIC X(100).
WORKING-STORAGE SECTION.
01  INITIAL-SORT-READ          PIC X   VALUE "Y".
01  FILE01-COUNT               PIC 9(5) VALUE ZEROES.
01  FILE02-COUNT               PIC 9(5) VALUE ZEROES.
```

(continued on next page)

Example 14–9 (Cont.): Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

```
01  SORT-COUNT                 PIC 9(5) VALUE ZEROES.
01  OUTPUT-COUNT               PIC 9(5) VALUE ZEROES.
01  SAVE-SORT-REC.
    03  SR-ACCOUNT-NUM         PIC X(8).
    03  FILLER                 PIC X(32).
    03  SR-AMOUNT              PIC S9(5)V99.
    03  FILLER                 PIC X(53).
PROCEDURE DIVISION.
000-START SECTION.
005-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         INPUT PROCEDURE IS 010-GET-INPUT
                       THRU 200-DONE-INPUT-GET
         OUTPUT PROCEDURE IS 300-CREATE-OUTPUT-FILE
                        THRU 600-DONE-CREATE.
*********************************************************
*   Notice the use of DISPLAY and record counters to    *
*   produce sort statistics.                            *
*********************************************************
    DISPLAY "TOTAL FIRST-FILE RECORDS IS              " FILE01-COUNT.
    DISPLAY "TOTAL SECOND-FILE RECORDS IS             " FILE02-COUNT.
    DISPLAY "TOTAL NUMBER OF SORTED RECORDS IS        " SORT-COUNT.
    DISPLAY "TOTAL NUMBER OF OUTPUT RECORDS IS        " OUTPUT-COUNT.
*************************************************************
*   At this point, you could transfer control to another    *
*   section of the program  and continue processing.        *
*************************************************************
    DISPLAY "END OF PROGRAM SORTC".
    STOP RUN.
010-GET-INPUT SECTION.
050-OPEN-FILES.
    OPEN INPUT FIRST-FILE.
100-READ-FIRST-FILE.
    READ FIRST-FILE AT END
        CLOSE FIRST-FILE
        OPEN INPUT SECOND-FILE
        GO TO 150-READ-SECOND-FILE.
    ADD 1 TO FILE01-COUNT.
    IF R1-STATUS-CODE = "D"
        GO TO 100-READ-FIRST-FILE.
    RELEASE SORT-REC FROM RECORD1.
    GO TO 100-READ-FIRST-FILE.
150-READ-SECOND-FILE.
    READ SECOND-FILE AT END
        CLOSE SECOND-FILE
        GO TO 200-DONE-INPUT-GET.
    ADD 1 TO FILE02-COUNT.
    IF R2-STATUS-CODE = "D"
        GO TO 150-READ-SECOND-FILE.
    RELEASE SORT-REC FROM RECORD2.
    GO TO 150-READ-SECOND-FILE.
200-DONE-INPUT-GET SECTION.
250-EXIT-PARAGRAPH.
    EXIT.
```

(continued on next page)

Example 14–9 (Cont.): Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

```
300-CREATE-OUTPUT-FILE SECTION.
350-OPEN-OUTPUT.
    OPEN OUTPUT OUTPUT-FILE.
400-READ-SORT-FILE.
    RETURN SORT-FILE AT END
        PERFORM 500-WRITE-THE-OUTPUT
        CLOSE OUTPUT-FILE
        GO TO 600-DONE-CREATE.
    ADD 1 TO SORT-COUNT.
    IF INITIAL-SORT-READ = "Y"
        MOVE SORT-REC TO SAVE-SORT-REC
        MOVE "N" TO INITIAL-SORT-READ
        GO TO 400-READ-SORT-FILE.
450-COMPARE-ACCOUNT-NUM.
    IF S-ACCOUNT-NUM = SR-ACCOUNT-NUM
        ADD S-AMOUNT TO SR-AMOUNT
        GO TO 400-READ-SORT-FILE.
500-WRITE-THE-OUTPUT.
    MOVE SAVE-SORT-REC TO OUT-REC.
    WRITE OUT-REC.
    ADD 1 TO OUTPUT-COUNT.
550-GET-A-REC.
    MOVE SORT-REC TO SAVE-SORT-REC.
    GO TO 400-READ-SORT-FILE.
600-DONE-CREATE SECTION.
650-EXIT-PARAGRAPH.
    EXIT.
```

Example 14–10 shows how to use the COLLATING SEQUENCE IS phrase.

Example 14–10: Using the COLLATING SEQUENCE IS Phrase

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTD.
**************************************************
*   This program sorts a file into a non-ASCII   *
*   collating sequence. The collating sequence   *
*   is defined by the alphabet-name MYSEQUENCE   *
*   in the SPECIAL-NAMES paragraph of the        *
*   ENVIRONMENT DIVISION.                        *
*   The collating sequence is:                   *
*       1. The letters A to Z                    *
*       2. The digits 0 to 9                     *
**************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       VAX.
OBJECT-COMPUTER.       VAX.
```

(continued on next page)

Example 14–10 (Cont.): Using the COLLATING SEQUENCE IS Phrase

```
SPECIAL-NAMES.
    ALPHABET MYSEQUENCE IS
             "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 ".
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "INPFIL".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL".
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NAME     PIC X(23).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC                     PIC X(30).
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC                    PIC X(30).
PROCEDURE DIVISION.
000-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         COLLATING SEQUENCE IS MYSEQUENCE
         USING INPUT-FILE GIVING OUTPUT-FILE.
*************************************************************
*   At this point, you could transfer control to another    *
*   section of the program and continue processing.         *
*************************************************************
    DISPLAY "END OF PROGRAM SORTD".
    STOP RUN.
```

Example 14–11 is an example of creating a new sort key.

Example 14–11: Creating a New Sort Key

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   SORTE.
*************************************************
*   This program increases the size of the     *
*   variable input records by a new six-        *
*   character field and uses this field         *
*   as the sort key.                            *
*************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX.
OBJECT-COMPUTER.    VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN TO "INFILE".
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
    SELECT OUT-FILE ASSIGN TO "OUTFILE".
```

(continued on next page)

Example 14–11 (Cont.): Creating a New Sort Key

```
DATA DIVISION.
FILE SECTION.
FD      INFILE
        RECORD VARYING FROM 100 TO 490 CHARACTERS
        DEPENDING ON IN-LENGTH.
01      INREC.
        03 ACCOUNT                 PIC 9(5).
        03 INCOME-FIRST-QUARTER    PIC 9(5)V99.
        03 INCOME-SECOND-QUARTER   PIC 9(5)V99.
        03 INCOME-THIRD-QUARTER    PIC 9(5)V99.
        03 INCOME-FOURTH-QUARTER   PIC 9(5)V99.
        03 ORDER-COUNT             PIC 9(2).
        03 ORDERS OCCURS 1 TO 7 TIMES
            DEPENDING ON ORDER-COUNT.
              05  ORDER-DATE       PIC 9(6).
              05  FILLER           PIC X(59).
SD      SORT-FILE
        RECORD VARYING FROM 106 TO 496 CHARACTERS
        DEPENDING ON SORT-LENGTH.
01     SORT-REC.
        03  SORT-ANNUAL-INCOME     PIC 9(6).
        03  SORT-REST-OF-RECORD    PIC X(490).
FD      OUT-FILE
        RECORD VARYING FROM 106 TO 496 CHARACTERS
        DEPENDING ON OUT-LENGTH.
01      OUT-REC                    PIC X(496).
WORKING-STORAGE SECTION.
01  IN-LENGTH                      PIC 9(3) COMP.
01  SORT-LENGTH                    PIC 9(3) COMP.
01  OUT-LENGTH                     PIC 9(3) COMP.
PROCEDURE DIVISION.
000-START SECTION.
005-SORT-HERE.
    SORT SORT-FILE
           ON DESCENDING SORT-ANNUAL-INCOME
           INPUT PROCEDURE 010-GET-INPUT
                      THRU 070-DONE-INPUT
           OUTPUT PROCEDURE 100-WRITE-OUTPUT.
    DISPLAY "END OF PROGRAM SORTE".
    STOP RUN.
010-GET-INPUT SECTION.
020-OPEN-INPUT.
    OPEN INPUT INFILE.
030-READ-INPUT.
    READ INFILE AT END
        CLOSE INFILE
        GO TO 070-DONE-INPUT.
```

(continued on next page)

Example 14–11 (Cont.): Creating a New Sort Key

```
040-ADD-INCOME.
    ADD INCOME-FIRST-QUARTER
        INCOME-SECOND-QUARTER
        INCOME-THIRD-QUARTER
        INCOME-FOURTH-QUARTER
        GIVING SORT-ANNUAL-INCOME.
050-CREATE-SORT-REC.
    ADD 6 IN-LENGTH GIVING SORT-LENGTH.
    MOVE INREC TO SORT-REST-OF-RECORD.
    RELEASE SORT-REC.
    GO TO 030-READ-INPUT.
070-DONE-INPUT SECTION.
080-EXIT.
    EXIT.
100-WRITE-OUTPUT SECTION.
110-OPEN.
    OPEN OUTPUT OUT-FILE.
120-WRITE.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 130-DONE.
    MOVE SORT-LENGTH TO OUT-LENGTH.
    WRITE OUT-REC.
    GO TO 120-WRITE.
130-DONE.
    EXIT.
```

Example 14–12 merges three identically sequenced files into one file.

Example 14–12: Merging Files

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    MERGE01.
******************************************************
*   This program merges three identically sequenced  *
*   regional sales files into one total sales file.  *
*   The program adds sales amounts and writes one    *
*   record for each product code.                    *
******************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.     VAX.
OBJECT-COMPUTER.     VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT REGION1-SALES ASSIGN TO "REG1SLS".
    SELECT REGION2-SALES ASSIGN TO "REG2SLS".
    SELECT REGION3-SALES ASSIGN TO "REG3SLS".
    SELECT MERGE-FILE    ASSIGN TO "MRGFILE".
    SELECT TOTAL-SALES   ASSIGN TO "TOTLSLS".
```

(continued on next page)

Example 14–12 (Cont.): Merging Files

```
DATA DIVISION.
FILE SECTION.
FD  REGION1-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION1-RECORD            PIC X(100).
FD  REGION2-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION2-RECORD            PIC X(100).
FD  REGION3-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION3-RECORD            PIC X(100).
SD  MERGE-FILE.
    01  MERGE-REC.
        03  M-REGION-CODE     PIC XX.
        03  M-PRODUCT-CODE    PIC X(10).
        03  M-SALES-AMT       PIC S9(7)V99.
        03  FILLER            PIC X(79).
FD  TOTAL-SALES
    LABEL RECORDS ARE STANDARD.
01  TOTAL-RECORD              PIC X(100).
WORKING-STORAGE SECTION.
01  INITIAL-READ              PIC X  VALUE "Y".
01  THE-COUNTERS.
    03  PRODUCT-AMT           PIC S9(7)V99.
    03  REGION1-AMT           PIC S9(9)V99.
    03  REGION2-AMT           PIC S9(9)V99.
    03  REGION3-AMT           PIC S9(9)V99.
    03  TOTAL-AMT             PIC S9(11)V99.
01  SAVE-MERGE-REC.
    03  S-REGION-CODE         PIC XX.
    03  S-PRODUCT-CODE        PIC X(10).
    03  S-SALES-AMT           PIC S9(7)V99.
    03  FILLER                PIC X(79).
PROCEDURE DIVISION.
000-START SECTION.
010-MERGE-FILES.
    OPEN OUTPUT TOTAL-SALES.
    MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
          USING REGION1-SALES REGION2-SALES REGION3-SALES
          OUTPUT PROCEDURE IS 020-BUILD-TOTAL-SALES
                         THRU 100-DONE-TOTAL-SALES.
    DISPLAY "TOTAL SALES FOR REGION 1 " REGION1-AMT.
    DISPLAY "TOTAL SALES FOR REGION 2 " REGION2-AMT.
    DISPLAY "TOTAL SALES FOR REGION 3 " REGION3-AMT.
    DISPLAY "TOTAL ALL SALES          " TOTAL-AMT.
    CLOSE TOTAL-SALES.
    DISPLAY "END OF PROGRAM MERGE01".
    STOP RUN.
020-BUILD-TOTAL-SALES SECTION.
030-GET-MERGE-RECORDS.
    RETURN MERGE-FILE AT END
           MOVE PRODUCT-AMT TO S-SALES-AMT
           WRITE TOTAL-RECORD FROM SAVE-MERGE-REC
           GO TO 100-DONE-TOTAL-SALES.
    IF INITIAL-READ = "Y"
           MOVE "N" TO INITIAL-READ
           MOVE MERGE-REC TO SAVE-MERGE-REC
           PERFORM 050-TALLY-AMOUNTS
           GO TO 030-GET-MERGE-RECORDS.
```

(continued on next page)

Example 14–12 (Cont.): Merging Files

```
040-COMPARE-PRODUCT-CODE.
    IF M-PRODUCT-CODE = S-PRODUCT-CODE
           PERFORM 050-TALLY-AMOUNTS
           GO TO 030-GET-MERGE-RECORDS.
    MOVE PRODUCT-AMT TO S-SALES-AMT.
    MOVE ZEROES TO PRODUCT-AMT.
    WRITE TOTAL-RECORD FROM SAVE-MERGE-REC.
    MOVE MERGE-REC TO SAVE-MERGE-REC.
    GO TO 040-COMPARE-PRODUCT-CODE.
050-TALLY-AMOUNTS.
    ADD M-SALES-AMT TO PRODUCT-AMT TOTAL-AMT.
    IF M-REGION-CODE = "01"
           ADD M-SALES-AMT TO REGION1-AMT.
    IF M-REGION-CODE = "02"
           ADD M-SALES-AMT TO REGION2-AMT.
    IF M-REGION-CODE = "03"
           ADD M-SALES-AMT TO REGION3-AMT.
100-DONE-TOTAL-SALES SECTION.
120-DONE.
    EXIT.
```

Chapter 15

Database Programming with VAX COBOL

VAX COBOL database programming allows you to access data without designing separate files for specific applications. This chapter introduces the database programmer to the database management system (VAX DBMS) and the COBOL data manipulation language (DML). It also discusses the following topics:

- VAX COBOL database program development
- VAX COBOL database concepts
- VAX COBOL programming tips and techniques
- Debugging and testing VAX COBOL database programs

Database programmers and readers unfamiliar with VAX DBMS concepts and definitions should run the online self-paced demonstration package (see Section 15.1) as a prerequisite to this chapter. The demonstration package lets you test VAX DBMS features and concepts as you learn them. You should also read the introductory material to VAX application development with the VAX Information Architecture, and the VAX DBMS documentation on database administration.

15.1 The Self-Paced Demonstration Package

To help you learn how to use a database, Digital has provided you with a database called PARTS. PARTS is an online self-paced demonstration database configured to show some of the features of VAX DBMS. You create the PARTS database as part of the demonstration package. Examples in this chapter refer to either the PARTSS1 or PARTSS3 subschema in the PARTS database. A complete listing of the PARTS schema, including the PARTSS1 and PARTSS3 subschemas, can be found in the VAX DBMS documentation on data manipulation.

Before beginning the demonstration, you should do the following:

1. Create your own node in CDD/Plus using the Dictionary Management Utility (DMU). (Refer to the CDD/Plus documentation for more information.)

   ```
   $ RUN SYS$SYSTEM:DMU RET
   DMU> CREATE nodename RET
   DMU> SHOW DEFAULT RET
       defaultname
   DMU> EXIT RET
   $
   ```

   where:

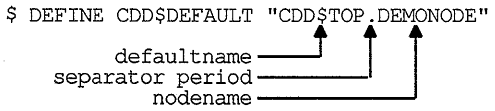
   nodename names the new node in the CDD to contain your personal PARTS database.

   defaultname is your CDD default.

For example:

```
$  RUN SYS$SYSTEM:DMU RET
DMU>  CREATE DEMONODE RET
DMU>  SHOW DEFAULT RET
    CDD$TOP
DMU>  EXIT RET
$
```

2. Define CDD$DEFAULT using the defaultname, a period, and the nodename from step 1. For example:



```
$ DEFINE CDD$DEFAULT "CDD$TOP.DEMONODE"
```

ZK-1434A-GE

To run the demonstration package, type the following:

```
$  @SYS$COMMON:[SYSTEST.DBM]DBMDEMO   RET
```

You must run the entire demonstration to create and load the PARTS database. If you have already created the PARTS database but are unsure of or have changed its contents, you can reload it by running option 11 of the self-paced demonstration package.

The demonstration package creates the NEW.ROO database instance. See the VAX Information Architecture documentation for more information. If you have any problems with the demonstration package, see your system manager or database administrator.

15.2 VAX COBOL Data Manipulation Language (DML)

The VAX COBOL data manipulation language (DML) is a programming language extension that provides a way for a COBOL application program to access a database. A VAX COBOL database application program contains DML statements that tell the Database Control System (DBCS) what to do with specified data; the DBCS provides all database processing control at run time. The four classes of DML statements are *data definition*, *control*, *retrieval*, and *update*. An explanation of each class follows:

- **Data definition**—These entries define the specific part of the database to be accessed by the application program and any keeplists needed to navigate through it. The entries also result in the creation of a database user work area (UWA). Transfer of data between your program and the database takes place in the UWA. Your program delivers data for the DBCS to this area; it is here that the DBCS places data requested from the database for retrieval to your program.

SUB-SCHEMA SECTION	Is the first section of the Data Division. It contains two paragraphs: the Subschema entry (DB) and the Keeplist Description entry (LD).
DB	Names the target subschema, translates subschema record descriptions to compatible VAX COBOL record descriptions, and creates a user work area (UWA).

LD	Names a keeplist to help you navigate through the database.

For more information on these entries see the Data Division section of the *VAX COBOL Reference Manual.*

- **Control**—The DML control functions tell the DBCS when and how to begin or end a database transaction.

COMMIT	Terminates your transaction, makes permanent all changes made to the database since the last quiet point, and establishes a new quiet point for the next run unit.
READY	Prepares selected realms for use.
ROLLBACK	Ends your transaction, cancels all changes made to the database since the start of your transaction, empties all keeplists, and nulls all currency indicators.

- **Retrieval**—The DML retrieval functions are used to find a record in the database and, if necessary, retain the record in the user work area (UWA) for later use.

FIND	Locates a record in the database.
FIND ALL	Locates all records specified in the database and puts them in a keeplist.
FETCH	Locates a record in the database, retrieves its data item values, and places them in the user work area (UWA).
FREE	Releases references to records.
GET	Retrieves data item values of a previously located record and places them in the user work area (UWA).
KEEP	Remembers a record so you can later refer to it.

Records can be found in several ways in the database. By using a Record Selection Expression in a FIND or FETCH statement, a program has four formats to choose from: (1) database key identifier access, (2) set owner access, (3) record search access, or (4) DB-KEY access. The *VAX COBOL Reference Manual* explains these in detail.

A COBOL program can sequentially search the database or individual realm. In all cases, once a record is found by the COBOL application program, the DBCS sets a currency indicator to hold the database key value of that record or the position of that record. The COBOL program can indirectly use this value in KEEP, FIND ALL, or FREE statements or use the RETAINING option as a placemarker to help the program navigate through the database.

- **Update**—These functions allow the creation, modification, and deletion of database records.

CONNECT	Makes a record a member in one or more sets.
DISCONNECT	Removes a record from one or more sets.
ERASE	Deletes records from the database.
MODIFY	Changes the contents of a record in the database.
RECONNECT	Moves a record from one occurrence of a set type to another (possibly the same) occurrence.
STORE	Adds a record to the database.

The *VAX COBOL Reference Manual* discusses the effects of the schema data definition language (DDL) INSERTION and RETENTION options on each of the DML update verbs.

Once a record has been located by a COBOL program, it can be changed or even erased from the database. DML programming operations also change the fundamental relationships within sets, causing records to change as well. For example, each set is owned by a record or VAX DBMS itself. If the program erases a record that is the owner of the set, all member records may also be deleted.

The *VAX COBOL Reference Manual* contains more information on DML statements, database conditional expressions, and the special registers DB-CONDITION, DB-CURRENT-RECORD-NAME, DB-CURRENT-RECORD-ID, DB-UWA, and DB-KEY.

15.3 Creating a VAX COBOL DML Program

When you create a VAX COBOL DML program, you must include the SUB-SCHEMA SECTION entry as the first section in the Data Division. The SUB-SCHEMA SECTION is followed by a DB statement and the DML verbs described in this chapter.

15.4 Compiling a VAX COBOL DML Program

Your database administrator (DBA) creates schema and subschema definitions in CDD/Plus. These record definitions are defined in DMU format and are intended to serve all VAX languages that might access them. In this format, the record definitions are not compatible with COBOL record definitions. Therefore, when the VAX COBOL compiler retrieves the subschema definition from CDD/Plus, it translates the file into an internal form acceptable to the VAX COBOL compiler.

If the translation results in compiler errors, they will probably be fatal. For example:

```
              DB  PARTSS4 WITHIN PARTS.
                  1
%COBOL-F-ERROR  513, (1) Reserved word "DIVISION" used as name in
                         sub-schema
%COBOL-F-ERROR  513, (1) Reserved word "QUOTE" used as name in
                         sub-schema
```

You should alert your DBA to any errors resulting from a DB statement.

You can define the logical name CDD$DEFAULT as the starting schema node in CDD/Plus. There is only one logical name translation in the DB statement for schema-name. If you do not define it, CDD$TOP is the default.

NOTE

> You must recompile a VAX COBOL DML program each time the subschema referenced by a DB statement is created. At compile time, the date and time of subschema creation (date and time stamps) are included with the translated subschema record definitions. If you do not recompile, your program will receive a fatal error at run time.

15.4.1 Copying Database Records in a VAX COBOL Program

A separately compiled VAX COBOL database program must include the SUB-SCHEMA SECTION header and only one DB statement. The compiler copies and translates the record, set, and realm definitions in the subschema named by the DB statement into compatible VAX COBOL record definitions. You will not see any database record definitions listed immediately following the DB statement. The translated record, set, and realm definitions are only in the compiler's subschema map listing. To list these definitions in your program listing, use the /MAP compiler command line qualifier.

15.4.2 Using the /MAP Compiler Qualifier

Use the /MAP compiler qualifier to generate a subschema map containing a translated subschema listing. Section 15.29 contains two subschema map listings and explains how to read them. The following example compiles DBPROG and creates a listing that includes a subschema map:

```
$  COBOL/LIST/MAP DBPROG
```

15.5 Linking a VAX COBOL DML Program

VAX COBOL DML programs must be linked with the shareable VAX DBMS Library (SYS$LIBRARY:DBMDML/OPT). This library was created as part of the VAX DBMS installation procedure. Therefore, to link a VAX COBOL DML object program named DMLPROG.OBJ with the shareable VAX DBMS Library, you would use this DCL command:

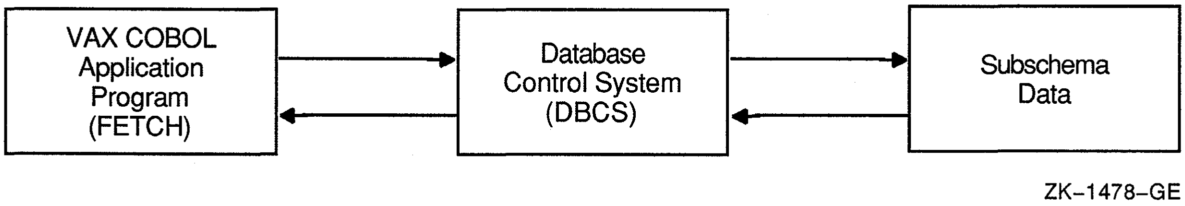
```
$  LINK DMLPROG,SYS$LIBRARY:DBMDML/OPT
```

15.6 Running a VAX COBOL DML Program

You use the DCL command RUN to execute your VAX COBOL DML program. At run time, the Database Control System (DBCS) fills a variety of roles in VAX COBOL. Its major functions are to monitor database usage, act as an intermediary between VAX COBOL and the VMS operating system, and manipulate database records on behalf of user programs. Upon execution of the first DML statement, the DBCS implicitly executes a BIND statement that links the run unit to the database. If the BIND statement is unsuccessful, a database exception occurs.

The DBCS also enforces the subschema view of the database. For example, a database schema record may contain 20 data items. However, a subschema record may only define 10 of those data items. If a FETCH statement references this record, the DBCS only retrieves those defined 10 data items and makes them available to the COBOL program in the user work area (UWA). The other 10 items are not available to the COBOL program. Figure 15–1 illustrates the run-time relationships between an application program requesting subschema data (a FETCH statement, for example), the DBCS, and the data the subschema describes.

Figure 15–1: Database and Application Program Relationship


VAX COBOL Application Program (FETCH)
Database Control System (DBCS)
Subschema Data
ZK–1478–GE

15.7 A Database

A database is a collection of your organization's data gathered into a number of logically related units. The database administrator (DBA) and representatives from user departments decide on the organization's informational needs. After these individuals agree on the contents of the database, the DBA assumes responsibility for designing, creating, and maintaining the database.

15.8 Schema

The schema is a program written by the DBA using DDL statements. It describes the logical structure of the database, defining all record types, set types, areas, and data items in the database. The DBA writes the schema independently of any application run unit. Only one schema can exist for a database. For a more detailed description of the schema DDL, see the VAX DBMS documentation on database administration and design.

15.9 Storage Schema

The storage schema describes the physical structure of the database. It is written by the DBA using data storage description language (DSDL) statements. For a complete description of the storage schema, see the VAX DBMS documentation on database administration and design.

15.10 Subschema

The subschema is a subset of the schema; it is your run unit's view of the database. The DBA uses the subschema DDL to write a subschema, defining only those areas, set types, record types, and data items needed by one or more run units. You specify a subschema to be used by your run unit with the DB statement. A subschema contains data description entries like the record description entries you use for file processing. However, subschema data description entries are not compatible with COBOL data description entries; the VAX COBOL compiler must translate them. The translated entries are made available to the COBOL program at compile time. By using the /MAP compiler qualifier, you obtain a database map showing the translated entries as part of your program listing.

Many subschemas can exist for a database. For further information on writing a subschema, see the VAX documentation on DBMS database administration and design.

15.11 Stream

A stream is an independent access channel between a run unit and a database. A stream has its own keeplists, locks, and currency indicators. You specify a stream to be used by your run unit with the DB statement. Streams let you do the following:

- Access multiple subschemas within the same database
- Access multiple databases

Because streams can lock against one another, it is possible to deadlock within a single process.

In VAX COBOL you can only specify one stream per separately compiled program. To access multiple subschemas within the same database or multiple databases, you must use multiple separately compiled programs and perform calls between the programs. For example, to gain multiple access to the databases OLD.ROO and NEW.ROO, you could set up a run unit as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MULTI-STREAM-1.
DATA DIVISION.
SUB-SCHEMA SECTION.
DB PARTS1 WITHIN PARTS FOR "NEW.ROO" THRU STREAM-1.
    .
    .
    .
    CALL MULTI-STREAM-2
    .
    .
    .
END PROGRAM MULTI-STREAM-1.
IDENTIFICATION DIVISION.
PROGRAM-ID.  MULTI-STREAM-2.
DATA DIVISION.
SUB-SCHEMA SECTION.
DB DEFAULT_SUBSCHEMA WITHIN PARTS FOR "NEW.ROO" THRU STREAM-2.
    .
    .
    .
    CALL MULTI-STREAM-3.
    EXIT PROGRAM.
IDENTIFICATION DIVISION.
PROGRAM-ID.  MULTI-STREAM-3.
DATA DIVISION.
SUB-SCHEMA SECTION.
DB OLDPARTS1 WITHIN OLDPARTS FOR "OLD.ROO" THRU "STREAM-3".
    .
    .
    .
    EXIT PROGRAM.
```

In this run unit the main program (MULTI-STREAM-1) accesses the database NEW.ROO through STREAM-1 and performs a call to a subprogram. The subprogram (MULTI-STREAM-2) accesses another subschema to the database NEW.ROO through STREAM-2 and calls another subprogram. This subprogram (MULTI-STREAM-3) accesses a second database (OLD.ROO) through STREAM-3.

STREAM-1, STREAM-2, and STREAM-3 are stream names. Stream names assign a character string name to the database/subschema combination you specify in your DB statement. For more information, refer to the *VAX COBOL Reference Manual* and the DBMS documentation.

15.12 Using CDD/Plus

You can store the schema, storage schema, and subschemas in CDD/Plus. CDD/Plus separates data descriptions from actual data values that reside in VMS files. (For more information, see the VAX DBMS documentation on Common Data Dictionary Utilities and the CDD/Plus documentation.) Because of this separation, VAX COBOL DML programs can be written independently of data. In addition, several subschemas can describe the same data according to their particular needs. This eliminates the need for redundant data and ensures data integrity.

At compile time, the COBOL DB statement, in effect, references CDD/Plus to obtain the data descriptions of a specific subschema. It is not until run time that the COBOL program has access to the database data values.

15.13 Database Records

A database record, like a record in a file, is a named collection of elementary database data items. Records appear in the database as record occurrences. DBMS records are linked into sets.

In VAX COBOL database applications, you do not describe database records in the COBOL program. Rather, you must use the DB statement to extract and translate subschema record definitions into your COBOL program as COBOL record definitions.

Each record description entry defined by the DBA in the schema describes one record type (see Section 15.16). For example, in Figure 15–8, PART is one record type and SUPPLY is another record type. Any number of records can be stored in a database.

In VAX DBMS, records are also called record occurrences. Figure 15–7 shows one occurrence of PART record type and two occurrences of SUPPLY record type.

The subschema describes records that you can access in your program. Note that subschema record descriptions might define only a portion of a schema record. For example, if a schema record description is 200 characters long, a corresponding subschema record description could be less than 200 characters long and use different data types.

Individual database records are locked by the DBCS as they are retrieved by the run unit, and the degree of locking depends on the specific DML command used. For more information, see Section 15.24.1.1.

15.14 Database Data Item

A database data item is the smallest unit of named data. Data items occur in the database as data values. These values can be character strings or any of several numeric data types.

15.15 Database Key

A database key (dbkey) identifies a record in the database. The value of the database key is the storage address of the database record. You can use this key to refer to the record pointed to by a currency indicator or an entry in a keeplist. For example, KEEP, FIND ALL, and FREE statements store and release these values from a keeplist you define in the SUB-SCHEMA SECTION.

15.16 Record Types

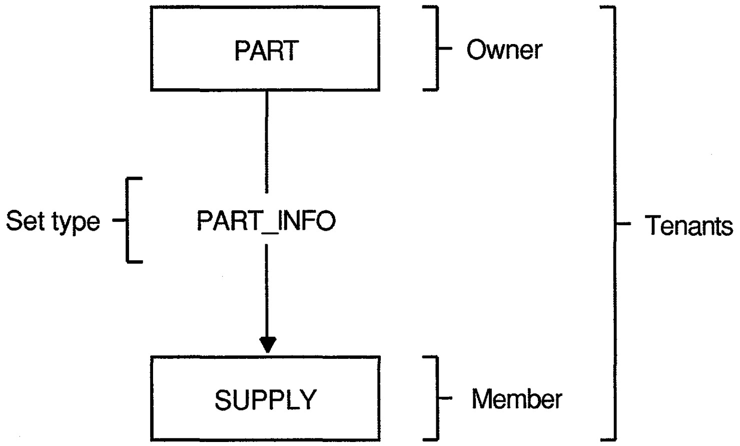
Records are grouped according to common features into record types. The database administrator (DBA) describes record types in the schema; record occurrences exist in the database. For example, a record that contains a specific part name, weight, and cost is a record occurrence. The PART record type, describing the structure of all occurrences of part records, would be defined in the schema. The unqualified term record implies record occurrence.

15.17 Set Types

A set type is a named relationship between two or more record types. The major characteristic of a set type is a relationship that relates one or more member records to an owner record. The owner and members of a set are called tenants of the set. For example, the PART record type could own a SUPPLIER record type in the set PART_INFO.

As with records, the DBA describes set types in the schema; set occurrences exist in the database. The unqualified term set implies set occurrence. A set occurrence is the actual data in the set, not its definition, which is the set type. Figure 15–2 illustrates a set relationship using a Bachman diagram.

Figure 15–2: Bachman Diagram



ZK–1479–GE

A Bachman diagram shows how member records are linked with owner records by arrows that point toward the members. It is a graphic representation of the set relationships between owner and member records used to analyze and document a database design. This simple format can be extended to describe many complex set relationships. The VAX DBMS documentation on data manipulation contains a complete Bachman diagram of the PARTS database.

Most of the examples in this chapter use the set types in the PARTSS1 and PARTSS3 subschemas (see the subschema compiler map listings in Section 15.29 and the Bachman diagrams in Figure 15–3 and Figure 15–4. Figure 15–5 and Figure 15–6 contain three PART records, two VENDOR records, and six SUPPLY records. The SUPPLY records show suppliers' lag times. Lag time starts when an item is ordered and ends when the item is received.

The examples assume the records are in the following order:

1. PART record type: LABEL, CASSETTE, TAPE
2. SUPPLY record type: 4-DAYS, 2-DAYS, 1-MONTH, 1-WEEK, 2-WEEKS, 5-DAYS
3. VENDOR record type: MUSICO INC., SOUND-OFF CO.

NOTE

All occurrence diagrams display member records within a set in counterclockwise order.

Figure 15–3: Partial Bachman Diagram of the PARTSS1 Subschema

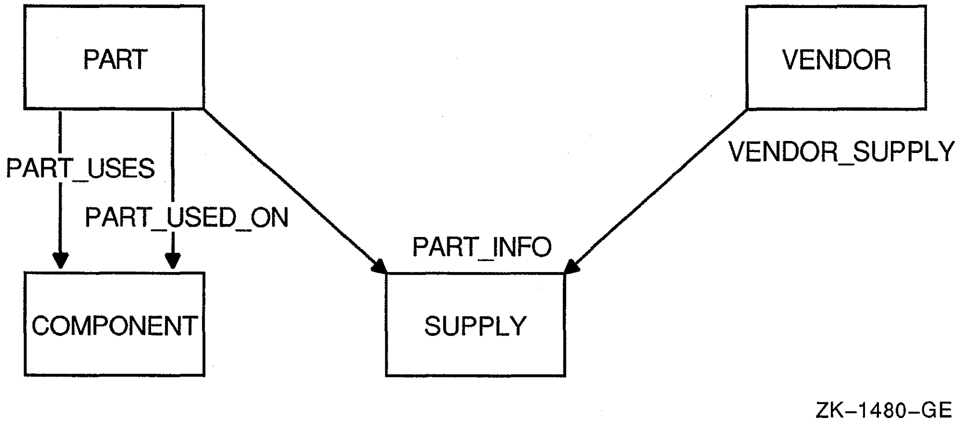


Figure 15–4: Bachman Diagram of the PARTSS3 Subschema

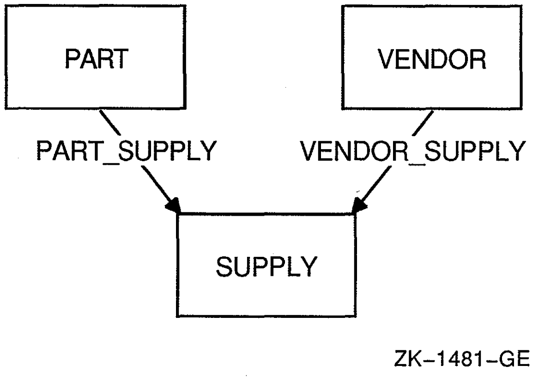


Figure 15–5: Sample Occurrence Diagram 1

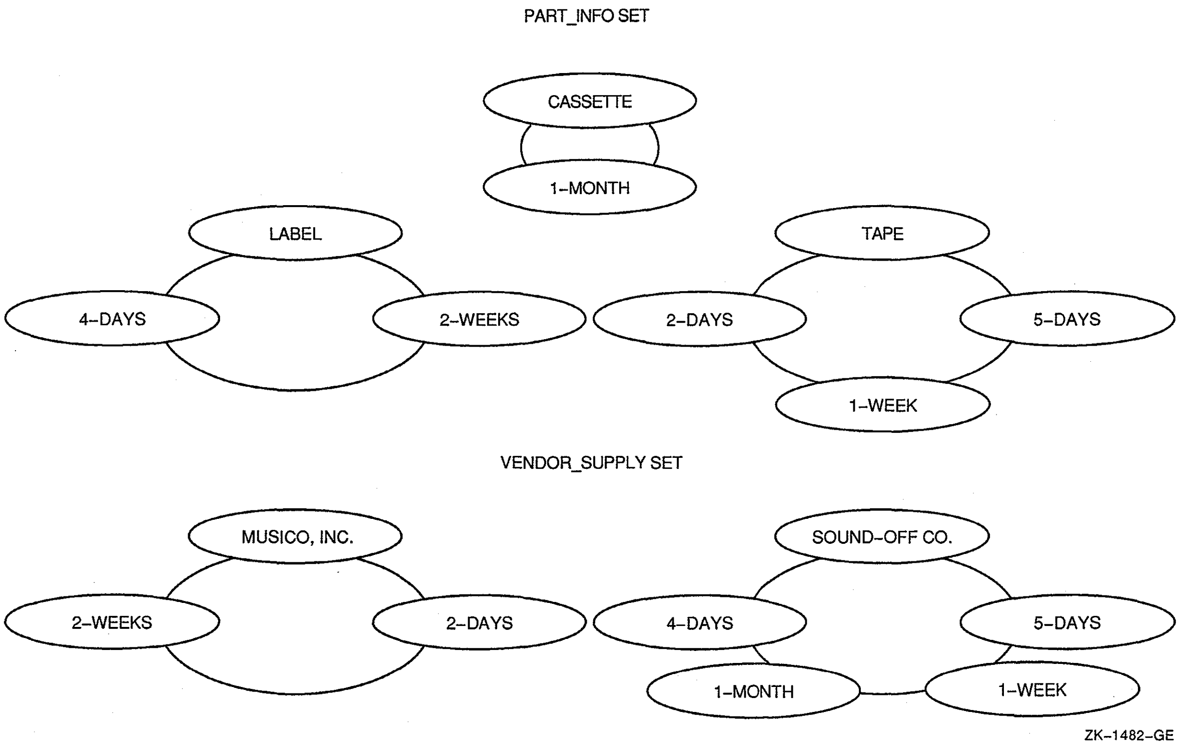
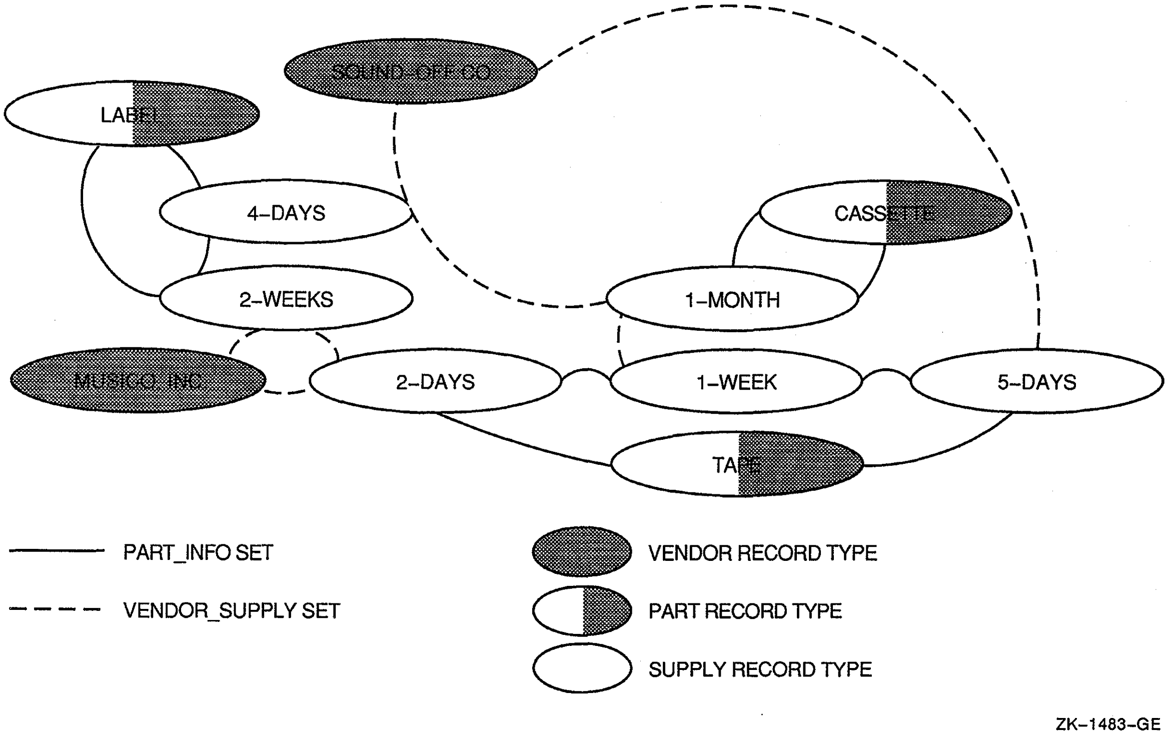


Figure 15–6: Sample Occurrence Diagram 2

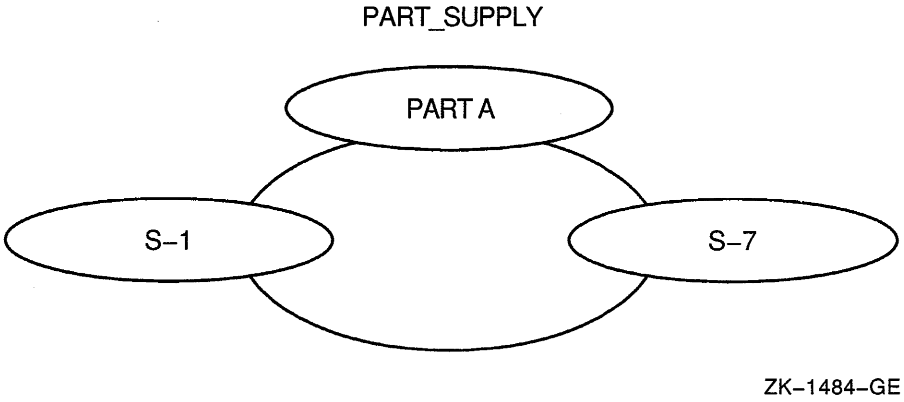


15.18 Sets

Sets are the basic structural units of a database. A set occurrence has one owner record occurrence and zero, one, or several member record occurrences. Figure 15–7 shows one occurrence of PART_SUPPLY set where PART A owner record occurrence owns two SUPPLY member record occurrences.

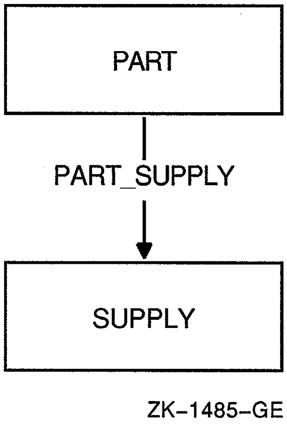
Set types establish a logical relationship between two or more types of records. A subschema usually includes one or more set types. Each set type has one record type that participates as the owner record and one or more record types that participate as members. These owner and member records are grouped into set occurrences.

Figure 15–7: One Occurrence of Set PART_SUPPLY



The DBA can specify a set type where each PART record occurrence can own SUPPLY record occurrences. Figure 15–8 is a Bachman diagram that shows the relationship between PART record types and SUPPLY record types. Bachman diagrams give you a picture of the schema or a portion of the schema. Each record type is enclosed in a box. Each set type is represented by an arrow pointing from the owner record type to the member record type or types. Thus, in Figure 15–8, PART is the owner record type of the PART_SUPPLY set type, and SUPPLY is the member record type.

Figure 15–8: Set Relationship



You can have many set relationships in a subschema. Figure 15–9 shows a set relationship where vendor records are also owners of supply records. You would use this relationship when many parts are supplied by one vendor, and many vendors supply one part. For example, Figure 15–10 shows a gasket supplied by three vendors. The supply records show the minimum quantity each vendor is willing to ship.

Figure 15–9: Set Relationships

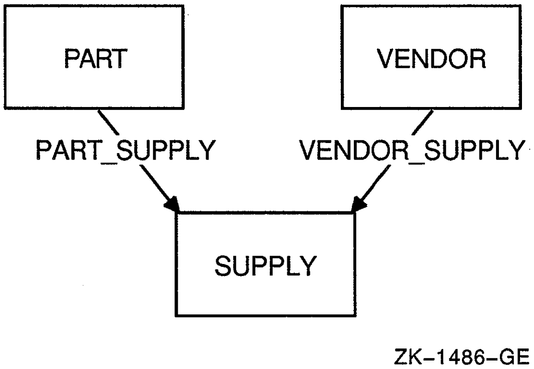
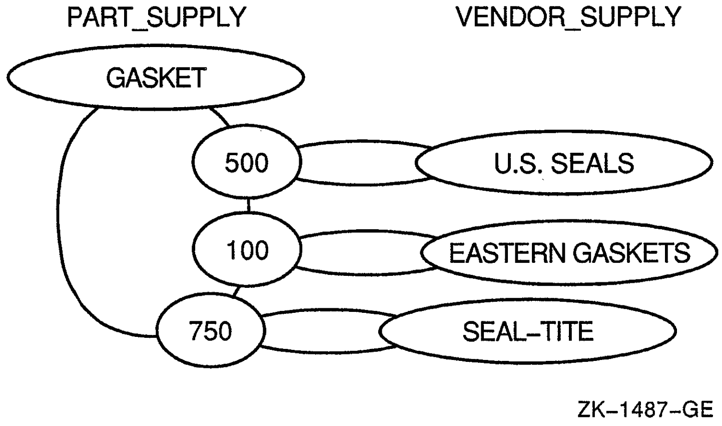


Figure 15–10: Occurrence Diagram of a Relationship Between Two Set Types



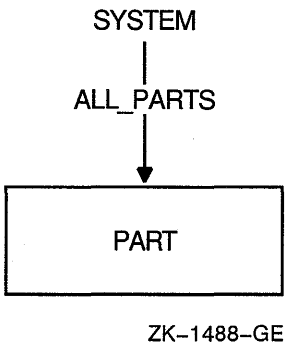
15.18.1 Simple Set Relationships

A simple set relationship contains one owner record type and one or more member record types. Simple relationships are used to represent a basic one-to-many relationship where one owner record occurrence owns zero, one, or several member record occurrences. Simple relationships are created with a single set type. There are three kinds of sets in simple relationships: system-owned sets, simple sets, and forked sets.

15.18.1.1 System-Owned Sets

By definition, a set contains one owner record and may contain zero or more member records. Sets owned by the system, however, have only one occurrence in the database and are called system-owned sets. System-owned sets are used as entry points into the database. You cannot access the owner of a system-owned set (the system), but you can access its member records. System-owned sets are also called singular sets. Figure 15–11 is an example of a system-owned set type.

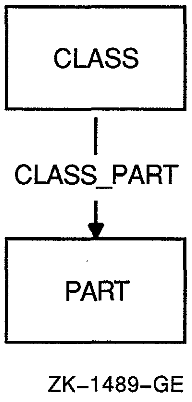
Figure 15–11: Bachman Diagram of a System-Owned Set Type



15.18.1.2 Simple Sets

In simple sets, each set contains only one type of member record. Figure 15–12 is a Bachman diagram of a simple set type where similar parts are grouped by class code. For example, plastic parts could be member records owned by a class record with a class code PL.

Figure 15–12: Bachman Diagram of a Simple Set Type



Example 15–1: Printing a Listing of a Simple Set

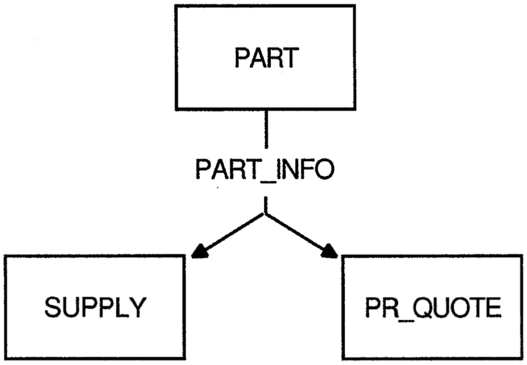
```
PROCEDURE DIVISION.
    .
    .
    .
100-GET-PLASTICS-CLASS.
    MOVE "PL" TO CLASS_CODE
    FIND FIRST CLASS USING CLASS_CODE.
200-GET-PLASTICS-PARTS.
    FETCH NEXT PART WITHIN CLASS_PART
      AT END GO TO 900-DONE-PLASTIC-PARTS.
   *****************************************************
   *   Plastic parts print routine.
   *****************************************************
      GO TO 200-GET-PLASTICS-PARTS.
```

Example 15–1 prints a listing of all parts with a class code of PL.

15.18.1.3 Forked Sets

A forked set has one owner record type and members of two or more different member record types. In most forked sets, the member record types have common data characteristics. One such example is the set type PART_INFO in Figure 15–13, where member record types SUPPLY and PR_QUOTE both contain information about parts.

Figure 15–13: Bachman Diagram of a Forked Set Type



ZK–1490–GE

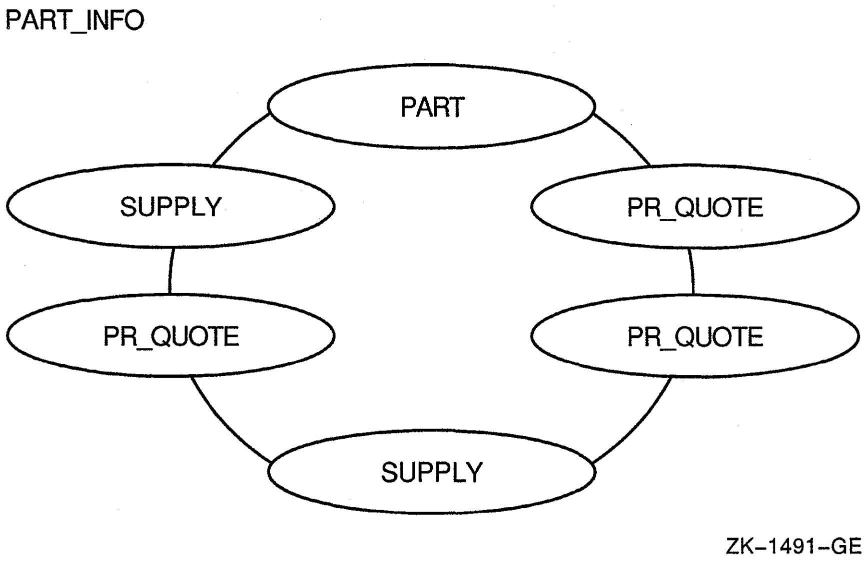
One advantage of a forked set type is the ability to connect many different record types to one set type. Another advantage is that owner records need only one set of pointers to access more than one member record type. Example 15–2 uses the forked set type shown in Figure 15–13 and the forked set occurrence in Figure 15–14 to perform a part analysis.

Example 15–2: Using Forked Sets

```
PROCEDURE DIVISION.
    .
    .
    .
100-GET-PART.
    DISPLAY "TYPE PART ID".
    ACCEPT PART_ID.
    IF PART_ID = "DONE"
       GO TO 900-DONE-PART-INQUIRY.
    FETCH FIRST PART USING PART_ID
       ON ERROR
       DISPLAY "PART " PART_ID " NOT IN DATABASE"
       GO TO 100-GET-PART.
200-GET-SUPPLY-INFO.
    FETCH NEXT SUPPLY WITHIN PART_INFO
       AT END
       FETCH OWNER WITHIN PART_INFO
       GO TO 300-GET-QUOTE-INFO.
 *****************************************************
 * The FETCH OWNER statement resets currency to      *
 * point to the owner. This allows the search for    *
 * PR_QUOTE records to begin with the first member  *
 * record occurrence rather than after the           *
 * last SUPPLY record occurrence.                    *
 *****************************************************
    PERFORM 500-SUPPLY-ANALYSIS.
    GO TO 200-GET-SUPPLY-INFO.
300-GET-QUOTE-INFO.
    FETCH NEXT PR_QUOTE WITHIN PART_INFO
       AT END
       GO TO 100-GET-PART.
    PERFORM 600-QUOTE-ANALYSIS.
    GO TO 300-GET-QUOTE-INFO.
```

Figure 15–14 is an occurrence diagram of a forked set. The figure shows a part record owning five PART_INFO member records.

Figure 15–14: Forked Set Occurrence



15.18.2 Multiset Relationships

A set cannot contain an owner record and a member record of the same type. Nor can a simple set represent a many-to-many relationship. To simulate such relationships, VAX DBMS uses the concept of multiset relationships. Multiset relationships occur when two set types share a common record type called a junction record. The junction record can contain information specific to the relationship, or it can be empty. An empty junction record contains only pointer information used by the DBCS to establish the multiset relationship. This section discusses three kinds of multiset relationships:

- Many-to-many relationships between two types of records
- Many-to-many relationships between records of the same type
- One-to-many relationships between records of the same type

15.18.2.1 Many-to-Many Relationships Between Two Types of Records

To build a many-to-many relationship between two types of records, the DBA uses a junction record. For example, a part can be supplied by many vendors, and one vendor can supply many parts. The SUPPLY record type in Figure 15–15 links or joins PART records with VENDOR records.

Figure 15–15: Bachman Diagram of a Many-to-Many Relationship Between Two Types of Records

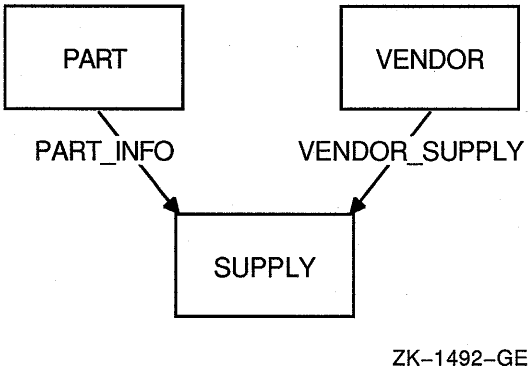
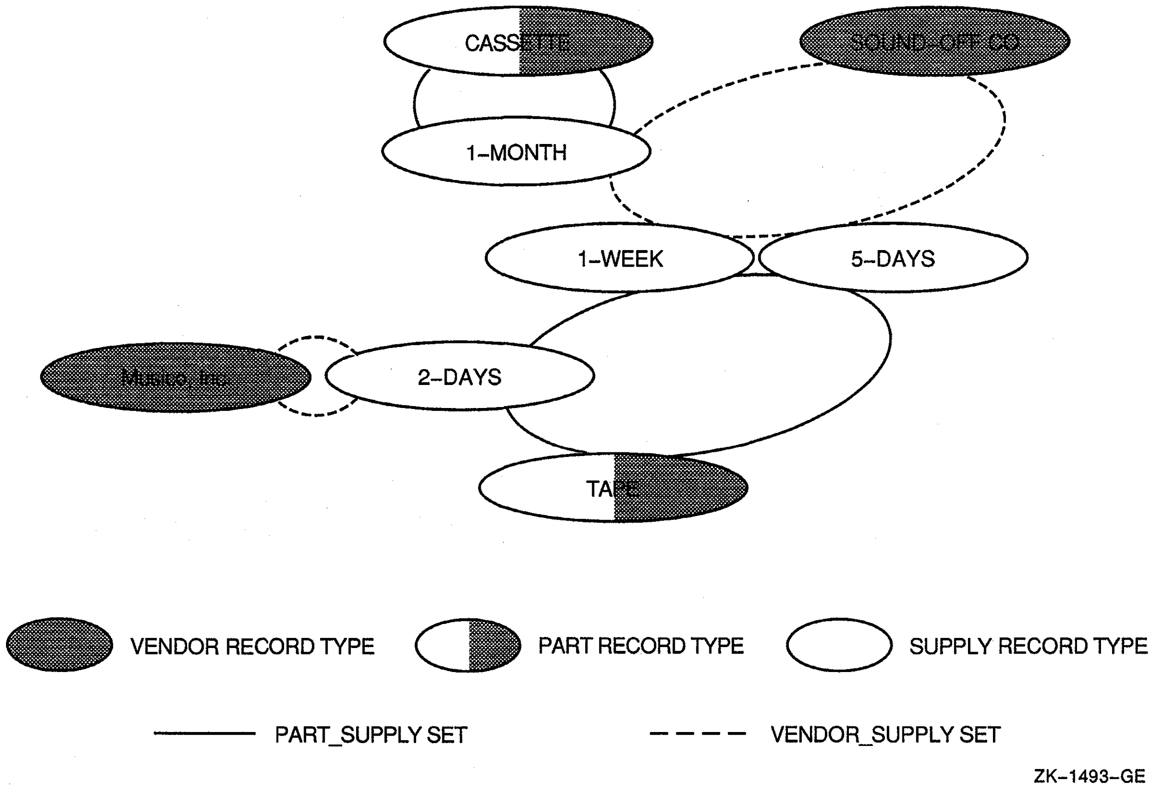


Figure 15–16 is an occurrence diagram of a many-to-many relationship between two types of records. This diagram typifies a many-to-many relationship because it shows a part (TAPE) being supplied by more than one vendor and a vendor (SOUND-OFF CO.) supplying more than one part. You could add additional vendors for a part by joining new supply records to a part and its new vendors. You could also add additional parts supplied by one vendor by joining supply records to the vendor and the new parts.

Figure 15–16: Many-to-Many Relationship Between Two Types of Records

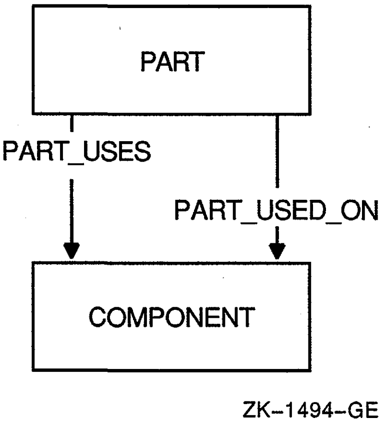


15.18.2.2 Many-to-Many Relationships Between Records of the Same Type

To represent a relationship between record occurrences of the same type, the DBA builds a many-to-many relationship using member records to create the necessary links. Figure 15–17 shows a many-to-many relationship between records of the same type, where PART is the owner of both PART_USES and PART_USED_ON set types and COMPONENT is the junction record.

PART_USES is a bill of materials set type that links a PART owner record through its COMPONENT member records to the part's subassemblies. The link to the subassemblies is from COMPONENT member records up to the PART_USED_ON set type and back to PART owner records.

Figure 15–17: Bachman Diagram of a Many-to-Many Relationship Between Records of the Same Type



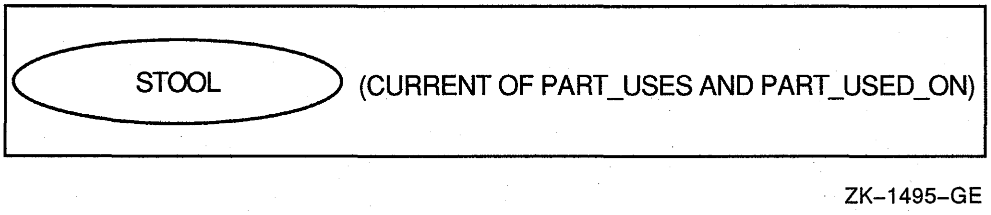
For example, assume you are creating a bill of materials and you have a finished part, a stool, made from one stool seat and four stool legs. Figure 15–18, Figure 15–19, Figure 15–20, and Figure 15–21 show occurrence diagrams of the bill of materials you would need to build a stool.

To complete the bill of materials you have to link the stool seat and stool legs to the finished part, the stool. You would:

1. Use the FIND statement to locate the stool.

```
PROCEDURE DIVISION.
100-FIND-STOOL.
    MOVE "STOOL" TO PART_DESC.
    FIND FIRST PART USING PART_DESC.
```

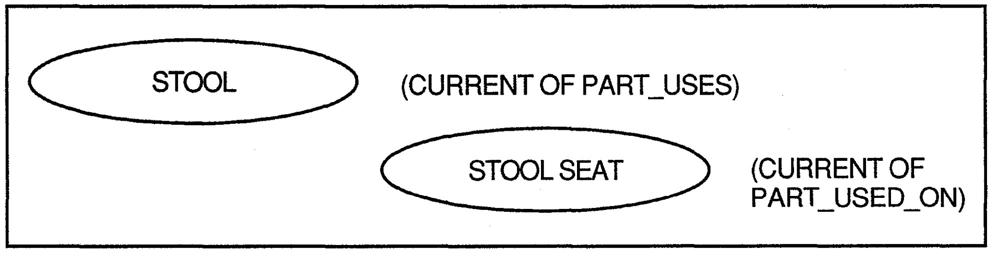
Figure 15–18: Current of PART_USES and PART_USED_ON



2. Use the FIND statement to locate the stool seat retaining PART_USES currency. Because PART usually owns both sets, using a FIND or FETCH statement to locate PART changes both set currency indicators. Retaining PART_USES currency keeps a pointer at STOOL; otherwise, STOOL SEAT would be current for both sets. Section 15.22 discusses currency indicators in more detail.

```
200-FIND-STOOL-SEAT.
    MOVE "STOOL SEAT" TO PART_DESC.
    FIND FIRST PART USING PART_DESC
               RETAINING PART_USES.
```

Figure 15–19: Retain PART_USES Currency



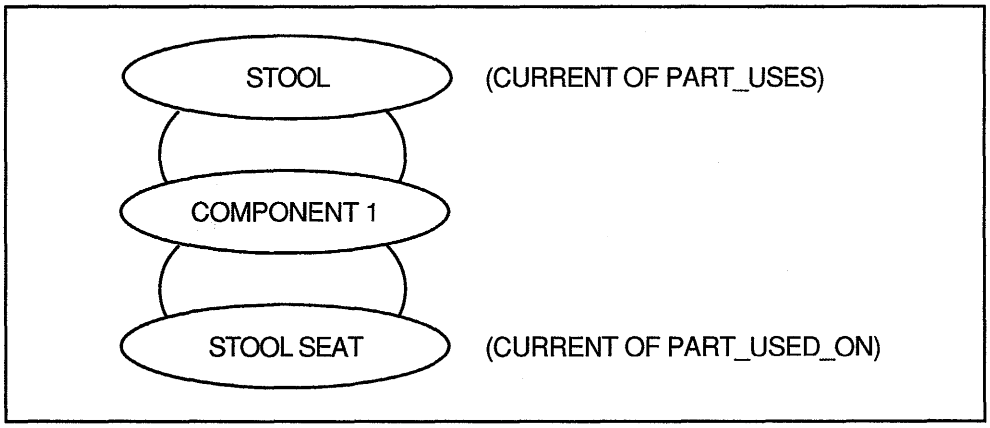
ZK–1496–GE

3. Build a COMPONENT record (component 1), and store it retaining PART_USES currency. Because COMPONENT participates in the PART_USES set, storing it normally changes the set's currency. Therefore, executing a STORE statement with the retaining clause keeps STOOL as current of PART_USES. At this point, STOOL is the PART_USES owner of component 1, and STOOL SEAT is the PART_USED_ON owner of component 1.

   Since the insertion mode for COMPONENT is automatic in both set types, a STORE COMPONENT automatically connects COMPONENT to both set types.

   ```
   300-CONNECT-COMPONENT-1.
       MOVE 1 TO COMP_QUANTITY.
       STORE COMPONENT RETAINING PART_USES.
   ```

Figure 15–20: COMPONENT Is Connected to Both Set Types

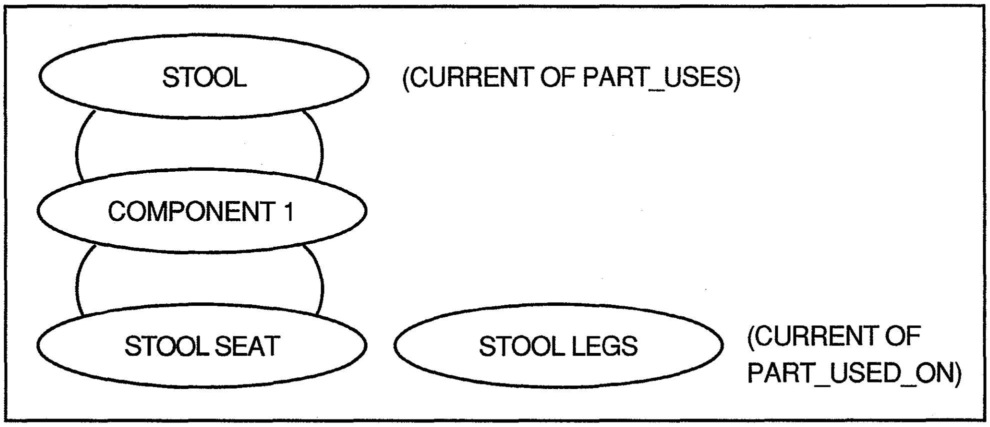


ZK–1497–GE

4. Use the FIND statement to locate the stool legs, again retaining PART_USES currency, thus keeping STOOL current of PART_USES.

```
400-FIND-STOOL-LEGS.
    MOVE "STOOL LEGS" TO PART_DESC.
    FIND FIRST PART USING PART_DESC
               RETAINING PART_USES.
```

Figure 15–21: Finding the Stool Legs While Keeping STOOL Current of PART_USES

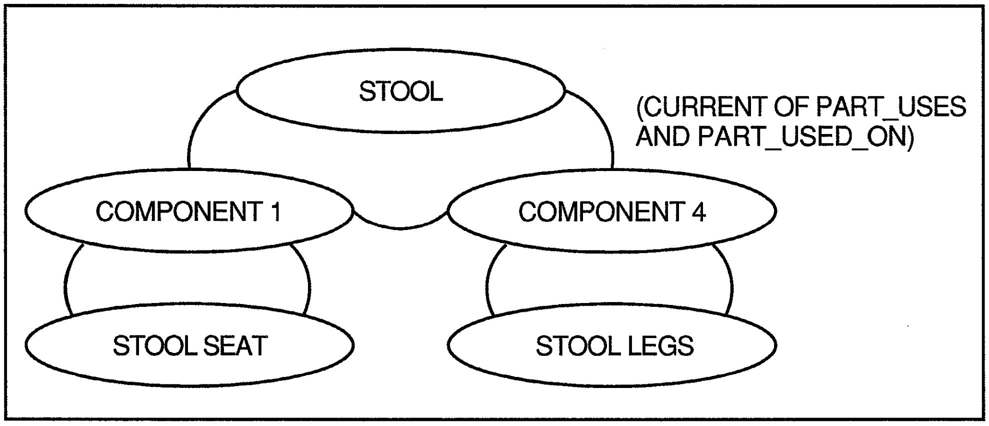


ZK–1498–GE

5. Build a second COMPONENT record (component 4) and store it. This links both PART_USES owner STOOL and PART_USED_ON owner STOOL LEGS to component 4. This completes all the necessary relationships you need to create the bill of materials shown in Figure 15–22.

```
500-CONNECT-COMPONENT-4.
    MOVE 4 TO COMP_QUANTITY.
    STORE COMPONENT.
```

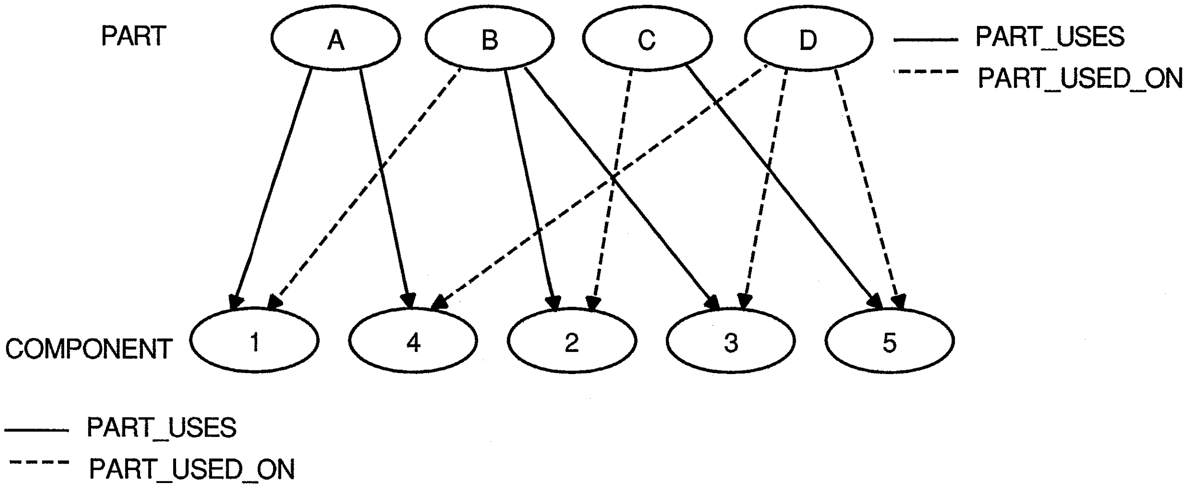
Figure 15–22: Completed Bill of Materials



ZK-1499-GE

Figure 15–23 shows the relationship between PART records and COMPONENT records. The solid lines connect PART_USES owners to their members and the dotted lines connect PART_USED_ON owners to their members.

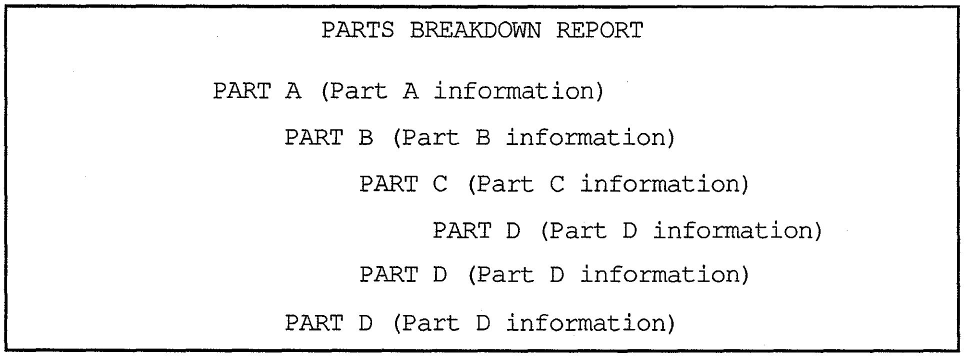
Figure 15–23: Occurrence Diagram of a Many-to-Many Relationship Between Records of the Same Type



ZK-1500-GE

The STOOL program in Example 15–19 loads and connects the parts for the STOOL bill of materials presented earlier in this section. It uses the relationship represented in Figure 15–17 to print its parts breakdown report in Section 15.30.6. Figure 15–24 explains how to read the parts breakdown report.

Figure 15–24: Sample Parts Breakdown Report



```
                 PARTS BREAKDOWN REPORT

           PART A (Part A information)

               PART B (Part B information)

                   PART C (Part C information)

                       PART D (Part D information)

                   PART D (Part D information)

               PART D (Part D information)
```

ZK–6062–GE

The sample parts breakdown report shows that:

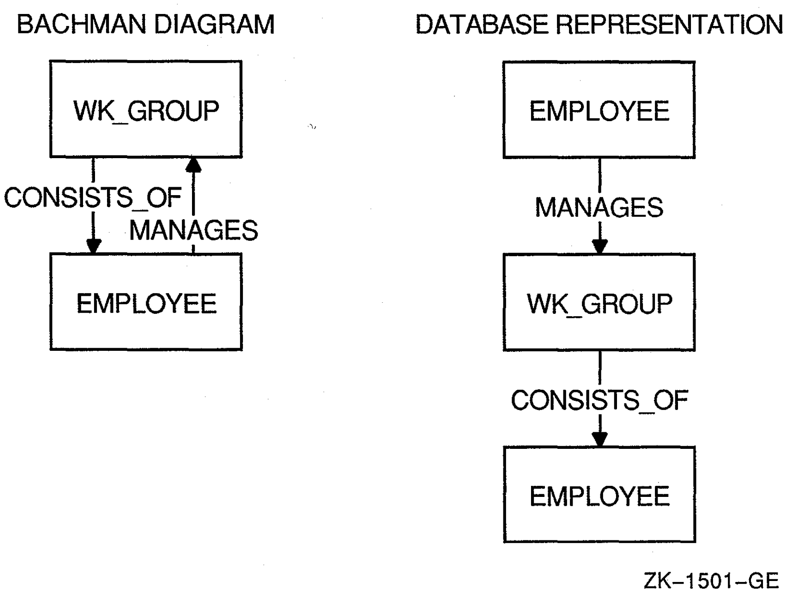
- PART A is built using two subassemblies: PART B and PART D.
- PART B is built using PART C and PART D.
- PART C is built using PART D.

15.18.2.3 One-to-Many Relationships Between Records of the Same Type

To build a one-to-many relationship between records of the same type, the DBA uses junction records. In a one-to-many relationship between records of the same type, either record type can be the junction record. However, in Figure 15–25 the WK_GROUP record type serves as the junction record because the EMPLOYEE record type has most of the relationship's data.

The record type EMPLOYEE includes all employees—supervisors, managers, and so forth. A manager can have many supervisors and a supervisor can have many employees. Conversely, an employee can have only one supervisor, and a supervisor can have only one manager.

Figure 15–25: One-to-Many Relationship Between Records of the Same Type



To show a relationship between employees (that is, who works for whom), Figure 15–25 uses the record type WK_GROUP as a link to establish an owner-to-member relationship. For example, a manager or supervisor would own a WK_GROUP record occurrence in the MANAGES set, and the same WK_GROUP occurrence owns any number of EMPLOYEE records in the CONSISTS_OF set. The relationship would be as follows: one occurrence of EMPLOYEE owns a WK_GROUP record occurrence, which in turn owns zero or more occurrences of the EMPLOYEE record type.

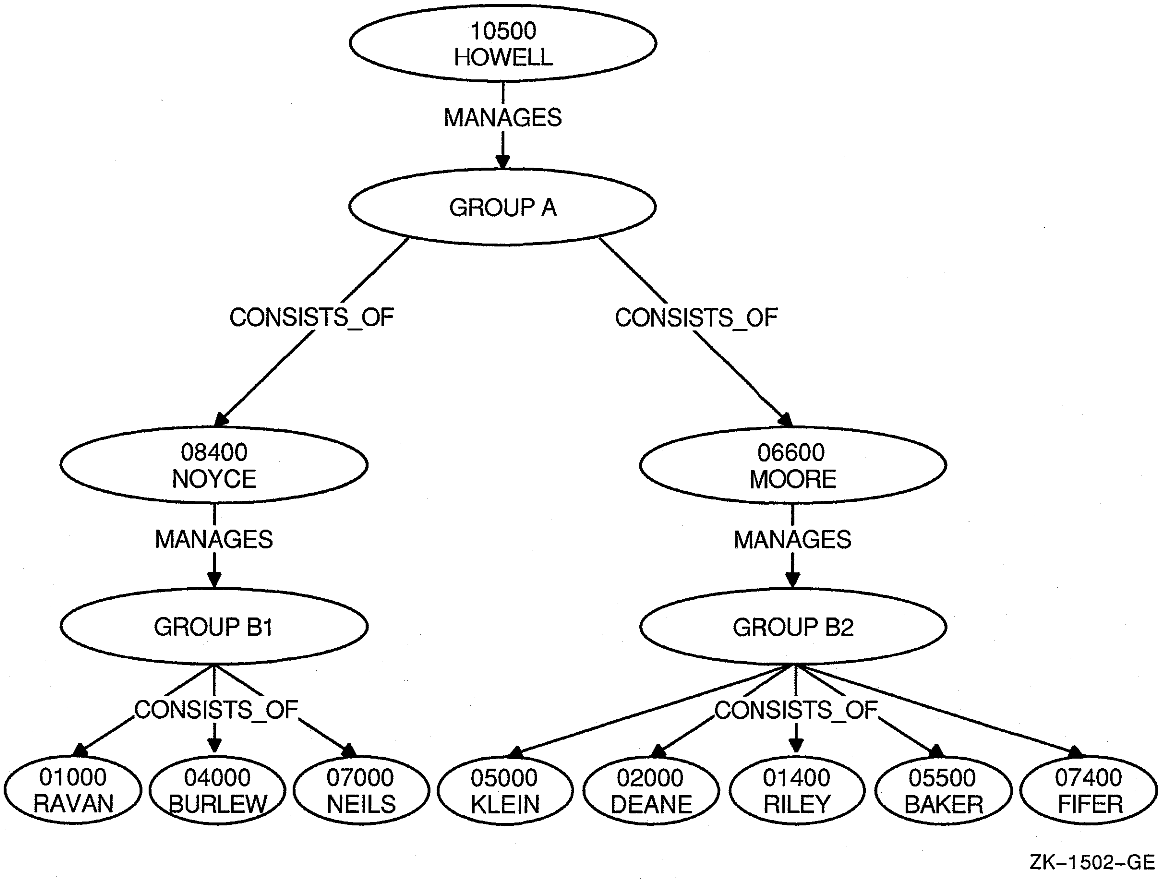
A one-to-many relationship between records of the same type is different from a many-to-many relationship between records of the same type because:

- An employee can have only one manager, while a part can be used on many subassemblies.
- The EMPLOYEE record type can participate both as an owner and a member in its relationship with WK_GROUP.
- The PART record type can participate only as an owner in its relationship with COMPONENT.

Example 15–20 shows how to use DML for hierarchical relationships. The example uses the diagram in Figure 15–25.

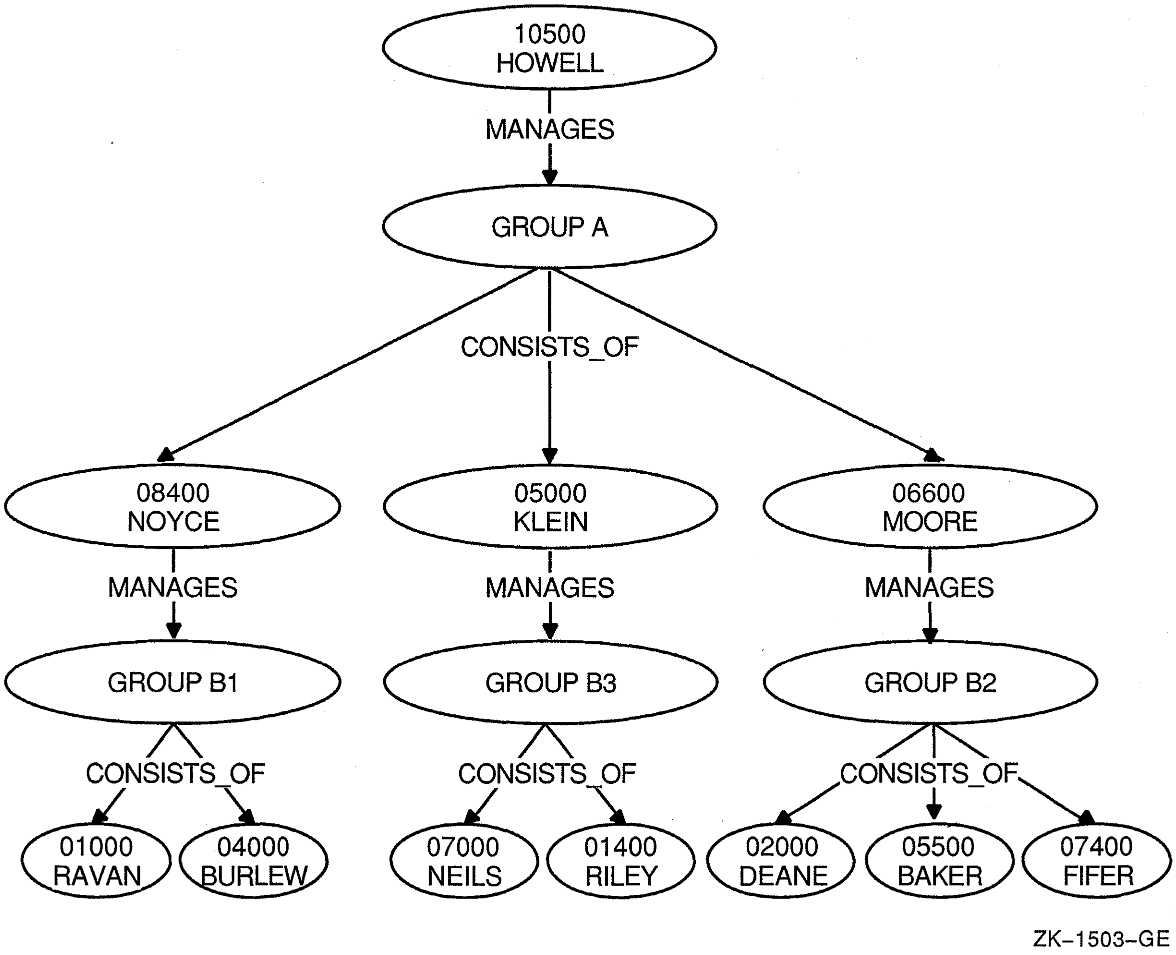
The data in Figure 15–26 shows sample EMPLOYEE records and the connecting WK_GROUP links (Groups A, B1, and B2). For example, employee Howell manages a group that consists of employees Noyce and Moore.

Figure 15–26: Sample Data Prior to Update



Assume that employee Klein is promoted to supervisor with Neils and Riley reassigned to work for him. Figure 15–26 shows the relationship between EMPLOYEE and WK_GROUP record types prior to the update, and Figure 15–27 shows the relationship after the update.

Figure 15–27: Sample Data After Update



Example 15–20 (PERSONNEL-UPDATE program) uses the data in Figure 15–26 and shows you how to:

1. Load the database (PERSONNEL-UPDATE).
2. Display the contents of the database on your terminal using the Report Writer before changing relationships (PERSONNEL-REPORT) (see Figure 15–26 and Example 15–21).
3. Create new relationships (PROMOTION-UPDATE).
4. Display the contents of the database on your terminal using the Report Writer after changing relationships (PERSONNEL-REPORT) (see Figure 15–27 and Example 15–22).

15.19 Areas

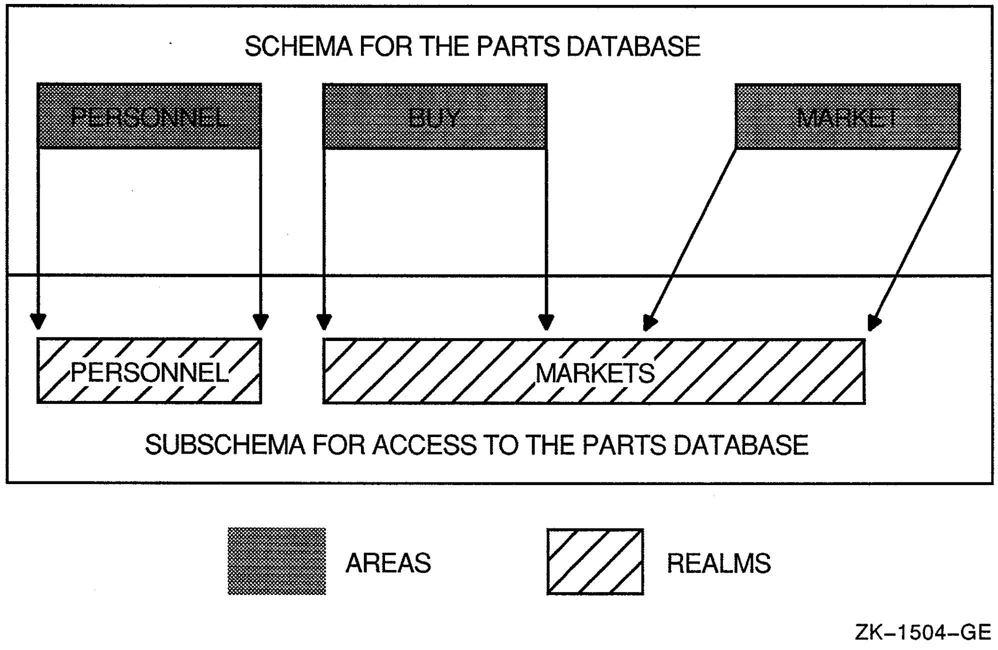
The DBA divides the database into areas so you can reference the database in sections instead of an entire unit. Areas are physical divisions of the database that are defined in the schema and are used to dump selectively, verify, or recover sections of the database; improve I/O; group logically related record types; and provide protection restrictions. Areas are stored as separate files and can be on separate volumes.

15.20 Realms

A realm is a group of one or more areas. Realms are logical divisions of the database. A realm is the object of the DML READY statement. Figure 15–28 shows the relationship between the schema, areas, subschema, and realms. Even though realms can contain data from more than one area, the type of data they contain is dependent on the subschema. It acts as a filter, allowing access to only specific data items.

Entire realms, as well as individual database records, are locked by the DBCS as they are retrieved by the run unit, and the degree of locking depends on the specific DML command used. For more information, see Section 15.24.1.

Figure 15–28: Database Relationships


SCHEMA FOR THE PARTS DATABASE
PERSONNEL
BUY
MARKET
PERSONNEL
MARKETS
SUBSCHEMA FOR ACCESS TO THE PARTS DATABASE
AREAS
REALMS
ZK-1504-GE

15.21 Run Unit

The term run unit and program are not the same. A run unit is an executable image that may access a database, while a program can be used in two or more run units. For example, program SHOW-EMPLOYEE can be run simultaneously by a payroll department employee to obtain employee data, and by an accountant to obtain job cost data. Each person controls his or her own run unit.

15.22 Currency Indicators

When you access database records, the database control system (DBCS) uses pointers called currency indicators to keep track of record storage and retrieval. VAX COBOL uses currency indicators to remember records and their positions in the database. Currency indicators can be changed by DML statement execution. Thus, they assist in defining the environment of a DML statement and are updated as a result of executing DML statements.

One currency indicator exists for each realm, set type, and record type defined in your subschema. Another currency indicator, called the run-unit currency indicator, also exists for the run unit.

All the currency indicators in a run unit are null prior to execution of the first DML statement. The null value indicates there is neither a current record nor a current position. Execution of certain DML statements can change the value of currency indicators. However, currency indicators do not change if statement execution fails.

The DBCS also uses currency indicators as place markers to control its sequence of access to the database. For example, if VENDOR is the name of the vendor records in Figure 15–3, then the current of VENDOR is normally the vendor record most recently accessed. Likewise, in the set VENDOR_SUPPLY, the current of VENDOR_SUPPLY is normally the most recently accessed record of that set. Note that current of set could be either a member or owner record because both record types are part of the VENDOR_SUPPLY set.

Failure to establish correct currency can produce incorrect or unpredictable results. For example, you might unknowingly modify or delete the wrong record. The following sections describe how the DBCS sets currency indicators and how to use currency status in a DML program.

15.22.1 Current of Realm

Each realm currency indicator can be null or it can identify:

- A record and its position in the realm
- A position in the realm but not a specific record

A record identified by the realm currency is called current of realm. The DBCS updates current of realm only when you reference a different record within the realm. For example:

```
000100 PROCEDURE DIVISION.
000110     .
000120     .
000130     .
000500    FIND FIRST PART WITHIN BUY.
000510    FIND FIRST PART WITHIN MAKE.
000520    FIND NEXT PART WITHIN BUY.
000600    FIND NEXT SUPPLY WITHIN PART_INFO.
000610    .
000620    .
000630    .
```

For example, if LABEL and CASSETTE are in the BUY realm, while TAPE is in the MAKE realm, statement 000500 sets the first occurrence of PART record in realm BUY (LABEL) as current of realm BUY. Statement 000510 sets the first occurrence of PART record in realm MAKE (TAPE) as current of realm MAKE. Notice that current of realm BUY is still the record occurrence accessed in statement 000500. Statement 000520 changes the current of realm BUY to the next occurrence PART record in realm BUY (CASSETTE). Current of realm MAKE remains the record accessed in statement 000510. Because the SUPPLY record type is located in the MARKET realm, statement 000600 sets the current of MARKET realm to the first SUPPLY record in the current PART_INFO set.

15.22.2 Current of Set Type

Each set type currency indicator can be null or it can identify:

- A record and its position in the set type
- A position in the set type but not a record

A record identified by a set type currency indicator is the current record for the set type, or current of set type.

If the ordering criterion for a set type is NEXT or PRIOR, the set type's currency indicator specifies the insertion point for member records. Therefore, if the currency indicator points to an empty position, a member record can be inserted in the specified position. If the currency indicator points to a record and NEXT is specified, a member record can be inserted after the current record for the set type. If the currency indicator points to a record and PRIOR is specified, a member record can be inserted before the current record for the set type.

The DBCS updates current of set type only when you reference a record that participates either as an owner or member in a set type occurrence. For example:

```
000100 PROCEDURE DIVISION.
            .
            .
            .
000500      FIND FIRST PART.
000510      FIND FIRST SUPPLY WITHIN PART_INFO.
000520      FIND OWNER WITHIN VENDOR_SUPPLY.
000600      .
            .
            .
```

Statement 000500 sets the first occurrence of PART (LABEL) as current of set types PART_USES, PART_USED_ON, and PART_INFO. This is because PART records participate in three sets (see Figure 15–3). Because LABEL is current of PART_INFO, statement 000510 sets the first occurrence of SUPPLY (4-DAYS) owned by LABEL as current of set type PART_INFO. Because SUPPLY also participates in the VENDOR_SUPPLY set, this statement also sets the current occurrence of SUPPLY as current of set type VENDOR_SUPPLY. Statement 000520 sets the VENDOR owner record occurrence (SOUND-OFF CO.), which owns the current SUPPLY record, as current of set type VENDOR_SUPPLY.

15.22.3 Current of Record Type

Each record type currency indicator can be null or it can identify:

- A record and its position among other records of the same type
- A position among records of the same type, but not identify a record

Record type currency indicators do not identify a record type's relationship with other record types.

A record identified by a record type currency indicator is called current of record type. The DBCS updates the current of record type only when you reference a different record occurrence of the record type. References to other record types do not affect this currency. For example:

```
000100 PROCEDURE DIVISION
            .
            .
            .
000500      FIND LAST PART.
000510      FIND FIRST SUPPLY WITHIN PART_INFO.
000520      FIND NEXT WITHIN PART_INFO.
000530      FIND FIRST VENDOR.
            .
            .
            .
```

Statement 000500 sets the last occurrence of PART (TAPE) as current of record type PART. Statement 000510 sets the SUPPLY record occurrence (2-DAYS) as current of record type SUPPLY. Statement 000520 updates current of record type for SUPPLY to record occurrence (1-WEEK). Statement 000530 sets VENDOR record occurrence (MUSICO INC.) as current of record type VENDOR.

15.22.4 Current of Run Unit

The Database Control System (DBCS) updates the currency indicator for current of run unit each time a run unit refers to a different record occurrence, regardless of realm, set, or record type. For example:

```
000100 PROCEDURE DIVISION.
            .
            .
            .
000500      FIND FIRST PART.
000510      FIND FIRST SUPPLY.
000520      FIND FIRST VENDOR.
000600      .
000610      .
000620      .
```

Statement 000500 sets the current of run unit to the first PART record occurrence (LABEL). Statement 000510 then sets the first SUPPLY record occurrence (4-DAYS) as current of run unit. Finally, statement 000520 sets the first VENDOR record (MUSICO INC.) as current of run unit. The first VENDOR record occurrence remains current of run unit until the run unit refers to another record occurrence.

15.23 Currency Indicators in a VAX COBOL DML Program

Currency indicators are the tools you use to navigate through a database. Because of the many set relationships a database can contain, touching a record with a DML statement often changes more than one currency indicator. For example, a FETCH to a set type record can change currency for the set type, the record type, the realm, and the run unit. Knowing currency indicator status, how currency indicators change, and what statements control them, will help you locate the correct data.

Example 15–3 searches for TAPE vendors with a supply rating equal to A. Assume that record TAPE resides in BUY realm and that the SUPPLY record occurrences 2-DAYS and 5-DAYS have a SUP_RATING equal to A. Figure 15–29 shows how DML statements affect currency status.

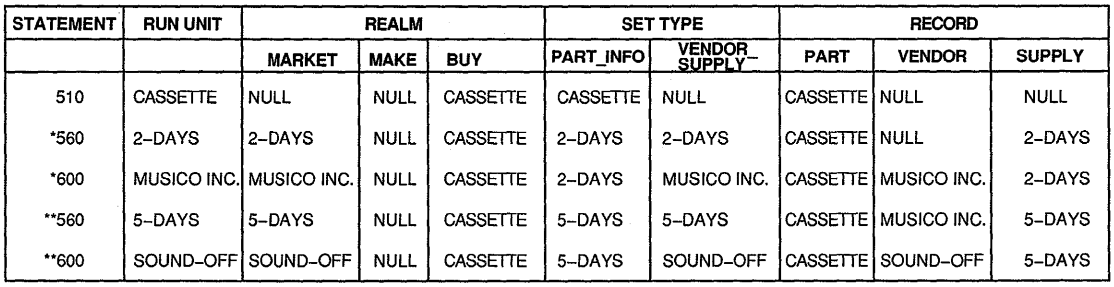
Example 15–3: Currency Indicators

```
000100 PROCEDURE DIVISION
           .
           .
           .
000490 100-FETCH-THE-PART.
000500     MOVE "TAPE" TO PART_DESC.
000510     FETCH FIRST PART USING PART_DESC.
000520     MOVE "A" TO SUP_RATING.
000550 200-FIND-SUPPLY.
000560     FIND NEXT SUPPLY WITHIN PART_INFO
000570                USING SUP_RATING.
000580          AT END
000590          GO TO 500-NO-MORE-SUPPLY.
000600     FETCH OWNER WITHIN VENDOR_SUPPLY.
000610     ***********************
000620     * VENDOR PRINT ROUTINE *
000630     ***********************
000640    GO TO 200-FIND-SUPPLY.
```

Statement 000500 provides the search argument used by statement 000510. Statement 000510 fetches the first occurrence of PART with a PART_DESC equal to TAPE. Statement 000520 provides the search argument used by statement 000560. Statement 000560 finds each member record occurrence of SUPPLY with a SUP_RATING equal to A owned by the PART with a PART_DESC equal to TAPE.

If, instead of its present structure, statement 000560 read “FIND NEXT SUPPLY USING SUP_RATING,” the search for supply records would not be restricted to supply member records in the PART_INFO set owned by TAPE. Instead, the search would extend to all supply records, finding all vendors with a supply rating equal to A, who may or may not be suppliers of TAPE.

Figure 15–29: Currency Status by Executable DML Statement



STATEMENT	RUN UNIT	REALM			SET TYPE		RECORD		
		MARKET	MAKE	BUY	PART_INFO	VENDOR_SUPPLY	PART	VENDOR	SUPPLY
510	CASSETTE	NULL	NULL	CASSETTE	CASSETTE	NULL	CASSETTE	NULL	NULL
*560	2–DAYS	2–DAYS	NULL	CASSETTE	2–DAYS	2–DAYS	CASSETTE	NULL	2–DAYS
*600	MUSICO INC.	MUSICO INC.	NULL	CASSETTE	2–DAYS	MUSICO INC.	CASSETTE	MUSICO INC.	2–DAYS
**560	5–DAYS	5–DAYS	NULL	CASSETTE	5–DAYS	5–DAYS	CASSETTE	MUSICO INC.	5–DAYS
**600	SOUND–OFF	SOUND–OFF	NULL	CASSETTE	5–DAYS	SOUND–OFF	CASSETTE	SOUND–OFF	5–DAYS

* First execution
** Second execution

ZK–1505–GE

15.23.1 Using the RETAINING Clause

You use the RETAINING clause to save a currency indicator you want to refer to. You use the RETAINING clause to: (1) navigate through the database and return to your original starting point, or (2) walk through a set type. (The expression "walk through a set type" implies a procedure where you access all owner records and their respective members.) Refer to the *VAX COBOL Reference Manual* for further information.

After finding all members for an owner, the current of run unit is the last accessed member record occurrence in the set. If the next statement is a FIND NEXT for an owner, you may not retrieve the next owner. This is because:

- Current of set type (in this case, the last member record occurrence) is also current of run unit.
- Without a WITHIN clause, the FIND (or FETCH) is based on current of run unit.

Because DBCS uses currency status as pointers, a FIND NEXT VENDOR WITHIN MARKET uses current of MARKET realm to find the next owner record occurrence. To make sure a FIND (or FETCH) next owner statement finds the next logical owner record, use the RETAINING clause, as shown in Example 15–4.

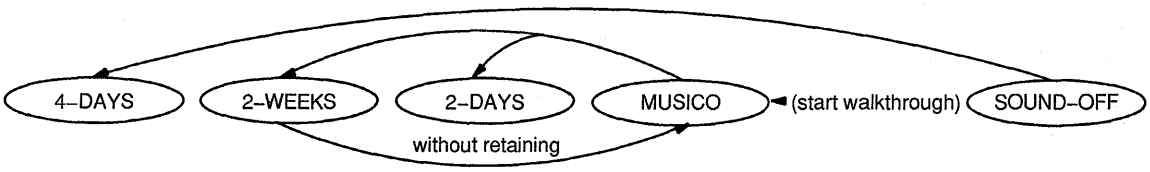
Example 15–4: Using the RETAINING Clause

```
000100 PROCEDURE DIVISION.
           .
           .
           .
000400 100-VENDOR-SUPPLY-WALKTHRU.
000410     FETCH NEXT VENDOR WITHIN MARKET
000420       AT END GO TO 900-ALL-DONE.
           .
           .
           .
        ***********************
        * VENDOR PRINT ROUTINE *
        ***********************
           .
           .
           .
000500 300-GET-VENDORS-SUPPLY.
000510     FETCH NEXT SUPPLY WITHIN VENDOR_SUPPLY
000520        RETAINING REALM
000530        AT END
000540        GO TO 100-VENDOR-SUPPLY-WALKTHRU.
              .
              .
              .
        ***********************
        * SUPPLY PRINT ROUTINE *
        ***********************
              .
              .
              .
000550     GO TO 300-GET-VENDORS-SUPPLY.
```

Statement 000410 fetches the vendors. Statement 000510 fetches the supply records owned by their respective vendors. Statement 000510 also uses the RETAINING clause to save the realm currency.

A FETCH NEXT SUPPLY (statement 000510) without the RETAINING clause makes SUPPLY current for the run unit, its record type, all sets in which it participates, and its realm. When SUPPLY record 2-WEEKS in Figure 15–30 is current of run unit, a FETCH NEXT VENDOR statement fetches the vendor whose physical location in the database follows the 2-WEEKS record. As shown in Figure 15–30, MUSICO would be the next vendor and the program would be in an infinite loop.

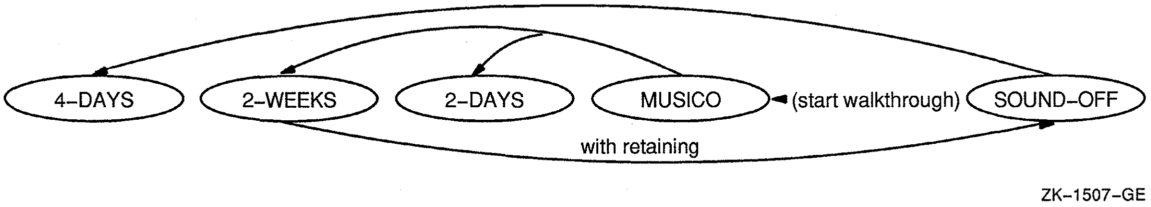
Figure 15–30: Physical Representation of a Realm Without a RETAINING Clause



ZK–1389–GE

A FETCH NEXT SUPPLY with the RETAINING clause makes SUPPLY current for the run unit and the set types but keeps the vendor record current for the realm shown in Figure 15–31. By retaining the realm currency when you fetch supply records, the last accessed vendor record remains current of realm. A FETCH NEXT VENDOR WITHIN MARKET statement uses the realm currency pointer, which points to MUSICO to fetch the next vendor, SOUND-OFF. Therefore, retaining the realm currency allows you to fetch the next logical vendor record.

Figure 15–31: Physical Representation of a Realm with a RETAINING Clause



15.23.2 Using Keeplists

A keeplist is a stack of database key values (see the description of KEEPLIST in the *VAX COBOL Reference Manual*). The KEEP and FIND ALL statements build a stack of keys that lets you retrieve DBMS records using the ordinal position of the stack entries. DBMS calls the table of entries a keeplist. Each execution of the KEEP or FIND ALL statement adds a record's database key (dbkey) value to the end of a keeplist and places a retrieval lock on the record. Therefore, other users cannot change a record while its database key is in your keeplist.

You can use a keeplist to retain the database key of a record after that record is no longer current. That is, by inserting a database key into a keeplist, you can continue to reference that record by specifying the keeplist name and database key value in your DML statement. This is especially useful when you want to remember a record during a long sequence of DML commands that affect currency, or when you want to remember a list of records.

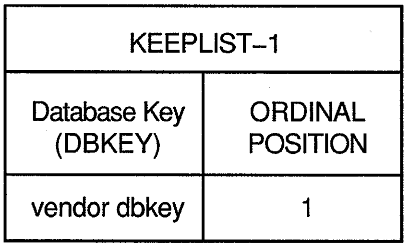
A keeplist can contain zero, one, or several database key values. To activate a keeplist, use the KEEP statement. To empty a keeplist, use the FREE statement. All keeplists are deallocated when you execute a COMMIT or ROLLBACK unless COMMIT RETAINING is used.

The following example adds database keys to a keeplist.

```
000100 PROCEDURE DIVISION.
           .
           .
           .
000140 100-KEEPLIST-EXAMPLE.
000150     FETCH FIRST VENDOR.
000160     KEEP CURRENT USING KEEPLIST-1.
000170     FETCH FIRST SUPPLY WITHIN VENDOR_SUPPLY.
000180     FETCH OWNER WITHIN PART_INFO.
000190     IF PART_STATUS = "M"
000200        KEEP CURRENT WITHIN VENDOR_SUPPLY USING KEEPLIST-1.
```

Statement 000160 adds the vendor record's dbkey value (the current of run unit) to KEEPLIST-1. Figure 15–32 shows the contents of KEEPLIST-1 after execution of statement 000160. Adding a record's database key to a keeplist also prevents record updating by other concurrent users. Statements 000190 and 000200 add a supply record's database key to KEEPLIST-1 whenever its PART_INFO owner has a status of M. Figure 15–33 shows the contents of KEEPLIST-1 after the execution of statements 000190 and 000200.

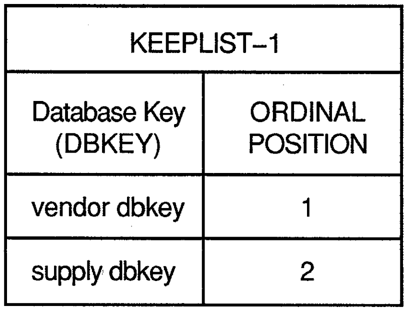
Figure 15–32: State of KEEPLIST-1 After Executing Line 000160



KEEPLIST-1	
Database Key (DBKEY)	ORDINAL POSITION
vendor dbkey	1

ZK-6063-GE

Figure 15–33: State of KEEPLIST-1 After Executing Lines 000190 and 000200



KEEPLIST-1	
Database Key (DBKEY)	ORDINAL POSITION
vendor dbkey	1
supply dbkey	2

ZK-6064-GE

You can use database key values as search arguments to locate database records. For example:

```
FIND 2 WITHIN KEEPLIST-1
```

This statement:

- Uses the value of the number 2 to locate the ordinal position of a database key value
- Uses the database key value to find a record

The KEEP statement can also transfer database key values from one keeplist to another. For example:

```
KEEP OFFSET 2 WITHIN KEEPLIST-1 USING KEEPLIST-2
```

This statement copies the second-positioned database key value in KEEPLIST-1 to the end of KEEPLIST-2.

The FREE statement removes database key value entries from a keeplist. For example:

```
FREE ALL FROM KEEPLIST-1
```

This statement removes all the entries from KEEPLIST-1.

You can remove keeplist entries by identifying their ordinal position within the keeplist. For example:

```
FREE 5 FROM KEEPLIST-2
```

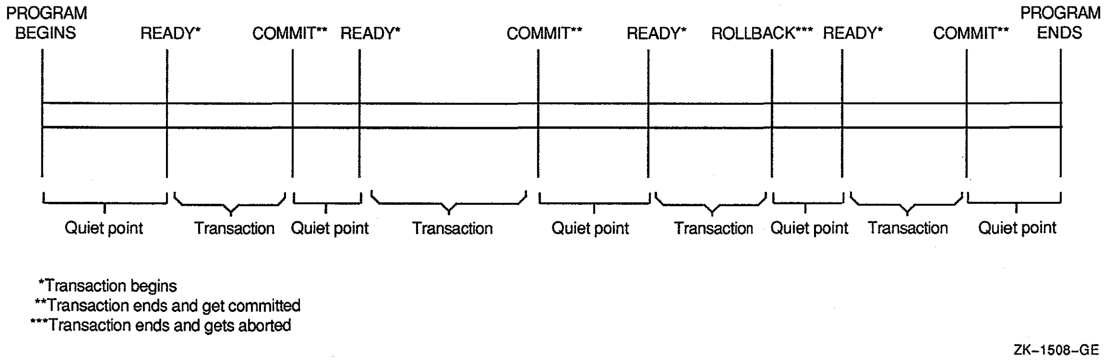
This statement removes the fifth-positioned database key value from KEEPLIST-2. Removing a keeplist entry changes the position of all the following entries. For example, after freeing entry 5, entry 6 becomes the fifth-positioned entry, entry 7 becomes the sixth-positioned entry, and so forth. The FREE statement changes the ordinal position of a database key value in the keeplist, not its contents.

15.23.3 Transactions and Quiet Points

You generally segment your run unit into transactions, bounded instances of run-unit activity. A transaction begins with the first DML statement in the run unit or with a READY statement that follows a COMMIT or ROLLBACK statement; continues through a series of DML data access statements; and ends with either a COMMIT statement, a ROLLBACK statement, or the termination of the run unit. Before the initial READY statement is issued, after the COMMIT or ROLLBACK, and before the next READY, the run unit is at a quiet point. A quiet point is the time that exists between the last executed COMMIT or ROLLBACK statement and the next READY statement, or the time prior to the first executed READY statement.

The Quiet Point—Transaction—Quiet Point continuum provides the DBCS with a structure that allows it to control access to and ensure the integrity of your data. To implement this control, the DBCS uses currency indicators and locking. Figure 15–34 shows the segmentation of a run unit into transactions and quiet points.

Figure 15–34: Transactions and Quiet Points



15.24 VAX COBOL DML Programming—Tips and Techniques

The following sections offer tips and techniques you can use to improve program performance and reduce development and debugging time.

15.24.1 The Ready Modes

Proper use of the READY usage modes can improve system performance.

You inform the DBCS of your record-locking requirements when you issue the READY command. The command takes the form:

```
READY <allow-mode> <access-mode>
```

or

```
READY <access-mode> <allow-mode>
```

The allow- and the access-mode arguments pass your requirements to the DBCS.

The allow-mode object of the READY command indicates what you will allow other run units to do while your run unit works with storage areas within the realm you readied. There are four different allow modes as follows:

CONCURRENT Permits other run units to ready the same realm or realms that contain the same storage areas as the realms your run unit readied. CONCURRENT also allows other run units to perform any DML function on those storage areas, including updates.

PROTECTED Permits other run units to use the same storage areas as your run unit, but does not allow those run units to update records in the storage areas.

EXCLUSIVE Prohibits other run units from even reading records from the restricted storage areas.

BATCH Allows concurrent run units to update the realm. BATCH also allows you to access or update any data in the realm while preventing concurrent run units from accessing or updating the realm.

While the allow mode says what your run unit will allow other run units to do, the access mode says that your run unit will either read or write records (RETRIEVAL or UPDATE).

Because the UPDATE access mode can lock out other users, use it only for applications that perform database updates. If an application accesses the database for inquiries only, use the RETRIEVAL access mode. The RETRIEVAL mode also prevents a run unit from accidentally updating the database.

The combination of the allow mode and the access mode is called the usage mode. There are eight READY usage modes as follows:

- CONCURRENT RETRIEVAL
- CONCURRENT UPDATE
- PROTECTED RETRIEVAL (the system default)
- PROTECTED UPDATE
- EXCLUSIVE RETRIEVAL
- EXCLUSIVE UPDATE
- BATCH RETRIEVAL
- BATCH UPDATE

Use the CONCURRENT usage modes for applications requiring separate run units to simultaneously access the database. They allow other run units to perform a READY statement on your realm, and possibly change or delete the database records in that realm.

Use the PROTECTED usage modes only when unrestricted access might produce incorrect or incomplete results. Protected access prevents other run units from making changes to the data in your realm. However, run units in RETRIEVAL mode can still access (read-only) your realm.

Use the EXCLUSIVE usage modes only when you want to lock out all other users. The EXCLUSIVE mode speeds processing for your run unit and prevents other run units from executing a READY statement on your realm. When you specify EXCLUSIVE access, use only the realms you need. Eliminating the use of unnecessary realms minimizes lockout. Use the EXCLUSIVE allow mode to get the best performance from a single run-unit application. Care must be taken, however, because other run units are locked out and must wait for the exclusive run unit to finish before it can begin operations.

Use the BATCH RETRIEVAL usage mode for concurrent run units to update the realm. Use the BATCH UPDATE usage mode to access or update any data in the realm while preventing concurrent run units from accessing or updating the realm.

For more information on READY usage mode conflicts, see the READY statement in the *VAX COBOL Reference Manual*. It summarizes the effects of usage mode options on run units readying the same realms.

15.24.1.1 Record Locking

Concurrent run units can reference realms that map to the same storage area; the same records can be requested by more than one transaction at the same time. If two different transactions were allowed to modify the same data, that data would be rendered invalid. Each modification to the original data would be made in ignorance of other modifications, and with unpredictable results. VAX DBMS preserves the integrity of data shared by multiple transactions. It also provides levels and degrees of record locking. You can control access to, or lock:

- All records in a realm you intend to access

- Individual records as they are retrieved by DML statements

You can also lock records totally or allow some retrieval functions.

Record locking begins with the execution of the first READY statement in the run unit. At that time the DBCS is told of your storage area locking requirements. If you specify EXCLUSIVE allow mode, no other run unit is allowed to access records in the specified realms. This is all the locking that the DBCS need do. If you specify CONCURRENT or PROTECTED modes, the DBCS initiates locking at the record level.

Individual records are locked as they are retrieved by the run unit. The degree of locking depends on the specific DML command used. For example, if your run unit executes a FETCH or FIND statement, the DBCS sets a read-only record lock, allowing other run units to read, but not update, the records. This lock is also set if your run unit assigns the database key associated with the record to a keeplist with the KEEP verb. (Note if you use FETCH or FIND FOR UPDATE, a no-read lock is placed on the specified record.)

As a record is retrieved, the lock is held at this level until there are no more currency indicators pointing to the record. If the program assigns a record to a keeplist, the lock is held by your run unit until it frees the record from the keeplist with a FREE statement. However, if a currency indicator points to a record whose database key is also in a keeplist, then a FREE statement to that keeplist entry still leaves the read-only lock active for that record. Similarly, if the same database key is in several keeplists, then freeing it from one keeplist does not release the other read-only locks.

However, the DBCS grants a no-read access lock if your run unit specifies a DML update verb, such as STORE, CONNECT, or MODIFY. Your run unit retains the lock on this record until the change is committed to the database by the DML COMMIT verb or the change is terminated or canceled by ROLLBACK.

The Run-Time System notifies the DBCS each time a run unit requests a locked record, thus keeping track of which records are locked and who is waiting for which records. This logging helps the DBCS determine whether a conflict exists, such as multiple run units requesting, but not being allowed, to access or change the same record. For more information on record locking, refer to the VAX DBMS documentation on database design and programming.

15.24.2 COMMIT and ROLLBACK

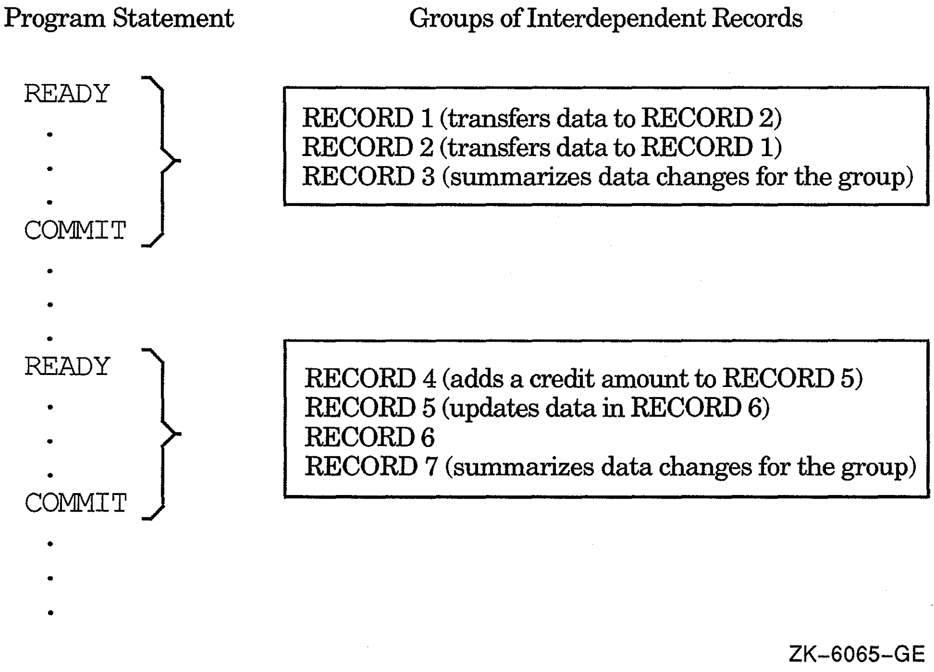
When you are in CONCURRENT UPDATE mode, any changes made to a record lock the record and prevent its access by other run units. For example, if a program updates 200 customer records in one transaction, the 200 customer records are unavailable to other run units. To minimize lockout, use the COMMIT statement as often as possible.

The COMMIT statement makes permanent all changes made to the database, frees all locks, and nulls all currencies. It also establishes a quiet point for your run unit.

The RETAINING clause can be used with the COMMIT statement. COMMIT RETAINING does not empty keeplists; retains all currency indicators; does not release realm locks; demotes no-read locks to read-only locks; then releases locks for all records except those in currency indicators or keeplists and makes visible any changes made to the database.

To use COMMIT properly, you need to know about application systems. For example, you might want to execute a COMMIT each time you accomplish a logical unit of work. Or, if you were updating groups of interdependent records like those in Figure 15–35, you would execute a COMMIT only after updating a record group.

Figure 15–35: Using the COMMIT Statement


Program Statement
Groups of Interdependent Records
READY
COMMIT
RECORD 1 (transfers data to RECORD 2)
RECORD 2 (transfers data to RECORD 1)
RECORD 3 (summarizes data changes for the group)
READY
COMMIT
RECORD 4 (adds a credit amount to RECORD 5)
RECORD 5 (updates data in RECORD 6)
RECORD 6
RECORD 7 (summarizes data changes for the group)
ZK–6065–GE

The ROLLBACK statement cancels all changes made to the database since the last executed READY statement and returns the database to its condition at the last quiet point. The DBCS performs an automatic ROLLBACK if your run unit ends without executing a COMMIT or if it ends abnormally.

In Example 15–5 an order-processing application totals all items ordered by a customer. If the order amount exceeds the credit limit, the program executes a ROLLBACK and cancels the transaction updates. Notice that the credit limit is tested for each ordered item, thus avoiding printing of an entire invoice prior to cancelling the order.

Example 15–5: ROLLBACK Statement

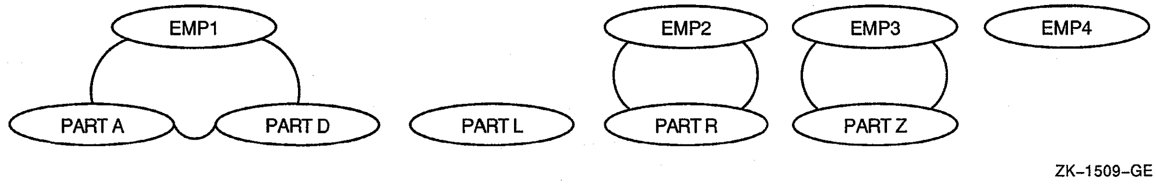
```
        .
        .
        .
READY-UPDATE.
   READY TEST_REALM CONCURRENT UPDATE.
 *************************
 * FETCH CUSTOMER ROUTINE *
 *************************
   .
   .
   .
 ******************************
 * FETCH ORDERED ITEMS ROUTINE *
 ******************************
   .
   .
   .
CREDIT-LIMIT-CHECK.
    MULTIPLY ORDERED-QUANTITY BY UNIT-PRICE
             GIVING ORDER-AMOUNT.
    ADD ORDER-AMOUNT TO TOTAL-AMT.
    IF TOTAL-AMT IS GREATER THAN CUST-CREDIT-LIMIT
            ROLLBACK
            PERFORM CREDIT-LIMIT-EXCEEDED
       ELSE PERFORM PRINT-INVOICE-LINE.
```

15.24.3 The Owner and Member Test Condition

The FIND OWNER statement finds the owner of the current of set type, which may not be the same as the current of run unit. Thus, executing a FIND OWNER WITHIN set-name when the current of run unit record is not connected to the specified set returns the owner of the member that is current of set type.

Figure 15–36 shows occurrences of the RESPONSIBLE_FOR set type where employees are responsible for the design of certain parts.

Figure 15–36: Occurrences of the RESPONSIBLE_FOR Set Type



Example 15–6 uses the data in Figure 15–36 to perform an analysis of PART D, PART L, and the work of the engineer responsible for each part. The set retention class is optional.

Example 15–6: Owner and Member Test Condition

```
        .
        .
        .
000130 MAIL-LINE ROUTINE.
000140     MOVE "PART D" TO PART_DESC.
000150     PERFORM FIND-PARTS.
000160     MOVE "PART L" TO PART_DESC.
000170     PERFORM FIND-PARTS.
000180     GO TO ALL-FINISHED.
000190 FIND-PARTS.
000200     FIND FIRST PART USING PART_DESC.
000210     IF PART-IS-MISSING
000220              PERFORM PART-MISSING.
000230     PERFORM PARTS-ANALYSIS.
000240     FIND OWNER WITHIN RESPONSIBLE_FOR.
000250     PERFORM WORKLOAD-ANALYSIS.
000250 DONE-ANALYSIS.
000260     EXIT.
        .
        .
        .
```

When PART L becomes current of run unit, a FIND OWNER (statement 000240) finds PART D's owner, thus producing incorrect results. This is because a FIND OWNER WITHIN set-name uses the current of set type and PART L is not a member of any RESPONSIBLE_FOR set type occurrence. To prevent this error, statement 000240 should read:

```
IF RESPONSIBLE_FOR MEMBER
        FIND OWNER WITHIN RESPONSIBLE_FOR
   ELSE
        PERFORM PART-HAS-NO-OWNER.
```

15.24.4 Using IF EMPTY Instead of IF OWNER

The OWNER test condition does not test whether the current record owns any member records. Rather, this condition tests if the current record participates as an owner record. If a record type is declared as the owner of a set type, an OWNER test for that record type will always be true. Therefore, referring to Figure 15–36, if EMP4 is the object of an IF RESPONSIBLE_FOR OWNER test, the result is true because EMP4 is an owner record, even though the set occurrence is empty.

To test if an owner record owns any members, use the EMPTY test condition. For example:

```
IF RESPONSIBLE_FOR IS EMPTY PERFORM EMPTY-ROUTINE
        ELSE ...
```

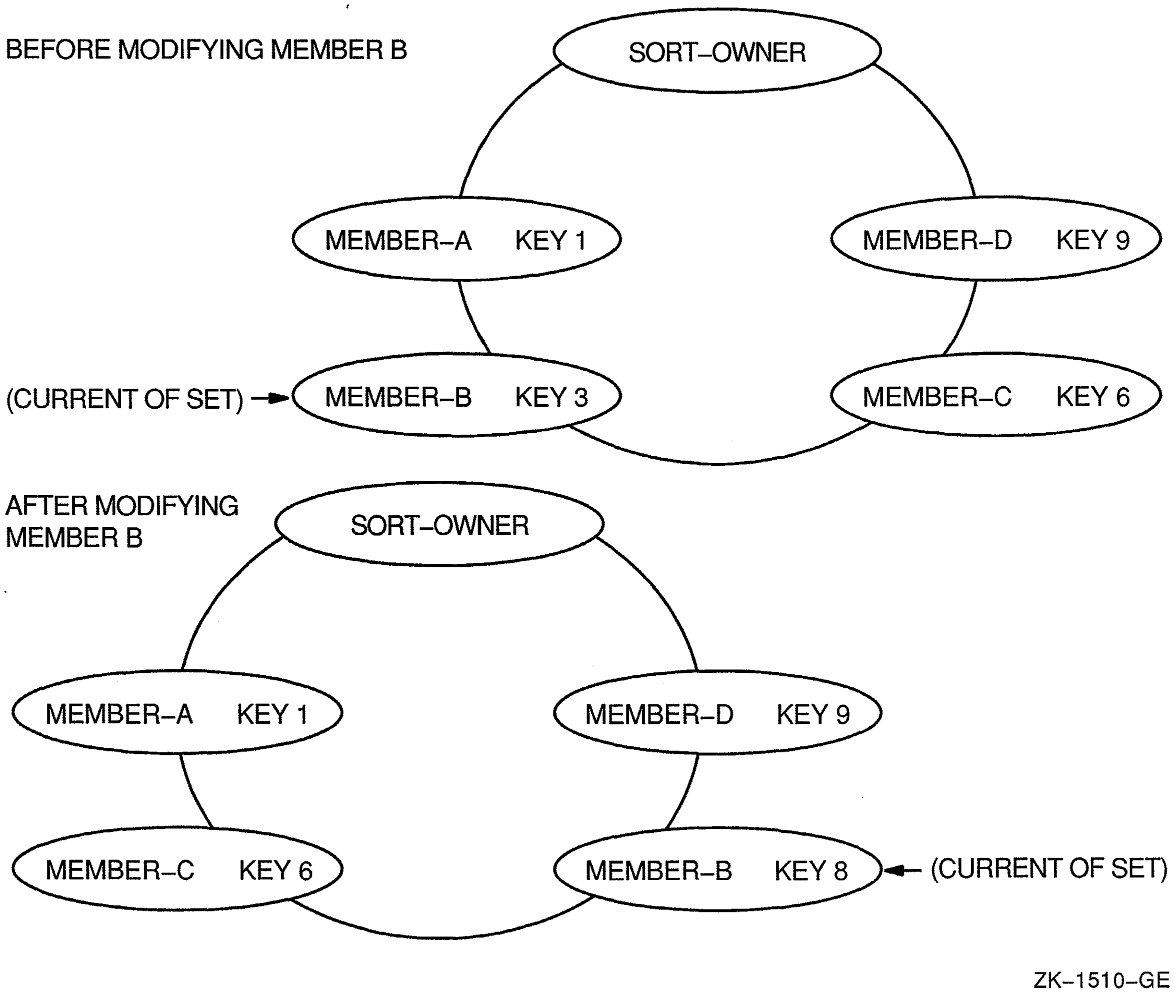
Thus, if EMP4 is the object of an IF RESPONSIBLE_FOR IS EMPTY test, the result is true because the set occurrence has no members.

15.24.5 Modifying Members of Sorted Sets

If the schema defines a set's order to be SORTED and you modify any data items specified in the ORDER IS clause of the schema, the record may change position within the set occurrence. If the record does change position, the set's currency changes to point to the member record's new position.

Figure 15–37 shows a set occurrence for SORT_SET where MEMBER-B's key (KEY 3) was changed to KEY 8. Before altering the record's key, the set currency pointed to MEMBER-B, and a FETCH NEXT MEMBER WITHIN SORT_SET fetched MEMBER-C. However, the modification to MEMBER-B's key repositions the record within the set occurrence. Now, a FETCH NEXT MEMBER WITHIN SORT_SET fetches the MEMBER-D record.

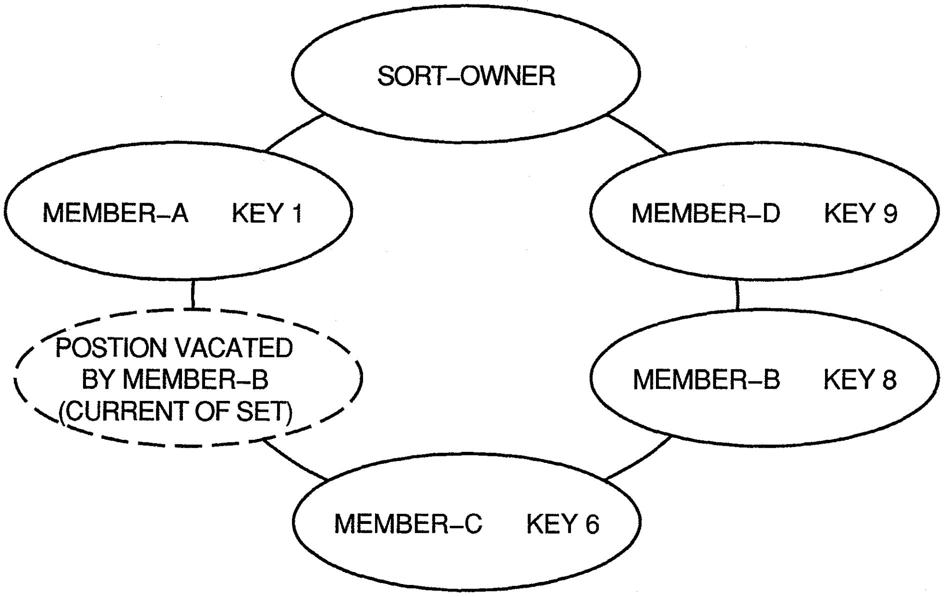
Figure 15–37: Modifying Members of Sorted Sets



When you change the contents of a data item specified in the ORDER IS SORTED clause and you do not want the set's currency to change, use the RETAINING clause with the MODIFY statement. Thus, MODIFY repositions the record and RETAINING keeps the currency indicator pointing at the position vacated by the record. Figure 15–38 shows how the following example retains currency for SORT_SET.

```
FETCH NEXT WITHIN SORT_SET.
IF MEMBER_KEY = "KEY 3"
   MOVE "KEY 8" TO MEMBER_KEY
   MODIFY MEMBER_KEY RETAINING SORT_SET.
```

Figure 15–38: After Modifying MEMBER_B and Using RETAINING



ZK-1512-GE

If MEMBER_B's key was changed to KEY 4, the record's position in the set occurrence would not change, and a FETCH NEXT WITHIN SORT_SET would fetch MEMBER_C.

15.24.6 CONNECT and DISCONNECT

When the set membership class is MANUAL, use the CONNECT statement to link a member record to its set occurrence. You can also use CONNECT for AUTOMATIC sets, provided that the retention class is OPTIONAL and you have disconnected the record.

When you use the CONNECT statement, specify the set or sets where the record is to be connected. Executing a CONNECT statement without the set list clause connects the record to all sets in which it can be, but is not yet, a member.

Before you execute a CONNECT statement, be sure that currency for the specified set type points to the correct set occurrence. If not, the member record will participate in the wrong set occurrence. (For more information on currency, see Section 15.22 and Section 15.23.) You cannot execute a CONNECT for a record that participates as an owner of the specified set.

If the set retention class is OPTIONAL, use the DISCONNECT statement to remove a member record from a specified set. The DISCONNECT statement does not delete a record from the database.

When you use the DISCONNECT statement, specify the sets from which the record will be disconnected. Executing a DISCONNECT without the set list clause disconnects the record from all the sets in which it participates as an optional member. You cannot execute a DISCONNECT for a record that participates as an owner of the specified set or that has a set retention class of FIXED or MANDATORY. Refer to the *VAX COBOL Reference Manual* for an explanation of how set membership class affects certain DML verbs.

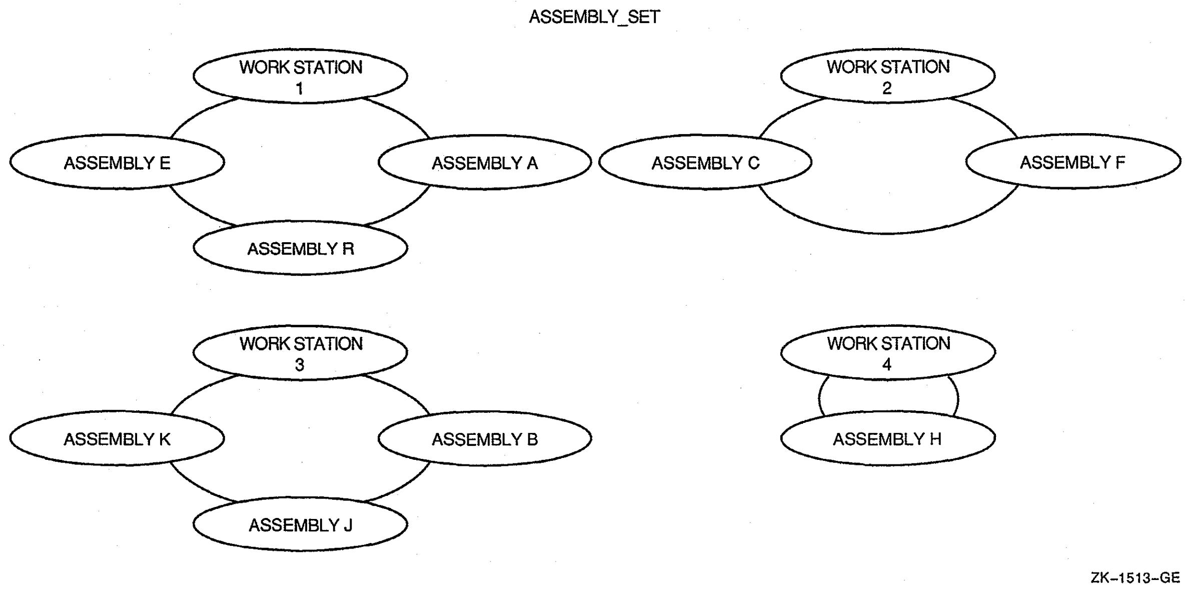
15.24.7 RECONNECT

Use the RECONNECT statement to remove a member record from one set occurrence and connect it to another occurrence of the same set type, or to a different position within the same set. To transfer a member record:

1. Use the FETCH (or FIND) statement to select a record in the set occurrence. This can be either a member or an owner of the set occurrence you want to connect to.
2. Use the FETCH (or FIND) statement with the RETAINING clause to transfer the member record you want. This keeps the currency for the targeted record.
3. Execute a RECONNECT statement using the WITHIN clause.

The RECONNECT statement is useful in applications such as production control where manufactured items move down an assembly line from one work station to another. In Figure 15–39, work stations are the owner records and assemblies are the member records.

Figure 15–39: Occurrence Diagram Prior to RECONNECT



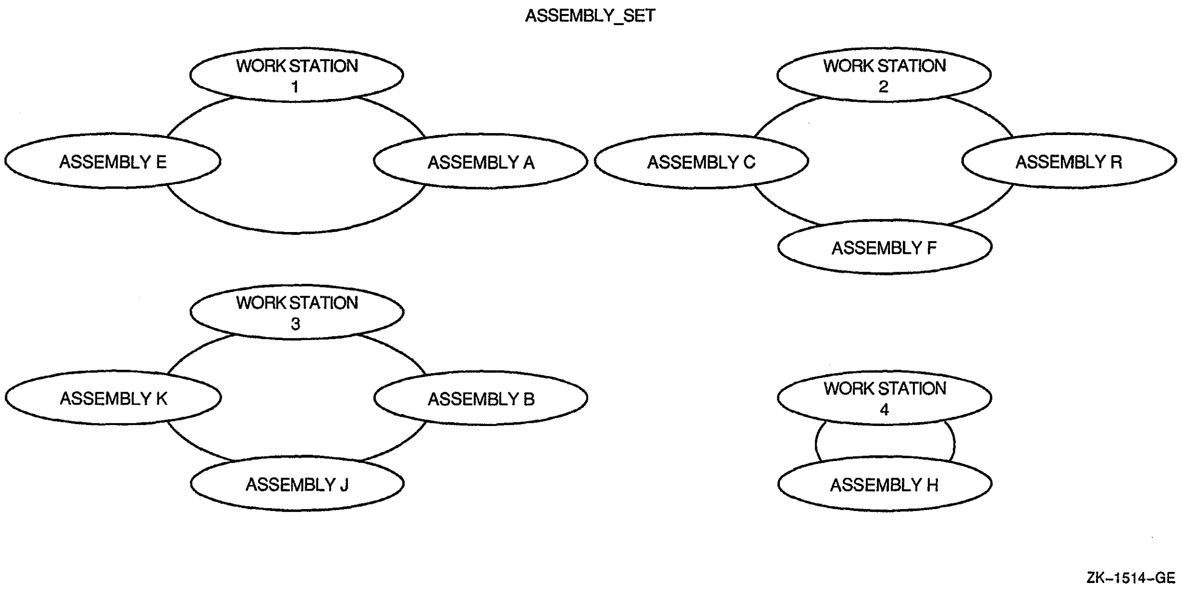
Example 15–7 transfers ASSEMBLY R, a machine base, to WORK STATION 2 for electrical assembly. The order of insertion is LAST.

Figure 15–40 shows the ASSEMBLY_SET after execution of the RECONNECT statement. Notice the ASSEMBLY A record replaces the R record's position in the WORK STATION 1 set occurrence. Also, execution of the RECONNECT makes the ASSEMBLY R record current for the ASSEMBLY_SET.

Example 15–7: RECONNECT Statement

```
    .
    .
    .
GET-WORK-STATION.
    MOVE 2 TO WORK_STATION_ID.
    FIND FIRST WORK_STATION USING WORK_STATION_ID.
    MOVE "R" TO ASSEMBLY_ID.
    FIND FIRST ASSEMBLY USING ASSEMBLY_ID
         RETAINING ASSEMBLY_SET.
 **********************************************************
 * The RETAINING clause retains work station 2 as         *
 * current of ASSEMBLY_SET. Otherwise, the found member   *
 * would be current of set and the RECONNECT would fail. *
 **********************************************************
    RECONNECT ASSEMBLY WITHIN ASSEMBLY_SET.
    .
    .
    .
```

Figure 15–40: Occurrence Diagram After RECONNECT



15.24.8 ERASE ALL

The ERASE statement deletes one or more records from the database. However, it can delete more than you intended. Accidental deletes can occur because of the ERASE statement's cascading effect. The cascading effect can happen whenever the erased record is the owner of a set. Thus, if the current record is an owner of a set type, an ERASE ALL deletes:

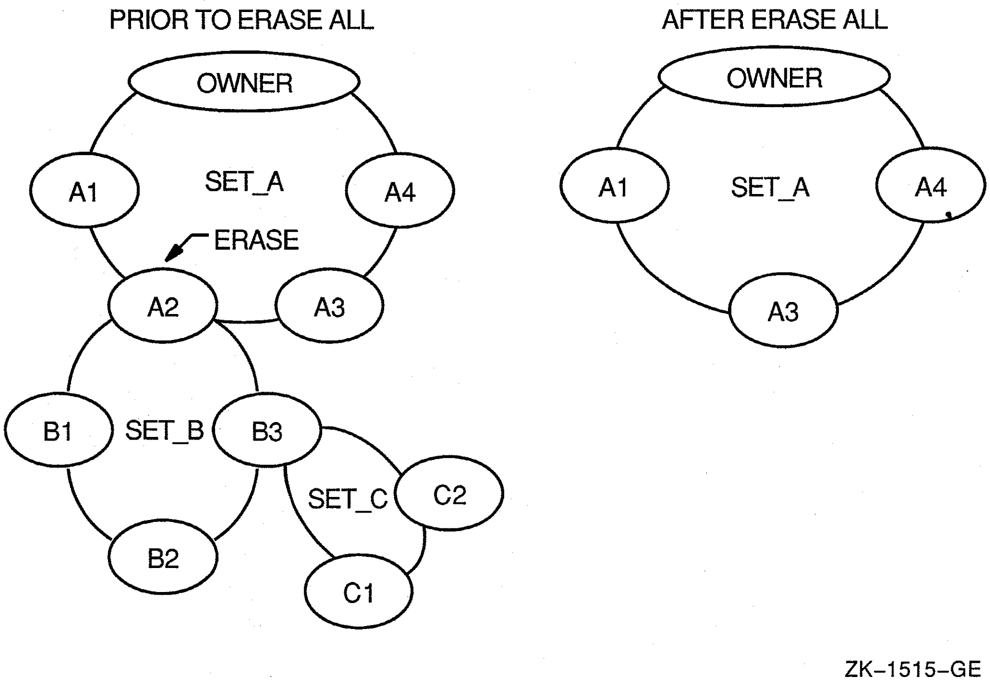
- The current record.
- All records in sets owned by the current record.

- Any records in sets owned by those members. Note that this is a repetitive process.

This is called a *cascading delete.*

The occurrence diagrams in Figure 15–41 show the results of using the ERASE ALL statement.

Figure 15–41: Results of an ERASE ALL



The ERASE ALL statement is the only way to erase an owner of sets with MANDATORY members.

15.24.9 ERASE Record-Name

If you do not use the ERASE ALL statement but use the ERASE record-name, and the erased record is the owner of a set, the ERASE statement deletes:

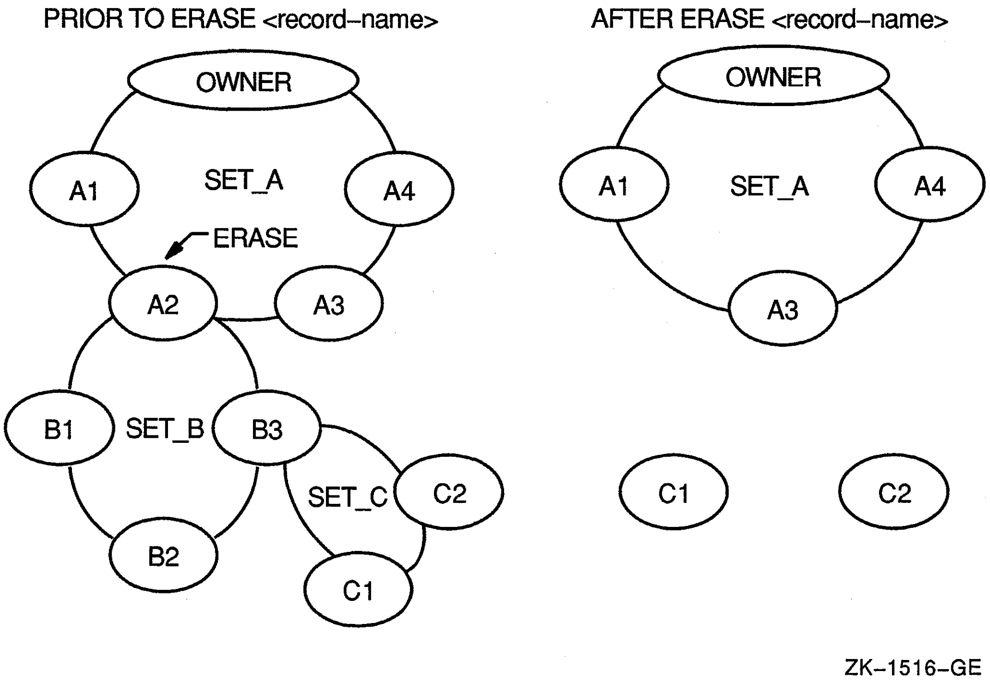
- The current record.
- All FIXED members of sets owned by the current record.
- All FIXED members of sets owned by records in rule 2. Note that this is a repetitive process.

If the current record owns sets with OPTIONAL members, these records are disconnected from the set, but remain in the database.

The occurrence diagrams in Figure 15–42 show the results of using the ERASE record-name statement when affected members have an OPTIONAL set membership. In this figure, B records are FIXED members of the SET_B set and C records are OPTIONAL members of the SET_C set. Notice that records

C1 and C2 are disconnected from the set, but remain in the database while B1 through B3 are erased.

Figure 15–42: Results of an ERASE Record-Name (with Both OPTIONAL and FIXED Retention Classes)



Remember, records removed from a set but not deleted from the database can still be accessed.

15.24.10 Freeing Currency Indicators

Use the FREE database-key-id statement to null the currency indicators for realms, records, sets, or the run unit. You use the FREE statement: (1) to establish a known currency condition before executing a program routine, and (2) to release record locks.

15.24.10.1 Establishing a Known Currency Condition

Establishing a known currency condition is helpful in many situations—for example, if you have a program that performs a customer analysis and prints three reports. The first report prints all customers with a credit rating greater than $1,000, the second report prints all customers with a credit rating greater than $5,000, and the third report prints all customers with a credit rating greater than $10,000. Because some customers will appear on more than one report, you want each report routine to start its customer analysis with the first customer in the database.

By using the FREE CURRENT statement at the end of a report routine, as shown in Example 15–8, you null the currency and allow the next print routine to start its analysis at the first customer.

Example 15–8: FREE CURRENT Statement

```
     .
     .
     .
MAIN-ROUTINE.
    READY TEST_REALM CONCURRENT RETRIEVAL.
    PERFORM FIRST-REPORT-HEADINGS.
    PERFORM PRINT-FIRST-REPORT THRU PFR-EXIT
            UNTIL AT-END = "Y".
    MOVE "N" TO AT-END.
    PERFORM SECOND-REPORT-HEADINGS.
    PERFORM PRINT-SECOND-REPORT THRU PSR-EXIT
            UNTIL AT-END = "Y".
    MOVE "N" TO AT-END.
    PERFORM THIRD-REPORT-HEADINGS.
    PERFORM PRINT-THIRD-REPORT THRU PTR-EXIT
            UNTIL AT-END = "Y".
    MOVE "N" TO AT-END.
     .
     .
     .
    STOP RUN.
PRINT-FIRST-REPORT.
    FETCH NEXT CUSTOMER_MASTER
          AT END FREE CURRENT
                 MOVE "Y" TO AT-END.
    IF AT-END = "N" AND
       CUSTOMER_CREDIT_RATING IS GREATER THAN 1000
           PERFORM PRINT-ROUTINE.
PFR-EXIT.
    EXIT.
PRINT-SECOND-REPORT.
    FETCH NEXT CUSTOMER_MASTER
          AT END FREE CURRENT
                 MOVE "Y" TO AT-END.
    IF AT-END = "N" AND
          CUSTOMER_CREDIT_RATING IS GREATER THAN 5000
          PERFORM PRINT-ROUTINE.
PSR-EXIT.
    EXIT.
PRINT-THIRD-REPORT.
    FETCH NEXT CUSTOMER_MASTER
          AT END MOVE "Y" TO AT-END.
    IF AT-END = "N" AND
       CUSTOMER_CREDIT_RATING IS GREATER THAN 10000
       PERFORM PRINT-ROUTINE.
PTR-EXIT.
    EXIT.
```

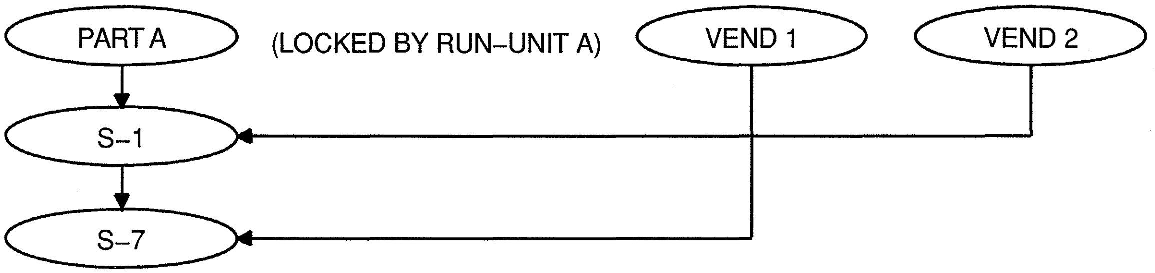
The FREE CURRENT statement in the PRINT-FIRST-REPORT paragraph nulls the default run-unit currency, thereby providing a starting point for the PRINT-SECOND-REPORT paragraph. The FREE CURRENT statement in the PRINT-SECOND-REPORT paragraph does the same for the PRINT-THIRD-REPORT paragraph. Thus, by nullifying the default run-unit currency, the FREE CURRENT statements allow the first execution of the FETCH NEXT CUSTOMER_MASTER statement to fetch the first customer master in TEST_REALM.

15.24.10.2 Releasing Record Locks

Regardless of the READY mode used, you always have a record lock on the current of run unit. Even the READY CONCURRENT RETRIEVAL mode locks the current record and puts it in a read-only condition. Furthermore, if you are traversing the database, the current record for each record type you touch with a DML statement is locked and placed in a read-only condition. Record locking prevents other users from updating any records locked by your run unit.

A locked record can prevent accessing of other records. Figure 15–43 shows PART A locked by run unit A. Assume PART A has been locked by a FETCH statement. If run unit B is in READY UPDATE mode and tries to: (1) update PART A, and (2) find all of PART A's member records and their vendor owners, then run unit B is locked out and placed in a wait state. A wait state occurs when a run unit cannot continue processing until another run unit completes its database transaction. Because run unit B uses PART A as an entry point for an update, the lock on PART A also prevents access to PART A's member records and the vendor owners of these member records.

Figure 15–43: Record Locking



ZK–1517–GE

If a record is not locked by a STORE or a MODIFY statement, or the database key for the record is not in a keeplist, you can unlock it by using the FREE CURRENT statement. By using the FREE CURRENT statement, you reduce lockout and optimize processing for other run units.

15.24.11 FIND and FETCH Statements

The FIND and FETCH statements locate a record in the database and make that record the current record of the run unit. The FETCH statement also copies the record to the user work area (UWA), thus giving you access to the record's data. The FIND does not place a record in the UWA. However, if your only requirement is to make a record current of run unit, use the more efficient FIND statement. For example, use the FIND statement if you want to connect, disconnect, or reconnect without examining a record's contents.

15.24.12 FIND ALL Option

The FIND ALL statement puts the database key values of one or more records into a keeplist. (See the description of FIND ALL in the *VAX COBOL Reference Manual* for syntax details.)

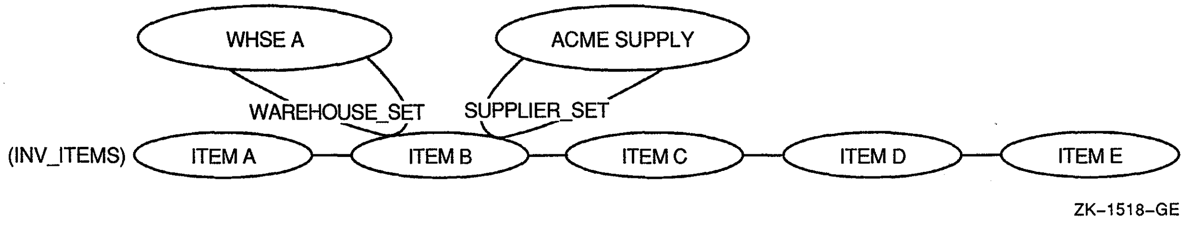
The following example locates all PART records with a PART_STATUS of J and puts their dbkey values in keeplist TWO.

```
FIND ALL TWO PART USING PART_STATUS
PART_STATUS X(1) = J
```

15.24.13 FIND NEXT and FETCH NEXT Loops

If you have a FIND NEXT or FETCH NEXT loop in your program, the first execution of the loop is the same as executing a FIND FIRST or FETCH FIRST. Unless you properly initialize them, currency indicators can affect selection of the specified record. For example, if ITEM B in Figure 15–44 is current for INV_ITEMS, a FIND NEXT INV_ITEMS makes ITEM C the current record for the run unit. You can null a currency by executing a FREE CURRENT statement.

Figure 15–44: Using FIND NEXT and FETCH NEXT Loops



Example 15–9 makes the INV_ITEMS currency null prior to executing a FETCH NEXT loop.

Example 15–9: FETCH NEXT Loop

```
        .
        .
        .
000100 GET-WAREHOUSE.
000110     MOVE "A" TO WHSE-ID.
000120     FIND FIRST WHSE_REC USING WHSE-ID.
000130 UPDATE-ITEM.
000140     MOVE "B" TO ITEM-ID.
000150     FETCH FIRST WITHIN WAREHOUSE_SET
000160          USING ITEM-ID.
     ****************************
     * INVENTORY UPDATE ROUTINE *
     ****************************
         .
         .
         .
     *****************************************************
     * The next statement nulls the run unit currency.   *
     * Therefore, the first execution of the FETCH NEXT  *
     * gets the first INV_ITEMS record.                  *
     *****************************************************
000170         FREE CURRENT.
000180 ANALYZE-INVENTORY.
000190     FETCH NEXT INV_ITEMS
000200           AT END GO TO END-OF-PROGRAM.
000210     GO TO ANALYZE-INVENTORY.
         .
         .
         .
```

You can also use FETCH NEXT and FIND NEXT loops to walk through a set type. Assume you have to walk through the WAREHOUSE_SET and reduce the reorder point quantity by 10 percent for all items with a cost greater than $500. Furthermore, you also want to check the supplier's credit terms for each of these items. You could perform the task as shown in Example 15–10.

Example 15–10: Using a FETCH NEXT Loop to Walk Through a Set Type

```
        .
        .
        .
000100 FETCH-WAREHOUSE.
000110     FETCH NEXT WHSE_REC
000120           AT END PERFORM END-OF-WAREHOUSE
000130                  PERFORM WRAP-UP.
000140 ITEM-LOOP.
000150      FETCH NEXT INV_ITEM WITHIN WAREHOUSE_SET
000160            AT END
000170               FIND OWNER WITHIN WAREHOUSE_SET
000180               PERFORM FETCH-WAREHOUSE.
000190      IF INV_ITEM_COST IS GREATER THAN 500
000200         PERFORM SUPPLIER-ANALYSIS.
000210*     Reduce reorder point quantity by 10%.
000220      MODIFY INV_ITEM.
000230      GO TO ITEM-LOOP.
```

(continued on next page)

Example 15–10 (Cont.): Using a FETCH NEXT Loop to Walk Through a Set Type

```
000240 SUPPLIER-ANALYSIS.
000250      IF NOT SUPPLIER_SET MEMBER
000260               DISPLAY "NO SUPPLIER FOR THIS ITEM"
000270               EXIT.
000280       FETCH OWNER WITHIN SUPPLIER_SET.
000290*      Check credit terms.
          .
          .
          .
```

Notice the FIND OWNER WITHIN WAREHOUSE_SET statement on line 000170. At the end of a WAREHOUSE_SET collection, statement 000170 sets the WAREHOUSE_SET type currency to the owner of the current occurrence. This allows the next execution of FETCH NEXT WHSE_REC to use current of record type WHSE_REC to find the next occurrence of WHSE_REC. Without statement 000170, a FETCH NEXT WHSE_REC would use the current of run unit, which is an INV_ITEM record type.

15.24.14 Qualifying FIND and FETCH

You can locate records by using the contents of data items as search arguments. You can use more than one qualifier as a search argument. For example, assume you want to print a report of all employees in department 5 with a pay rate of $7.50 per hour. You could use the department number as a search argument and use a conditional test to find all employees with a pay rate of $7.50. Or you could use both the department number and pay rate as search arguments, as follows:

```
     .
     .
     .
000500 SETUP-QUALIFIES.
000510     MOVE 5    TO DEPARTMENT-NUMBER.
000520     MOVE 7.50 TO EMPLOYEE-RATE.
000530     FREE CURRENT.
000540 FETCH-EMPLOYEES.
000550     FETCH NEXT EMPLOYEE
000560           USING DEPARTMENT-NUMBER EMPLOYEE-RATE
000570                 AT END GO TO EXIT-ROUTINE.
000580     PERFORM EMPLOYEE-PRINT.
000590     GO TO FETCH-EMPLOYEES.
     .
     .
     .
```

You can also locate records by using a WHERE clause to designate a conditional expression as a search argument. The following example fetches the first SUPPLY record whose SUP_LAG_TIME is 2 days or less.

```
000450 FETCH-SUPPLY.
000460       FETCH FIRST SUPPLY
000470             WITHIN PART_INFO
000480             WHERE SUP_LAG_TIME LESS THAN 2
000490             AT END GO TO EXIT-ROUTINE.
```

15.25 Handling Database Exception Conditions

This section discusses how to program for database exception conditions.

15.25.1 AT END Phrase

Use the AT END phrase of the FETCH and FIND statements to handle the end of a collection of records condition. Your program will terminate if: (1) an at end condition occurs, (2) the program does not include the AT END phrase, and (3) there is no applicable USE statement.

15.25.2 ON ERROR Phrase

Use the ON ERROR phrase to transfer execution control to the associated statements' error handling routine. Once in this routine your program can supply useful and effective debugging information. (See Section 15.25.4, and the *VAX COBOL Reference Manual* for more information on VAX DBMS Database Special Registers.) The ON ERROR phrase can be part of every DML statement. It allows you to gracefully plan the end of a program that would otherwise terminate abnormally. (In a FETCH or FIND statement, you cannot specify both the ON ERROR and AT END phrases in the same statement.) For example:

```
PROCEDURE DIVISION.
    .
    .
    .
    RECONNECT PARTS_RECORD WITHIN ALL
              ON ERROR DISPLAY "Exception on RECONNECT"
                       PERFORM PROCESS-EXCEPTION.
    .
    .
    .
PROCESS-EXCEPTION.
    DISPLAY "Database Exception Condition Report".
    DISPLAY " ".
    DISPLAY "DB-CONDITION            = ", DB-CONDITION
                                          WITH CONVERSION.
    DISPLAY "DB-CURRENT-RECORD-NAME  = ", DB-CURRENT-RECORD-NAME.
    DISPLAY "DB-CURRENT-RECORD-ID    = ", DB-CURRENT-RECORD-ID
                                          WITH CONVERSION.
    DISPLAY " ".
    CALL "DBM$SIGNAL".
    STOP RUN.
```

15.25.3 USE Statement

Planning for exception conditions is an effective way to increase program and programmer productivity. A program with USE statements is more flexible than a program without them. They minimize operator intervention and often reduce or eliminate the time a programmer needs to debug and rerun the program.

The USE statement traps unsuccessful run-time DBMS exception conditions that cause the execution of a Declarative procedure. A Declarative procedure can:

- Supply useful and effective database debugging information (see Section 15.25.4 and the *VAX COBOL Reference Manual* for more information on VAX DBMS Database Special Registers)

- Provide alternate processing paths for specific exception conditions

Two sets of USE statements follow:

- The first set, shown in Example 15–11, consists of a single USE statement. This database declarative executes for any and all database exception conditions. If you select this set, it must be the only database USE statement in the Declarative Section. Its format is:

  USE [GLOBAL] FOR DB-EXCEPTION.

Example 15–11: A Single USE Statement

```
PROCEDURE DIVISION.
DECLARATIVES.
200-DATABASE-EXCEPTIONS SECTION. USE FOR DB-EXCEPTION.
DB-ERROR-ROUTINE.
    DISPLAY "Database Exception Condition Report".
    DISPLAY "-------------------------------------------".
    DISPLAY "DB-CONDITION            = ", DB-CONDITION
                                          WITH CONVERSION.
    DISPLAY "DB-CUR-REC-NAME  = ", DB-CURRENT-RECORD-NAME.
    DISPLAY "DB-CURRENT-RECORD-ID    = ", DB-CURRENT-RECORD-ID
                                                 WITH CONVERSION.
    DISPLAY "DB-CUR-REC-ID    = ", DB-CRID.
    DISPLAY " ".
    CALL "DBM$SIGNAL".
END DECLARATIVES.
```

- The second set, shown in Example 15–12, consists of one or more Format 1 USE statements, and one Format 2 USE statement.

  Format 1

  USE [GLOBAL] FOR DB-EXCEPTION ON DBM$_exception-condition [, DBM$_exception-condition]...

  A Format 1 database declarative executes whenever a database exception condition occurs and the corresponding DBM$_exception-condition is explicitly stated in the USE statement.

  Format 2

  USE [GLOBAL] FOR DB-EXCEPTION ON OTHER.

  A Format 2 declarative executes whenever a database exception condition occurs and the corresponding DBM$_exception-condition is not explicitly stated in any Format 1 USE statement.

Example 15–12: Multiple USE Statements

```
PROCEDURE DIVISION.
DECLARATIVES.
200-DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION ON DBM$_CRELM_NULL,
                            DBM$_CRTYPE_NULL.
200-DATABASE.
    PERFORM 300-REPORT-DATABASE-EXCEPTIONS.
    IF DB-CONDITION = ..... GO TO ...
    IF DB-CONDITION = ..... GO TO ...
    STOP RUN.
225-DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION ON DBM$_DUPNOTALL.
225-DATABASE.
    PERFORM 300-REPORT-DATABASE-EXCEPTIONS.
    GO TO ...
250-DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION ON OTHER.
250-DATABASE.
    PERFORM 300-REPORT-DATABASE-EXCEPTIONS.
    EVALUATE DB-CONDITION
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN .....             GO TO ...
             WHEN OTHER             PERFORM... .
    STOP RUN.
300-REPORT-DATABASE-EXCEPTIONS.
    DISPLAY "Database Exception Condition Report".
    DISPLAY " ".
    DISPLAY "DB-CONDITION     = ",     DB-CONDITION
                                       WITH CONVERSION.
    DISPLAY "DB-CUR-REC-NAME  = ",     DB-CURRENT-RECORD-NAME.
    DISPLAY "DB-CURRENT-RECORD-ID = ", DB-CURRENT-RECORD-ID
    DISPLAY " ".
    CALL "DBM$SIGNAL".
```

15.25.4 How to Translate DB-CONDITION Values to Exception Messages

VAX DBMS includes the following procedure for exception condition handling:

```
CALL "DBM$SIGNAL".
```

Use this procedure when it is necessary to output an exception message rather than, or in addition to, displaying the numeric value of DB-CONDITION. For more information on the VAX DBMS database special register DB-CONDITION, see the *VAX COBOL Reference Manual.*

15.26 Debugging and Testing VAX COBOL DML Programs

The Database Query utility (DBQ) commands and generic DML statements are the tools you use to debug and test your COBOL program's DML statements. For example, you can use DBQ commands to display currency indicators, test program loops, or check your program's execution efficiency.

It is important to eliminate any logic errors prior to running a VAX COBOL DML program against a live database, because poorly written or incorrect logic can corrupt a database. You can resolve some logic errors by desk-checking a program. Desk-checking involves reviewing the logical ordering and proper use of DML statements; for example, executing a FIND when you intend to execute a FETCH, or executing a CONNECT instead of a RECONNECT. You can also use the debugger described in Chapter 3. However, neither method gives you information on currency indicators and the effects DML statements have on them.

Another method of debugging VAX COBOL DML programs is to test DML statements using the DBQ utility. DBQ is an online interactive utility that uses a split screen to show the results of each execution of a DML statement. It is also an effective database programming learning tool. For a complete description of the DBQ utility, refer to the VAX DBMS documentation on data manipulation and programming.

It is recommended that you use all of these tools to design, test, and debug your VAX COBOL DML programs.

NOTE

> The split screen feature of the DBQ utility is not available to users of VT52 or VT05 terminals.

15.27 DBQ Commands and DML Statements

The DBQ utility provides both generic DML statements and DBQ-specific commands. Generic DML statements are similar to the VAX COBOL DML statements explained in the *VAX COBOL Reference Manual.* However, not all COBOL DML syntax is applicable to the DBQ utility. These statements and entries do not apply:

- SUB-SCHEMA SECTION
- LD statement
- AT END phrase
- ON ERROR phrase
- Scope terminators
- USE statement
- DB statement—Use the DBQ utility BIND command to identify the subschema you will use for testing and debugging. You cannot access a subschema until you bind it. If your program has this DB statement:

  ```
  DB PARTSS3 WITHIN PARTS FOR NEW.
  ```

  the comparable BIND statement is as follows:

  ```
  dbq>BIND PARTSS3 FOR NEW
  ```

- ANY clause—The DBQ utility does not allow the ANY clause in a Record Selection Expression. Instead, use the FIRST clause.
- DUPLICATE clause—The DBQ utility does not allow the DUPLICATE clause in a Record Selection Expression. Instead, use the NEXT clause.
- WHERE clause—The operators of this clause are different.

For a complete discussion of generic DML, refer to the VAX DBMS documentation on data manipulation and programming.

15.28 Sample Debugging and Testing Session

This section shows how to use the DBQ utility for debugging and testing VAX COBOL DML programs. Because the split screen limits the number of lines that can be displayed at one time, the split screen figures show the Bachman diagram only. Corresponding DBQ prompts, entries, and messages follow each Bachman diagram and are shown in their entirety.

The session tests and finds a logic error in the DML program statements in Example 15–13. The sample COBOL DML program is intended to:

1. Fetch the first PART in the database with a PART_ID equal to AZ177311
2. Fetch all SUPPLY records for the found PART
3. Check the PART's SUPPLY records for SUP_RATINGs equal to 0
4. Change all SUP_RATINGs equal to 0 to 5, and print SUPPLY records VENDOR_SUPPLY owners
5. Change PART's PART_STATUS to X if one or more of its SUPPLY records has a SUP_RATING equal to 5

Remember, the database key values displayed on your screen may be different from those in the examples.

NOTE

> If you are currently accessing PARTSS3 with the DBQ utility and have made any changes to the database, use the ROLLBACK statement to cancel your changes. Otherwise, you might change the results of the debugging session.

Example 15–13: Sample VAX COBOL DML Program Statements

```
DATA DIVISION.
DB PARTSS3 WITHIN PARTS FOR NEW.
    .
    .
    .
PROCEDURE DIVISION.
000-BEGIN.
    READY PROTECTED UPDATE.
    .
    .
    .
    MOVE "AZ177311" TO PART_ID.
    FETCH FIRST PART USING PART_ID.
    MOVE "N" TO END-OF-COLLECTION.
    PERFORM A100-LOOP THROUGH A100-LOOP-EXIT
            UNTIL END-OF-COLLECTION = "Y".
    .
    .
    .
    STOP RUN.
```

(continued on next page)

Example 15–13 (Cont.): Sample VAX COBOL DML Program Statements

```
A100-LOOP.
    FETCH NEXT WITHIN PART_SUPPLY
          AT END MOVE "Y" TO END-OF-COLLECTION
                 GO TO A100-LOOP-EXIT.
    IF SUP_RATING = "0"
          MOVE "5" TO SUP_RATING
          MODIFY SUP_RATING
          MOVE 1 TO MODIFY-COUNT
          FETCH OWNER WITHIN VENDOR_SUPPLY
          PERFORM PRINT-VENDOR.
    IF MODIFY-COUNT = 1
          MOVE "X" TO PART_STATUS
          MODIFY PART_STATUS.
A100-LOOP-EXIT.
    EXIT.
```

The following DBQ session tests and debugs the sample DML program statements in Example 15–13:

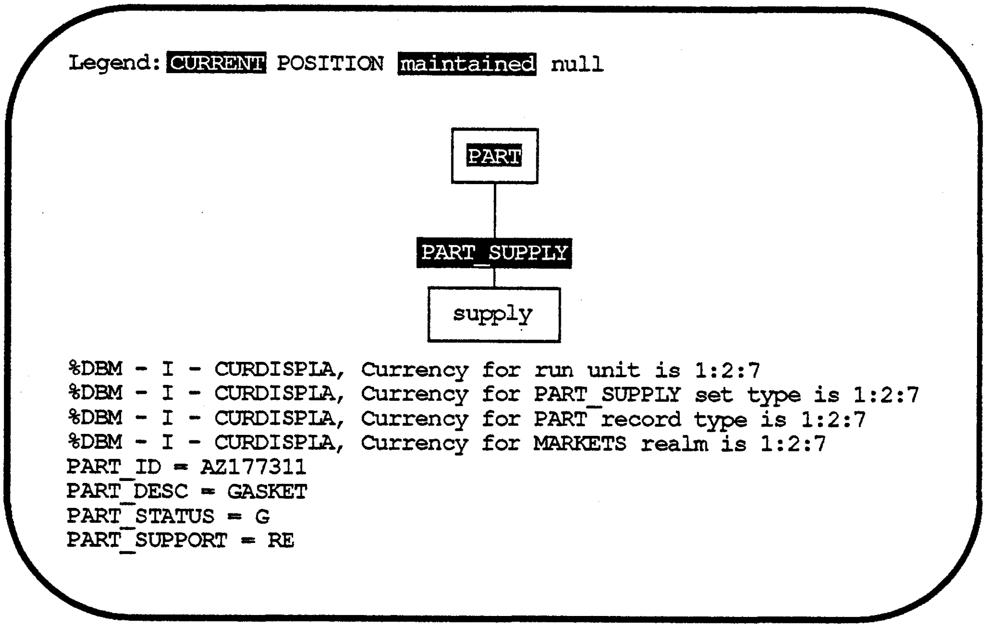
```
$ DBQ
dbq> BIND PARTSS3 FOR NEW
dbq> READY PROTECTED UPDATE
dbq> SET CURSIG
dbq> FETCH FIRST PART USING PART_ID
```

DBQ prompts you for a PART_ID value:

```
PART_ID [CHARACTER(8)] = AZ177311
```

Entering AZ177311 as the PART_ID value causes the Bachman diagram in Figure 15–45 to appear on your screen.

Figure 15–45: Split Screen After FETCH FIRST PART USING PART_ID



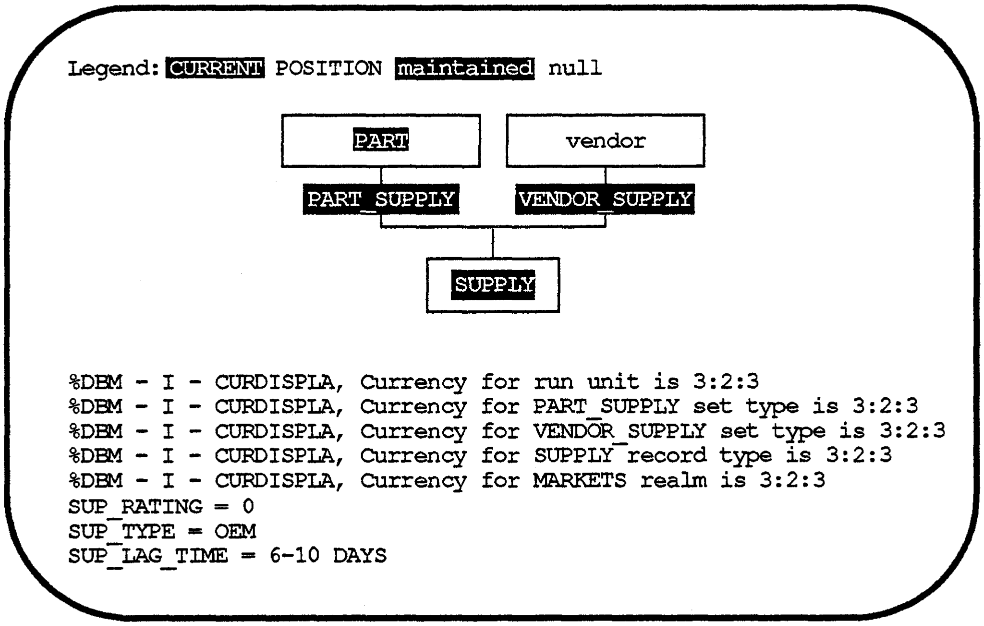
```
Legend: CURRENT POSITION maintained null

%DBM - I - CURDISPLA, Currency for run unit is 1:2:7
%DBM - I - CURDISPLA, Currency for PART_SUPPLY set type is 1:2:7
%DBM - I - CURDISPLA, Currency for PART record type is 1:2:7
%DBM - I - CURDISPLA, Currency for MARKETS realm is 1:2:7
PART_ID = AZ177311
PART_DESC = GASKET
PART_STATUS = G
PART_SUPPORT = RE
```

ZK-6067-GE

The next DML statement in Figure 15–46 is FETCH NEXT WITHIN PART_SUPPLY. Although this statement is in a performed loop, you can still test its logic by executing a series of FETCH NEXT WITHIN PART_SUPPLY until you find a SUP_RATING equal to 0.

```
dbq> FETCH NEXT WITHIN PART_SUPPLY
```

Figure 15–46: Split Screen After FETCH NEXT WITHIN PART_SUPPLY



```
%DBM - I - CURDISPLA, Currency for run unit is 3:2:3
%DBM - I - CURDISPLA, Currency for PART_SUPPLY set type is 3:2:3
%DBM - I - CURDISPLA, Currency for VENDOR_SUPPLY set type is 3:2:3
%DBM - I - CURDISPLA, Currency for SUPPLY record type is 3:2:3
%DBM - I - CURDISPLA, Currency for MARKETS realm is 3:2:3
SUP_RATING = 0
SUP_TYPE = OEM
SUP_LAG_TIME = 6-10 DAYS
```

ZK-6068-GE

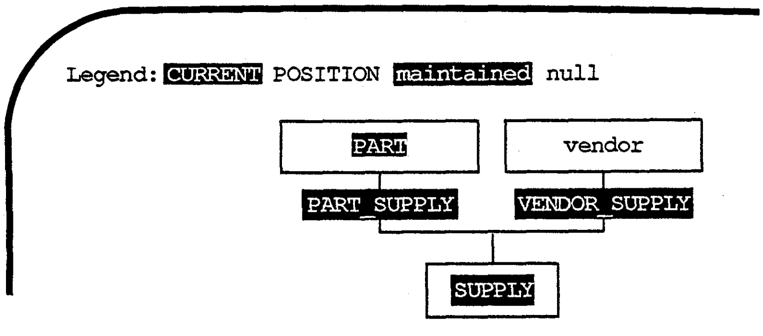
Because SUPPLY participates in two sets, the Bachman diagram in Figure 15–46 shows the set relationships for SUPPLY. Notice the SUPPLY record has a SUP_RATING equal to 0. Therefore, you can test the next DML statement.

```
dbq> MODIFY SUP_RATING
SUP_RATING [CHARACTER(1)]= 5
```

Notice how the MODIFY statement causes DBQ to issue a prompt, as shown in the preceding statement. When you MODIFY or STORE a record, DBQ prompts you for data entry by displaying the data name and its attributes. After entering the new SUP_RATING, use the RETURN key to execute the MODIFY statement.

Because this MODIFY statement does not change currency, the Bachman diagram in Figure 15–47 is the same as the one in Figure 15–46. Also, DBQ does not display currency update messages.

Figure 15–47: Split Screen After MODIFY SUP_RATING

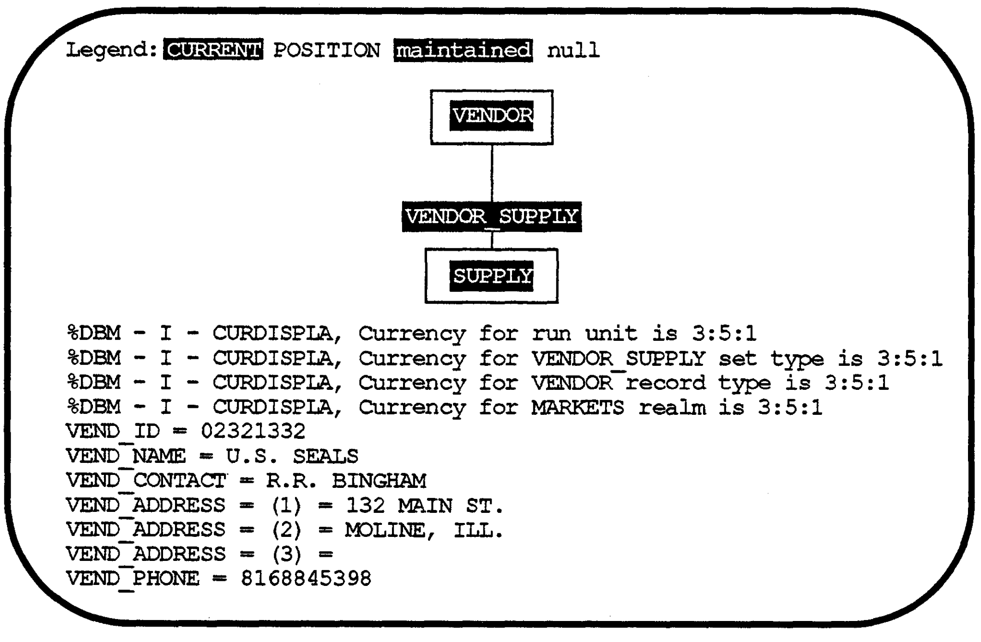


ZK-6069-GE

The next statement to test is the FETCH for SUPPLY record's owner in the VENDOR_SUPPLY set.

```
dbq> FETCH OWNER WITHIN VENDOR_SUPPLY
```

Figure 15–48: Split Screen After FETCH OWNER WITHIN VENDOR_SUPPLY



```
%DBM - I - CURDISPLA, Currency for run unit is 3:5:1
%DBM - I - CURDISPLA, Currency for VENDOR_SUPPLY set type is 3:5:1
%DBM - I - CURDISPLA, Currency for VENDOR record type is 3:5:1
%DBM - I - CURDISPLA, Currency for MARKETS realm is 3:5:1
VEND_ID = 02321332
VEND_NAME = U.S. SEALS
VEND_CONTACT = R.R. BINGHAM
VEND_ADDRESS = (1) = 132 MAIN ST.
VEND_ADDRESS = (2) = MOLINE, ILL.
VEND_ADDRESS = (3) =
VEND_PHONE = 8168845398
```

ZK-6070-GE

Assuming the data item MODIFY-COUNT has a value 1, you can test the last (MODIFY PART) DML statement.

```
dbq> MODIFY PART_STATUS
PART_STATUS [CHARACTER(1)]= X
dbq> DBM-F-WRONGRTYP, Specified record type not current record type
```

DBQ generates an error message indicating the MODIFY statement did not execute because the current of run unit is not a PART record. Comparing the shading and intensities of the Bachman diagram in Figure 15–48 with the legend shows the current record is a VENDOR record. Therefore, the diagram indicates that a MODIFY to the PART record will not work even before you attempt the MODIFY statement.

To correct the logic error, PART must be the current record type prior to execution of the MODIFY PART_STATUS statement. One way to correct the logic error is to execute a FETCH CURRENT PART statement before the MODIFY PART_STATUS statement. Example 15–14 shows a corrected version of the sample COBOL DML program statements in Example 15–13.

Example 15–14: Sample DML Program Statements

```
DATA DIVISION.
DB PARTSS3 WITHIN PARTS FOR NEW.
    .
    .
    .
PROCEDURE DIVISION.
000-BEGIN.
    READY PROTECTED UPDATE.
    .
    .
    .
    MOVE "AZ177311" TO PART_ID.
    FETCH FIRST PART USING PART_ID.
    MOVE "N" TO END-OF-COLLECTION.
    PERFORM A100-LOOP THROUGH A100-LOOP-EXIT
            UNTIL END-OF-COLLECTION = "Y".
    .
    .
    .
    STOP RUN.
A100-LOOP.
    FETCH NEXT WITHIN PART_SUPPLY
          AT END MOVE "Y" TO END-OF-COLLECTION
                 GO TO A100-LOOP-EXIT.
    IF SUP_RATING = "0"
          MOVE "5" TO SUP_RATING
          MODIFY SUP_RATING
          MOVE 1 TO MODIFY-COUNT
          FETCH OWNER WITHIN VENDOR_SUPPLY
          PERFORM PRINT-VENDOR.
    IF MODIFY-COUNT = 1
          MOVE "X" TO PART_STATUS
          FETCH CURRENT PART RETAINING PART_SUPPLY
          MODIFY PART_STATUS.
A100-LOOP-EXIT.
    EXIT.
```

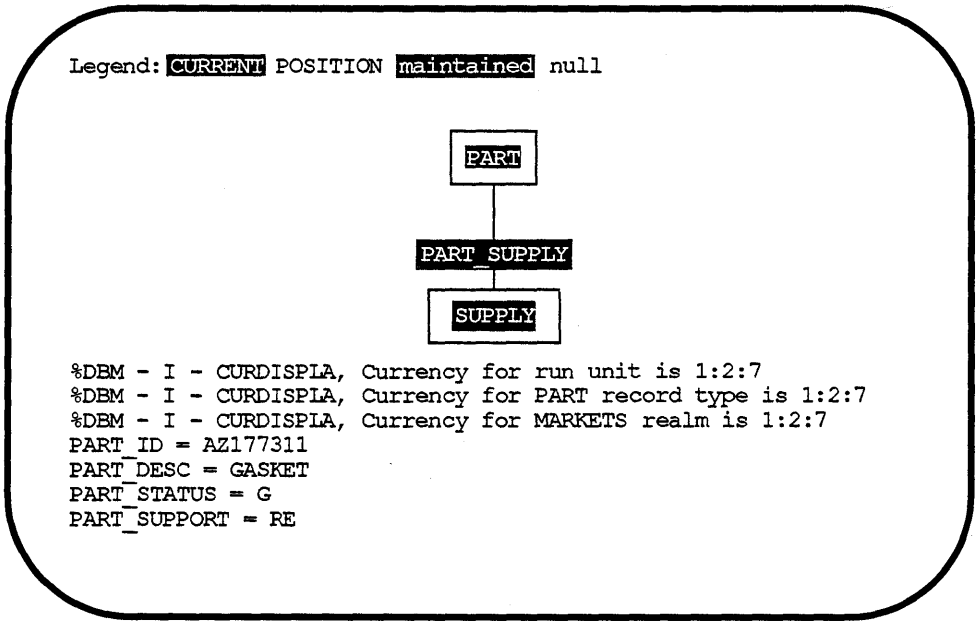
The FETCH CURRENT PART statement uses the RETAINING clause to keep the current SUPPLY record as current of PART_SUPPLY.

Continue testing, starting with the new FETCH statement.

```
dbq> FETCH CURRENT PART RETAINING PART_SUPPLY
```

Figure 15–49 shows that executing FETCH CURRENT PART RETAINING PART_SUPPLY makes PART the current record type, while the RETAINING clause keeps SUPPLY current of PART_SUPPLY set. Retaining the current supply record as current of PART_SUPPLY means the next execution of FETCH NEXT WITHIN PART_SUPPLY uses the current SUPPLY record's currency to locate the next SUPPLY record. If you executed a FETCH CURRENT PART without the RETAINING clause, a FETCH NEXT WITHIN PART_SUPPLY would use PART's currency and FETCH the first SUPPLY record belonging to PART.

Figure 15–49: Split Screen After FETCH CURRENT PART RETAINING PART_SUPPLY



```
%DBM - I - CURDISPLA, Currency for run unit is 1:2:7
%DBM - I - CURDISPLA, Currency for PART record type is 1:2:7
%DBM - I - CURDISPLA, Currency for MARKETS realm is 1:2:7
PART_ID = AZ177311
PART_DESC = GASKET
PART_STATUS = G
PART_SUPPORT = RE
```

ZK–6072–GE

Now you can retest the MODIFY PART_STATUS.

```
dbq> MODIFY PART_STATUS
PART_STATUS [CHARACTER(1)]= X
dbq>
```

The DBQ prompt indicates the MODIFY was successful.

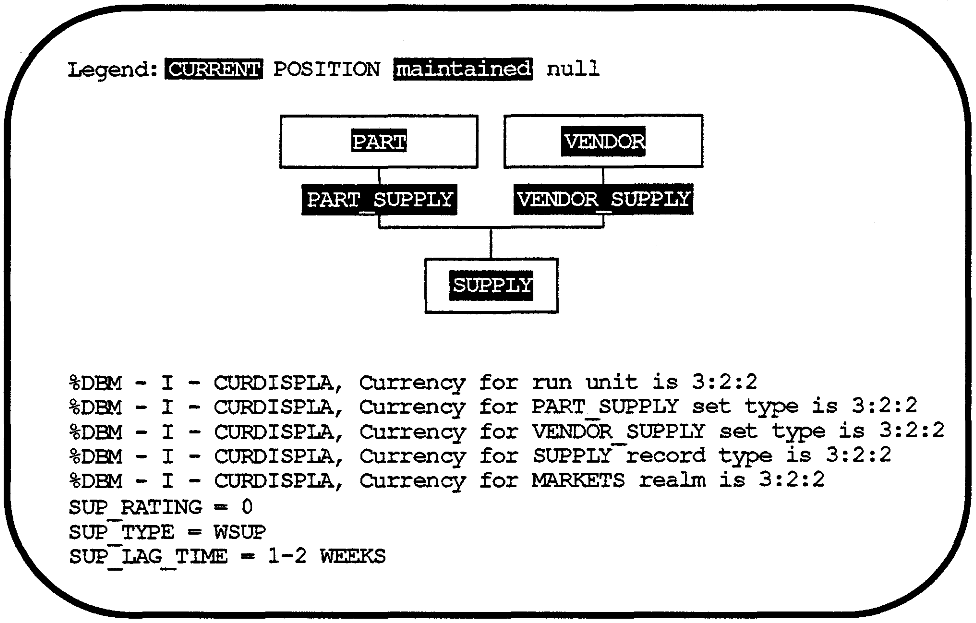
With the logic error found and fixed, you can test to see if the next execution of the FETCH NEXT WITHIN PART_SUPPLY fetches the next SUPPLY record belonging to the first PART record.

```
dbq> FETCH NEXT WITHIN PART_SUPPLY
```

The database keys displayed by the currency update messages in Figure 15–50 and Figure 15–51 are the same, thereby showing the A100-LOOP paragraph will fetch the next SUPPLY record owned by the first PART record.

Notice the data items also have the same value. Comparing data item contents instead of database key values is not a good practice because duplicate records may be allowed. For example, a PART may have two or more SUPPLY records containing the same data. Also, each SUPPLY record could point to a different owner in the VENDOR_SUPPLY set type.

Figure 15–50: Split Screen After FETCH NEXT WITHIN PART_SUPPLY



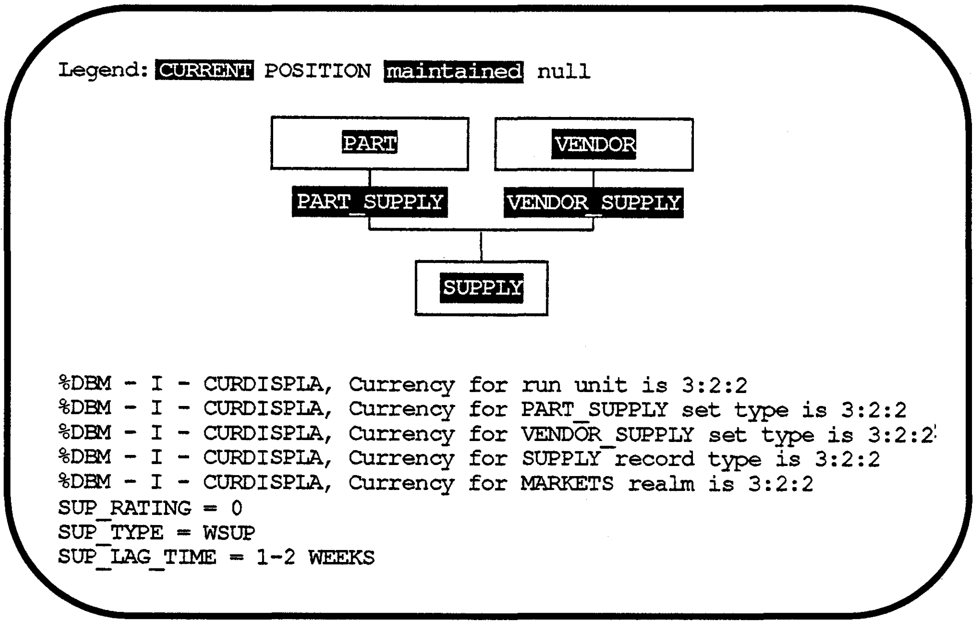
```
%DBM - I - CURDISPLA, Currency for run unit is 3:2:2
%DBM - I - CURDISPLA, Currency for PART_SUPPLY set type is 3:2:2
%DBM - I - CURDISPLA, Currency for VENDOR_SUPPLY set type is 3:2:2
%DBM - I - CURDISPLA, Currency for SUPPLY record type is 3:2:2
%DBM - I - CURDISPLA, Currency for MARKETS realm is 3:2:2
SUP_RATING = 0
SUP_TYPE = WSUP
SUP_LAG_TIME = 1-2 WEEKS
```

ZK-6073-GE

To show that the record is indeed the second SUPPLY record belonging to the first PART record, execute the following statement:

```
dbq> FETCH 2 WITHIN PART_SUPPLY
```

Figure 15–51: Split Screen After FETCH 2 WITHIN PART_SUPPLY



```
%DBM - I - CURDISPLA, Currency for run unit is 3:2:2
%DBM - I - CURDISPLA, Currency for PART_SUPPLY set type is 3:2:2
%DBM - I - CURDISPLA, Currency for VENDOR_SUPPLY set type is 3:2:2
%DBM - I - CURDISPLA, Currency for SUPPLY record type is 3:2:2
%DBM - I - CURDISPLA, Currency for MARKETS realm is 3:2:2
SUP_RATING = 0
SUP_TYPE = WSUP
SUP_LAG_TIME = 1-2 WEEKS
```

ZK–6074–GE

15.29 Reading a VAX COBOL Subschema Map Listing

The circled numbers on the programs PARTSS1-PROGRAM (Figure 15–52) and PARTSS3-PROGRAM (Figure 15–53) correspond to the following numbered text explanations. The examples in this chapter refer to the subschema map listings in this section.

❶ Subschema map. Lists the realms, records, and sets defined in the subschema and a COBOL-like record description for each record. Use the /MAP qualifier to get this listing.

❷ The complete pathname of the subschema node in CDD/Plus.

❸ A list of the description entries associated with the subschema.

❹ The subschema version number, also called the subschema time-stamp, indicates the date and time at which the subschema was successfully compiled and placed in CDD/Plus by the DDL Utility.

❺ The schema version number, also called the schema time-stamp, indicates the date and time at which the schema was successfully compiled and placed in CDD/Plus by the DDL Utility.

❻ The subschema name.

❼ The schema name.

❽ A list of the description entries associated with a realm entry.

❾ The realm name.

❿ A list of the areas that make up the realm.

⓫ A list of the description entries associated with a record entry.

⓬ A list of the areas in which the record can be stored. The assignment of records to areas is made in the record entry of the schema.

⓭ A list of the sets in the subschema in which the record participates as an owner record.

⓮ A list of the sets in the subschema in which the record participates as a member record.

⓯ A COBOL-like representation of the record entry defined in the subschema or the schema.

⓰ A list of the description entries associated with a set entry.

⓱ The set name.

⓲ Identifies the record as owner of the set. This field is SYSTEM for a singular set.

⓳ Identifies a member record of a set.

⓴ The insertion class of set membership.

㉑ The retention class of set membership.

㉒ Indicates the set ordering criterion.

NOTE

Items 3, 8, 11, and 16 are generated by the DDL Utility.

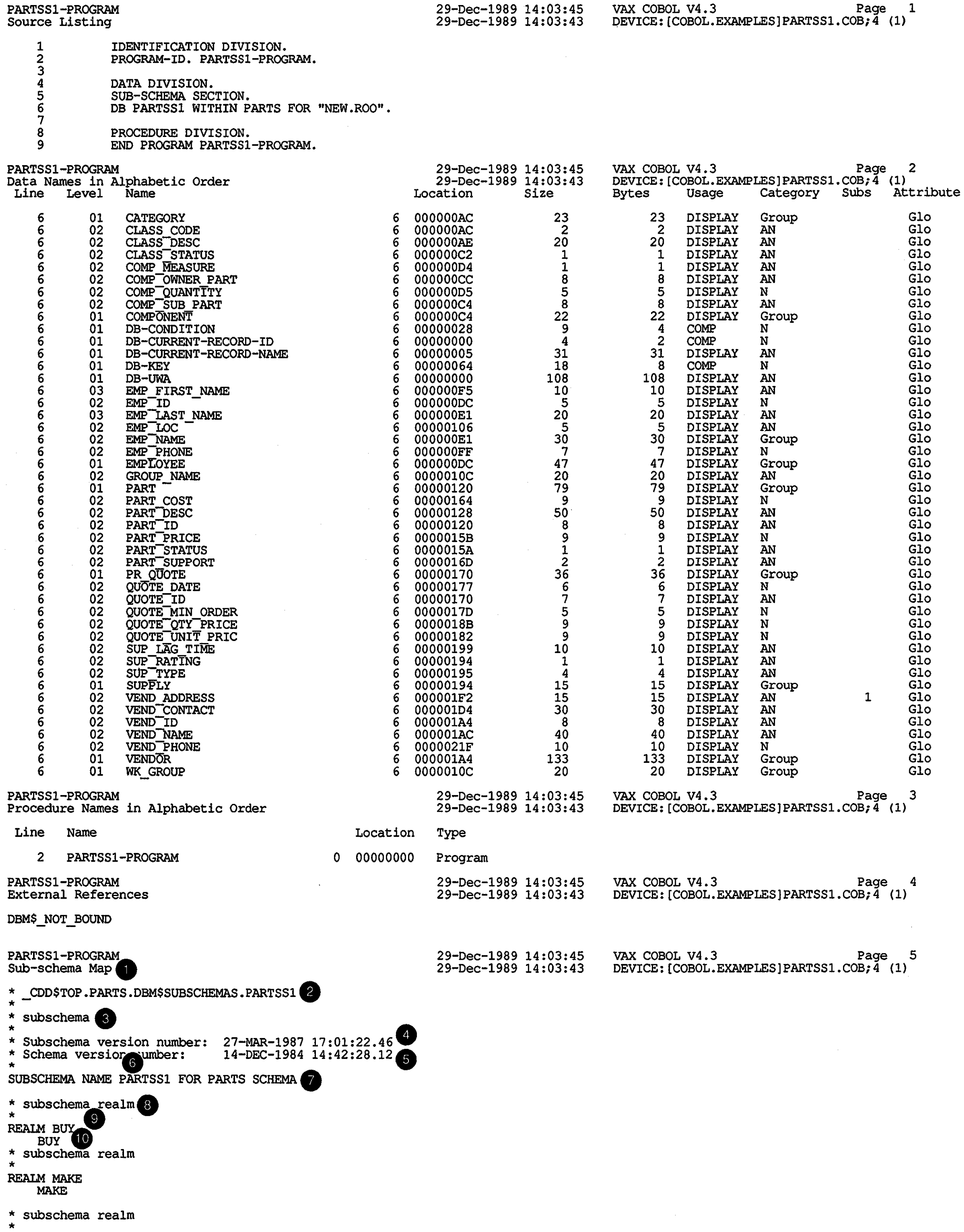
Items 6 and 7 may not be the same names that appear in the DB statement. They are the names that result after consideration of the value of CDD$DEFAULT and all logical name translation.

Only the set name appears in a subschema. The specification of owner and member record types, as well as set membership and ordering, is done in the schema.

15.29.1 PARTSS1 Subschema Map Listing

PARTSS1-PROGRAM in Figure 15–52 includes the VAX COBOL Subschema Map of the PARTSS1 subschema.

Figure 15–52: PARTSS1-PROGRAM Compiler Listing

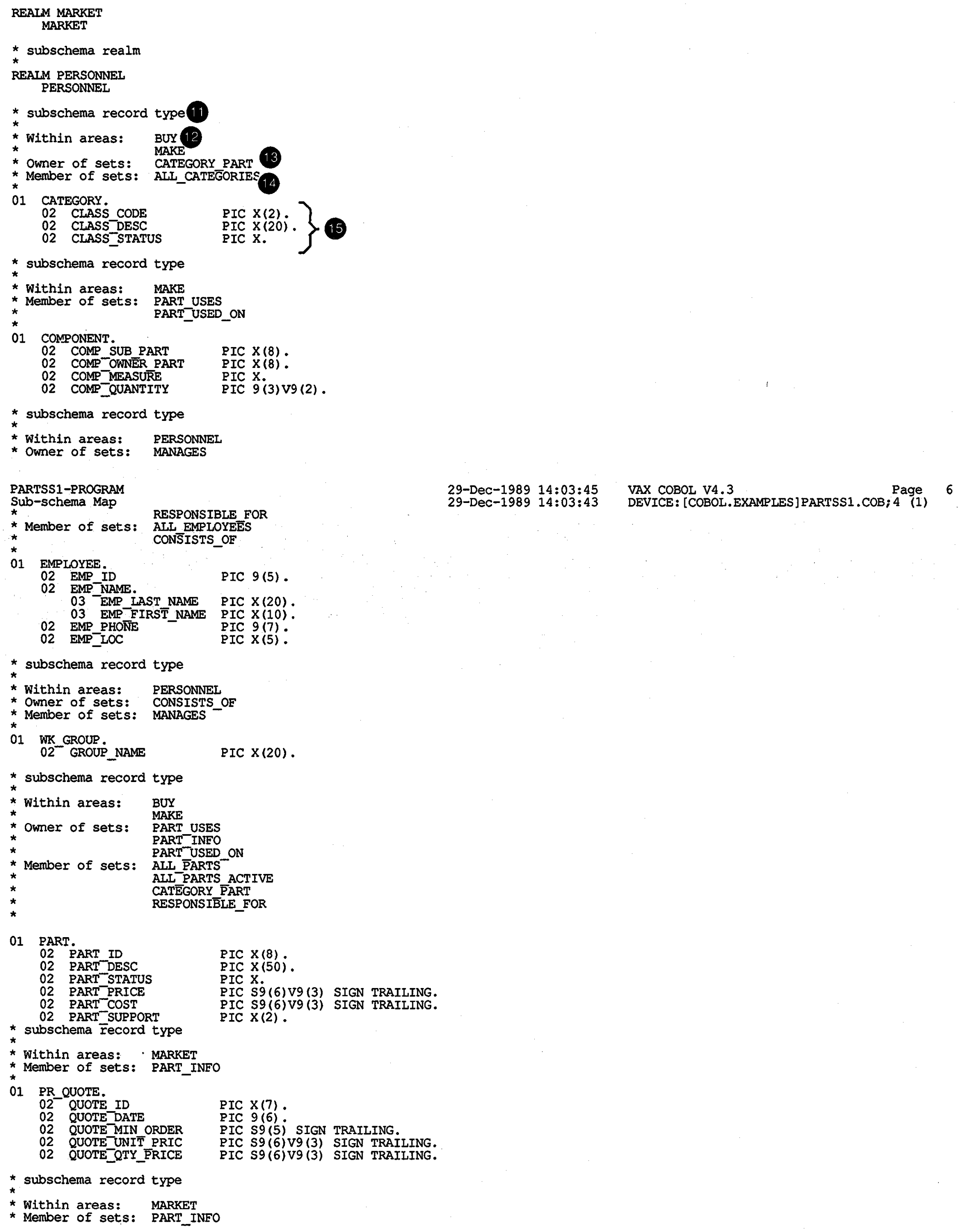


```
PARTSS1-PROGRAM                                      29-Dec-1989 14:03:45    VAX COBOL V4.3                Page   1
Source Listing                                       29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)

    1          IDENTIFICATION DIVISION.
    2          PROGRAM-ID. PARTSS1-PROGRAM.
    3
    4          DATA DIVISION.
    5          SUB-SCHEMA SECTION.
    6          DB PARTSS1 WITHIN PARTS FOR "NEW.ROO".
    7
    8          PROCEDURE DIVISION.
    9          END PROGRAM PARTSS1-PROGRAM.

PARTSS1-PROGRAM                                      29-Dec-1989 14:03:45    VAX COBOL V4.3                Page   2
Data Names in Alphabetic Order                       29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
 Line   Level   Name                              Location         Size      Bytes    Usage     Category  Subs  Attribute

    6     01    CATEGORY                        6 000000AC           23         23    DISPLAY   Group            Glo
    6     02    CLASS_CODE                      6 000000AC            2          2    DISPLAY   AN               Glo
    6     02    CLASS_DESC                      6 000000AE           20         20    DISPLAY   AN               Glo
    6     02    CLASS_STATUS                    6 000000C2            1          1    DISPLAY   AN               Glo
    6     02    COMP_MEASURE                    6 000000D4            1          1    DISPLAY   AN               Glo
    6     02    COMP_OWNER_PART                 6 000000CC            8          8    DISPLAY   AN               Glo
    6     02    COMP_QUANTITY                   6 000000D5            5          5    DISPLAY   N                Glo
    6     02    COMP_SUB_PART                   6 000000C4            8          8    DISPLAY   AN               Glo
    6     01    COMPONENT                       6 000000C4           22         22    DISPLAY   Group            Glo
    6     01    DB-CONDITION                    6 00000028            9          4    COMP      N                Glo
    6     01    DB-CURRENT-RECORD-ID            6 00000000            4          2    COMP      N                Glo
    6     01    DB-CURRENT-RECORD-NAME          6 00000005           31         31    DISPLAY   AN               Glo
    6     01    DB-KEY                          6 00000064           18          8    COMP      N                Glo
    6     01    DB-UWA                          6 00000000          108        108    DISPLAY   AN               Glo
    6     03    EMP_FIRST_NAME                  6 000000F5           10         10    DISPLAY   AN               Glo
    6     02    EMP_ID                          6 000000DC            5          5    DISPLAY   N                Glo
    6     03    EMP_LAST_NAME                   6 000000E1           20         20    DISPLAY   AN               Glo
    6     02    EMP_LOC                         6 00000106            5          5    DISPLAY   AN               Glo
    6     02    EMP_NAME                        6 000000E1           30         30    DISPLAY   Group            Glo
    6     02    EMP_PHONE                       6 000000FF            7          7    DISPLAY   N                Glo
    6     01    EMPLOYEE                        6 000000DC           47         47    DISPLAY   Group            Glo
    6     02    GROUP_NAME                      6 0000010C           20         20    DISPLAY   AN               Glo
    6     01    PART                            6 00000120           79         79    DISPLAY   Group            Glo
    6     02    PART_COST                       6 00000164            9          9    DISPLAY   N                Glo
    6     02    PART_DESC                       6 00000128           50         50    DISPLAY   AN               Glo
    6     02    PART_ID                         6 00000120            8          8    DISPLAY   AN               Glo
    6     02    PART_PRICE                      6 0000015B            9          9    DISPLAY   N                Glo
    6     02    PART_STATUS                     6 0000015A            1          1    DISPLAY   AN               Glo
    6     02    PART_SUPPORT                    6 0000016D            2          2    DISPLAY   AN               Glo
    6     01    PR_QUOTE                        6 00000170           36         36    DISPLAY   Group            Glo
    6     02    QUOTE_DATE                      6 00000177            6          6    DISPLAY   N                Glo
    6     02    QUOTE_ID                        6 00000170            7          7    DISPLAY   AN               Glo
    6     02    QUOTE_MIN_ORDER                 6 0000017D            5          5    DISPLAY   N                Glo
    6     02    QUOTE_QTY_PRICE                 6 0000018B            9          9    DISPLAY   N                Glo
    6     02    QUOTE_UNIT_PRIC                 6 00000182            9          9    DISPLAY   N                Glo
    6     02    SUP_LAG_TIME                    6 00000199           10         10    DISPLAY   AN               Glo
    6     02    SUP_RATING                      6 00000194            1          1    DISPLAY   AN               Glo
    6     02    SUP_TYPE                        6 00000195            4          4    DISPLAY   AN               Glo
    6     01    SUPPLY                          6 00000194           15         15    DISPLAY   Group            Glo
    6     02    VEND_ADDRESS                    6 000001F2           15         15    DISPLAY   AN          1    Glo
    6     02    VEND_CONTACT                    6 000001D4           30         30    DISPLAY   AN               Glo
    6     02    VEND_ID                         6 000001A4            8          8    DISPLAY   AN               Glo
    6     02    VEND_NAME                       6 000001AC           40         40    DISPLAY   AN               Glo
    6     02    VEND_PHONE                      6 0000021F           10         10    DISPLAY   N                Glo
    6     01    VENDOR                          6 000001A4          133        133    DISPLAY   Group            Glo
    6     01    WK_GROUP                        6 0000010C           20         20    DISPLAY   Group            Glo

PARTSS1-PROGRAM                                      29-Dec-1989 14:03:45    VAX COBOL V4.3                Page   3
Procedure Names in Alphabetic Order                  29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)

 Line   Name                             Location    Type

    2   PARTSS1-PROGRAM                0 00000000    Program

PARTSS1-PROGRAM                                      29-Dec-1989 14:03:45    VAX COBOL V4.3                Page   4
External References                                  29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)

DBM$_NOT_BOUND


PARTSS1-PROGRAM                                      29-Dec-1989 14:03:45    VAX COBOL V4.3                Page   5
Sub-schema Map ❶                                     29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)

* _CDD$TOP.PARTS.DBM$SUBSCHEMAS.PARTSS1 ❷
*
* subschema ❸
*
* Subschema version number:  27-MAR-1987 17:01:22.46 ❹
* Schema version number:     14-DEC-1984 14:42:28.12 ❺
*           ❻
SUBSCHEMA NAME PARTSS1 FOR PARTS SCHEMA ❼

* subschema realm ❽
*         ❾
REALM BUY
    BUY ❿
* subschema realm
*
REALM MAKE
    MAKE

* subschema realm
*
```

ZK-6430-GE

(continued on next page)

Figure 15–52 (Cont.): PARTSS1-PROGRAM Compiler Listing

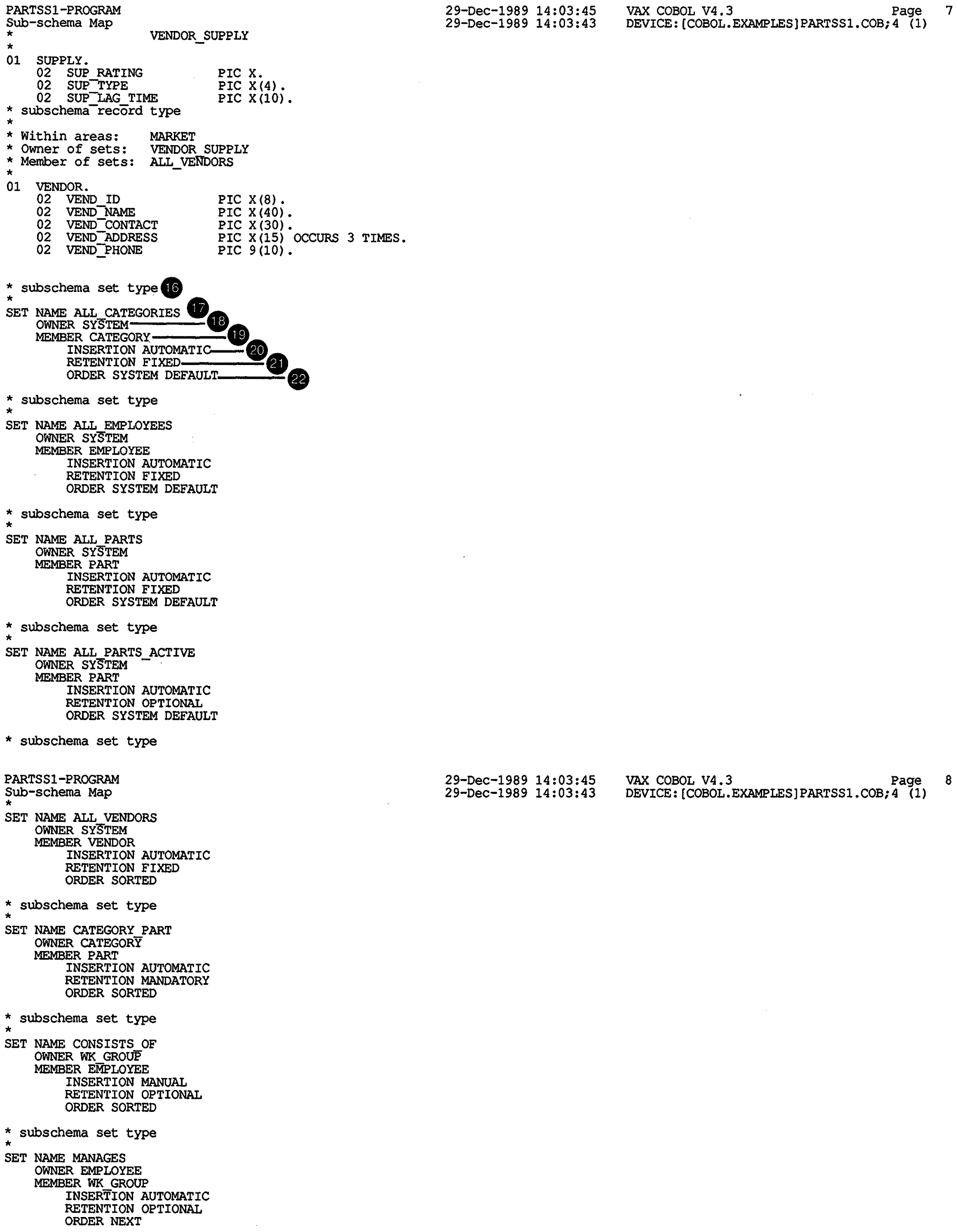

11
12
13
14
15

```
REALM MARKET
    MARKET

* subschema realm
*
REALM PERSONNEL
    PERSONNEL

* subschema record type
*
* Within areas:    BUY
*                  MAKE
* Owner of sets:   CATEGORY_PART
* Member of sets:  ALL_CATEGORIES
*
01  CATEGORY.
    02  CLASS_CODE          PIC X(2).
    02  CLASS_DESC          PIC X(20).
    02  CLASS_STATUS        PIC X.

* subschema record type
*
* Within areas:    MAKE
* Member of sets:  PART_USES
*                  PART_USED_ON
*
01  COMPONENT.
    02  COMP_SUB_PART       PIC X(8).
    02  COMP_OWNER_PART     PIC X(8).
    02  COMP_MEASURE        PIC X.
    02  COMP_QUANTITY       PIC 9(3)V9(2).

* subschema record type
*
* Within areas:    PERSONNEL
* Owner of sets:   MANAGES


PARTSS1-PROGRAM                                            29-Dec-1989 14:03:45    VAX COBOL V4.3                     Page   6
Sub-schema Map                                             29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
*                  RESPONSIBLE_FOR
* Member of sets:  ALL_EMPLOYEES
*                  CONSISTS_OF
*
01  EMPLOYEE.
    02  EMP_ID              PIC 9(5).
    02  EMP_NAME.
        03  EMP_LAST_NAME   PIC X(20).
        03  EMP_FIRST_NAME  PIC X(10).
    02  EMP_PHONE           PIC 9(7).
    02  EMP_LOC             PIC X(5).

* subschema record type
*
* Within areas:    PERSONNEL
* Owner of sets:   CONSISTS_OF
* Member of sets:  MANAGES
*
01  WK_GROUP.
    02  GROUP_NAME          PIC X(20).

* subschema record type
*
* Within areas:    BUY
*                  MAKE
* Owner of sets:   PART_USES
*                  PART_INFO
*                  PART_USED_ON
* Member of sets:  ALL_PARTS
*                  ALL_PARTS_ACTIVE
*                  CATEGORY_PART
*                  RESPONSIBLE_FOR
*

01  PART.
    02  PART_ID             PIC X(8).
    02  PART_DESC           PIC X(50).
    02  PART_STATUS         PIC X.
    02  PART_PRICE          PIC S9(6)V9(3) SIGN TRAILING.
    02  PART_COST           PIC S9(6)V9(3) SIGN TRAILING.
    02  PART_SUPPORT        PIC X(2).
* subschema record type
*
* Within areas:    MARKET
* Member of sets:  PART_INFO
*
01  PR_QUOTE.
    02  QUOTE_ID            PIC X(7).
    02  QUOTE_DATE          PIC 9(6).
    02  QUOTE_MIN_ORDER     PIC S9(5) SIGN TRAILING.
    02  QUOTE_UNIT_PRIC     PIC S9(6)V9(3) SIGN TRAILING.
    02  QUOTE_QTY_PRICE     PIC S9(6)V9(3) SIGN TRAILING.

* subschema record type
*
* Within areas:    MARKET
* Member of sets:  PART_INFO
```

ZK-6430-1-GE

(continued on next page)

Figure 15–52 (Cont.): PARTSS1-PROGRAM Compiler Listing

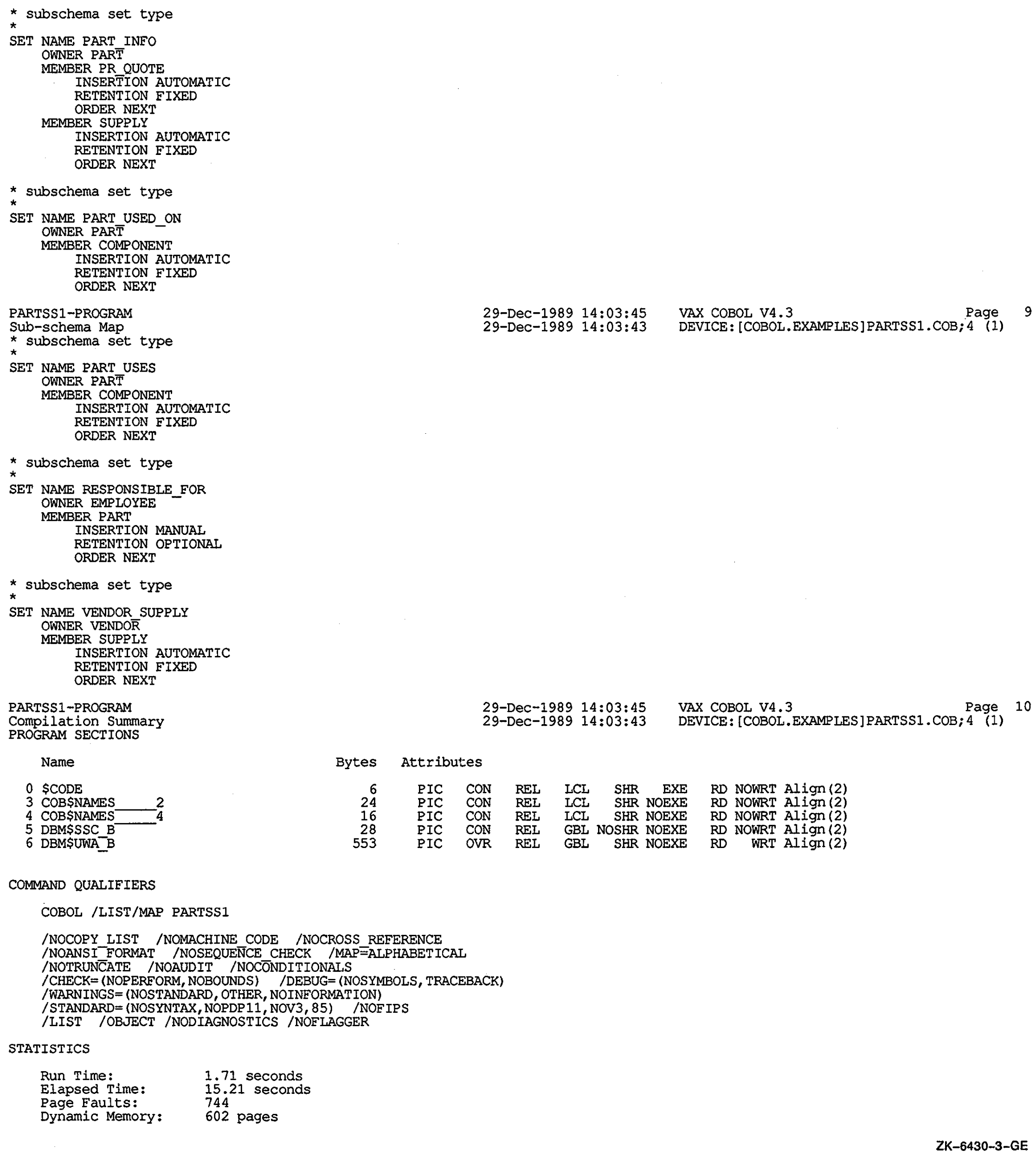

16
17
18
19
20
21
22

```
PARTSS1-PROGRAM                                    29-Dec-1989 14:03:45   VAX COBOL V4.3                     Page   7
Sub-schema Map                                     29-Dec-1989 14:03:43   DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
*                  VENDOR_SUPPLY
*
01  SUPPLY.
    02  SUP_RATING          PIC X.
    02  SUP_TYPE            PIC X(4).
    02  SUP_LAG_TIME        PIC X(10).
* subschema record type
*
* Within areas:    MARKET
* Owner of sets:   VENDOR_SUPPLY
* Member of sets:  ALL_VENDORS
*
01  VENDOR.
    02  VEND_ID             PIC X(8).
    02  VEND_NAME           PIC X(40).
    02  VEND_CONTACT        PIC X(30).
    02  VEND_ADDRESS        PIC X(15) OCCURS 3 TIMES.
    02  VEND_PHONE          PIC 9(10).

* subschema set type 16
*
SET NAME ALL_CATEGORIES 17
    OWNER SYSTEM 18
    MEMBER CATEGORY 19
        INSERTION AUTOMATIC 20
        RETENTION FIXED 21
        ORDER SYSTEM DEFAULT 22
* subschema set type
*
SET NAME ALL_EMPLOYEES
    OWNER SYSTEM
    MEMBER EMPLOYEE
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER SYSTEM DEFAULT
* subschema set type
*
SET NAME ALL_PARTS
    OWNER SYSTEM
    MEMBER PART
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER SYSTEM DEFAULT
* subschema set type
*
SET NAME ALL_PARTS_ACTIVE
    OWNER SYSTEM
    MEMBER PART
        INSERTION AUTOMATIC
        RETENTION OPTIONAL
        ORDER SYSTEM DEFAULT
* subschema set type

PARTSS1-PROGRAM                                    29-Dec-1989 14:03:45   VAX COBOL V4.3                     Page   8
Sub-schema Map                                     29-Dec-1989 14:03:43   DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
*
SET NAME ALL_VENDORS
    OWNER SYSTEM
    MEMBER VENDOR
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER SORTED
* subschema set type
*
SET NAME CATEGORY_PART
    OWNER CATEGORY
    MEMBER PART
        INSERTION AUTOMATIC
        RETENTION MANDATORY
        ORDER SORTED
* subschema set type
*
SET NAME CONSISTS_OF
    OWNER WK_GROUP
    MEMBER EMPLOYEE
        INSERTION MANUAL
        RETENTION OPTIONAL
        ORDER SORTED
* subschema set type
*
SET NAME MANAGES
    OWNER EMPLOYEE
    MEMBER WK_GROUP
        INSERTION AUTOMATIC
        RETENTION OPTIONAL
        ORDER NEXT
```

ZK-6430-2-GE

(continued on next page)

Figure 15–52 (Cont.): PARTSS1-PROGRAM Compiler Listing



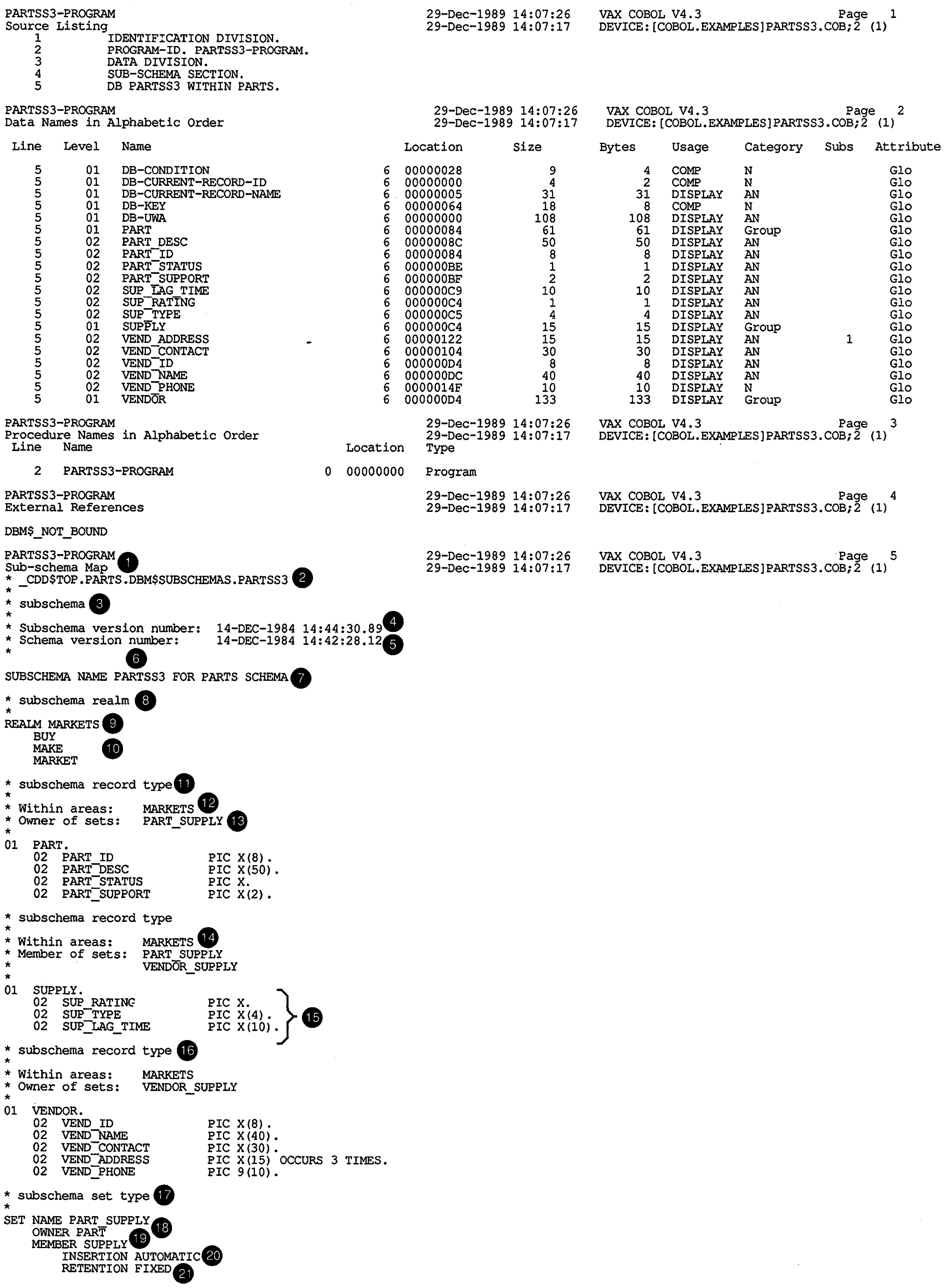
```
* subschema set type
*
SET NAME PART_INFO
    OWNER PART
    MEMBER PR_QUOTE
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT
    MEMBER SUPPLY
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT

* subschema set type
*
SET NAME PART_USED_ON
    OWNER PART
    MEMBER COMPONENT
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT

PARTSS1-PROGRAM                                           29-Dec-1989 14:03:45    VAX COBOL V4.3                     Page   9
Sub-schema Map                                            29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
* subschema set type
*
SET NAME PART_USES
    OWNER PART
    MEMBER COMPONENT
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT

* subschema set type
*
SET NAME RESPONSIBLE_FOR
    OWNER EMPLOYEE
    MEMBER PART
        INSERTION MANUAL
        RETENTION OPTIONAL
        ORDER NEXT

* subschema set type
*
SET NAME VENDOR_SUPPLY
    OWNER VENDOR
    MEMBER SUPPLY
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT

PARTSS1-PROGRAM                                           29-Dec-1989 14:03:45    VAX COBOL V4.3                     Page  10
Compilation Summary                                       29-Dec-1989 14:03:43    DEVICE:[COBOL.EXAMPLES]PARTSS1.COB;4 (1)
PROGRAM SECTIONS

    Name                                 Bytes   Attributes

  0 $CODE                                    6     PIC    CON    REL    LCL    SHR    EXE    RD NOWRT Align(2)
  3 COB$NAMES_____2                         24     PIC    CON    REL    LCL    SHR NOEXE    RD NOWRT Align(2)
  4 COB$NAMES_____4                         16     PIC    CON    REL    LCL    SHR NOEXE    RD NOWRT Align(2)
  5 DBM$SSC_B                               28     PIC    CON    REL    GBL  NOSHR NOEXE    RD NOWRT Align(2)
  6 DBM$UWA_B                              553     PIC    OVR    REL    GBL    SHR NOEXE    RD   WRT Align(2)

COMMAND QUALIFIERS

    COBOL /LIST/MAP PARTSS1

    /NOCOPY_LIST  /NOMACHINE_CODE  /NOCROSS_REFERENCE
    /NOANSI_FORMAT  /NOSEQUENCE_CHECK  /MAP=ALPHABETICAL
    /NOTRUNCATE  /NOAUDIT  /NOCONDITIONALS
    /CHECK=(NOPERFORM,NOBOUNDS)  /DEBUG=(NOSYMBOLS,TRACEBACK)
    /WARNINGS=(NOSTANDARD,OTHER,NOINFORMATION)
    /STANDARD=(NOSYNTAX,NOPDP11,NOV3,85)  /NOFIPS
    /LIST  /OBJECT /NODIAGNOSTICS /NOFLAGGER

STATISTICS

    Run Time:           1.71 seconds
    Elapsed Time:       15.21 seconds
    Page Faults:        744
    Dynamic Memory:     602 pages
```

ZK-6430-3-GE

15.29.2 PARTSS3 Subschema Map Listing

PARTSS3-PROGRAM in Figure 15–53 includes the VAX COBOL Subschema Map of the PARTSS3 subschema.

Figure 15–53: PARTSS3-PROGRAM Compiler Listing

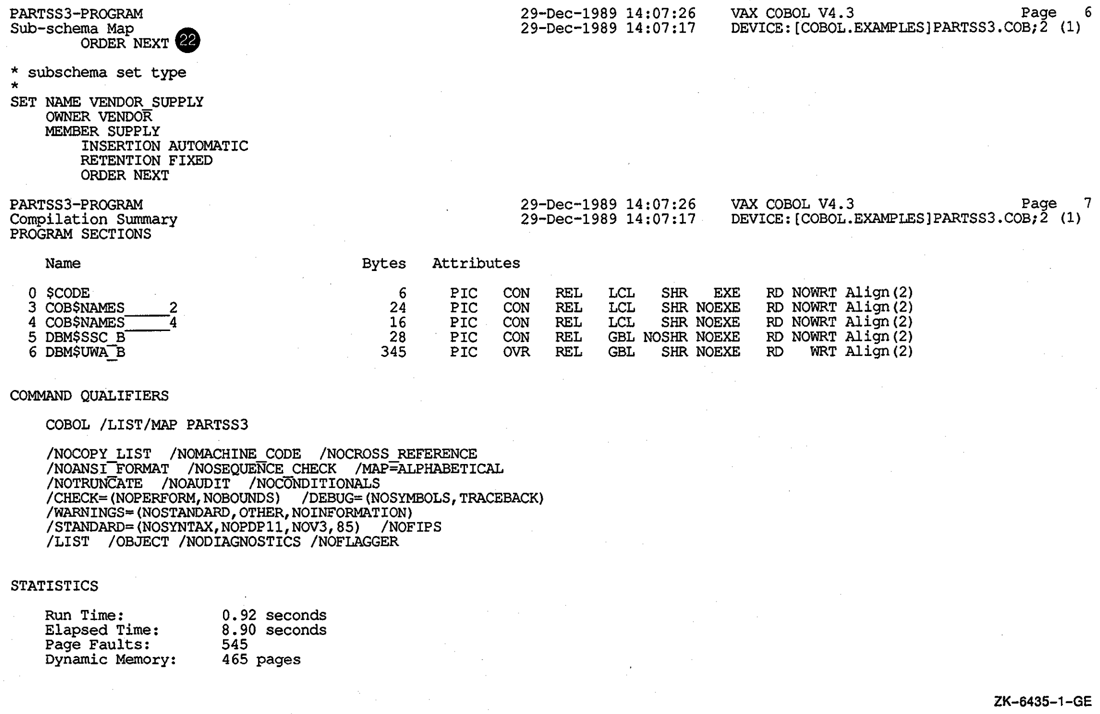


```
PARTSS3-PROGRAM                                             29-Dec-1989 14:07:26   VAX COBOL V4.3                 Page   1
Source Listing                                              29-Dec-1989 14:07:17   DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)
    1          IDENTIFICATION DIVISION.
    2          PROGRAM-ID. PARTSS3-PROGRAM.
    3          DATA DIVISION.
    4          SUB-SCHEMA SECTION.
    5          DB PARTSS3 WITHIN PARTS.

PARTSS3-PROGRAM                                              29-Dec-1989 14:07:26   VAX COBOL V4.3                 Page   2
Data Names in Alphabetic Order                               29-Dec-1989 14:07:17   DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)

 Line   Level   Name                                   Location          Size         Bytes    Usage      Category   Subs   Attribute

    5     01    DB-CONDITION                          6  00000028            9            4    COMP       N                   Glo
    5     01    DB-CURRENT-RECORD-ID                  6  00000000            4            2    COMP       N                   Glo
    5     01    DB-CURRENT-RECORD-NAME                6  00000005           31           31    DISPLAY    AN                  Glo
    5     01    DB-KEY                                6  00000064           18            8    COMP       N                   Glo
    5     01    DB-UWA                                6  00000000          108          108    DISPLAY    AN                  Glo
    5     01    PART                                  6  00000084           61           61    DISPLAY    Group               Glo
    5     02    PART_DESC                             6  0000008C           50           50    DISPLAY    AN                  Glo
    5     02    PART_ID                               6  00000084            8            8    DISPLAY    AN                  Glo
    5     02    PART_STATUS                           6  000000BE            1            1    DISPLAY    AN                  Glo
    5     02    PART_SUPPORT                          6  000000BF            2            2    DISPLAY    AN                  Glo
    5     02    SUP_LAG_TIME                          6  000000C9           10           10    DISPLAY    AN                  Glo
    5     02    SUP_RATING                            6  000000C4            1            1    DISPLAY    AN                  Glo
    5     02    SUP_TYPE                              6  000000C5            4            4    DISPLAY    AN                  Glo
    5     01    SUPPLY                                6  000000C4           15           15    DISPLAY    Group               Glo
    5     02    VEND_ADDRESS                          6  00000122           15           15    DISPLAY    AN           1      Glo
    5     02    VEND_CONTACT                          6  00000104           30           30    DISPLAY    AN                  Glo
    5     02    VEND_ID                               6  000000D4            8            8    DISPLAY    AN                  Glo
    5     02    VEND_NAME                             6  000000DC           40           40    DISPLAY    AN                  Glo
    5     02    VEND_PHONE                            6  0000014F           10           10    DISPLAY    N                   Glo
    5     01    VENDOR                                6  000000D4          133          133    DISPLAY    Group               Glo

PARTSS3-PROGRAM                                             29-Dec-1989 14:07:26   VAX COBOL V4.3                 Page   3
Procedure Names in Alphabetic Order                         29-Dec-1989 14:07:17   DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)
 Line   Name                                    Location    Type

    2   PARTSS3-PROGRAM                       0 00000000    Program

PARTSS3-PROGRAM                                             29-Dec-1989 14:07:26   VAX COBOL V4.3                 Page   4
External References                                         29-Dec-1989 14:07:17   DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)

DBM$_NOT_BOUND

PARTSS3-PROGRAM                                             29-Dec-1989 14:07:26   VAX COBOL V4.3                 Page   5
Sub-schema Map ❶                                            29-Dec-1989 14:07:17   DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)
* _CDD$TOP.PARTS.DBM$SUBSCHEMAS.PARTSS3 ❷
*
* subschema ❸
*
* Subschema version number:  14-DEC-1984 14:44:30.89❹
* Schema version number:     14-DEC-1984 14:42:28.12❺
*                 ❻
SUBSCHEMA NAME PARTSS3 FOR PARTS SCHEMA❼

* subschema realm ❽
*
REALM MARKETS❾
    BUY
    MAKE      ❿
    MARKET

* subschema record type⓫
*
* Within areas:    MARKETS⓬
* Owner of sets:   PART_SUPPLY ⓭
*
01  PART.
    02  PART_ID              PIC X(8).
    02  PART_DESC            PIC X(50).
    02  PART_STATUS          PIC X.
    02  PART_SUPPORT         PIC X(2).

* subschema record type
*
* Within areas:    MARKETS⓮
* Member of sets:  PART_SUPPLY
*                  VENDOR_SUPPLY
*
01  SUPPLY.
    02  SUP_RATING           PIC X.      ⎫
    02  SUP_TYPE             PIC X(4).   ⎬ ⓯
    02  SUP_LAG_TIME         PIC X(10).  ⎭

* subschema record type ⓰
*
* Within areas:    MARKETS
* Owner of sets:   VENDOR_SUPPLY
*
01  VENDOR.
    02  VEND_ID              PIC X(8).
    02  VEND_NAME            PIC X(40).
    02  VEND_CONTACT         PIC X(30).
    02  VEND_ADDRESS         PIC X(15) OCCURS 3 TIMES.
    02  VEND_PHONE           PIC 9(10).

* subschema set type⓱
*
SET NAME PART_SUPPLY⓲
    OWNER PART
    MEMBER SUPPLY⓳
        INSERTION AUTOMATIC⓴
        RETENTION FIXED㉑
```

ZK-6435-GE

(continued on next page)

Figure 15–53 (Cont.): PARTSS3-PROGRAM Compiler Listing



```
PARTSS3-PROGRAM                                           29-Dec-1989 14:07:26    VAX COBOL V4.3                   Page    6
Sub-schema Map                                            29-Dec-1989 14:07:17    DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)
        ORDER NEXT 22

* subschema set type
*
SET NAME VENDOR_SUPPLY
    OWNER VENDOR
    MEMBER SUPPLY
        INSERTION AUTOMATIC
        RETENTION FIXED
        ORDER NEXT

PARTSS3-PROGRAM                                           29-Dec-1989 14:07:26    VAX COBOL V4.3                   Page    7
Compilation Summary                                       29-Dec-1989 14:07:17    DEVICE:[COBOL.EXAMPLES]PARTSS3.COB;2 (1)
PROGRAM SECTIONS

    Name                                Bytes   Attributes

  0 $CODE                                   6     PIC   CON   REL   LCL   SHR   EXE   RD NOWRT Align(2)
  3 COB$NAMES_____2                        24     PIC   CON   REL   LCL   SHR NOEXE   RD NOWRT Align(2)
  4 COB$NAMES_____4                        16     PIC   CON   REL   LCL   SHR NOEXE   RD NOWRT Align(2)
  5 DBM$SSC_B                              28     PIC   CON   REL   GBL NOSHR NOEXE   RD NOWRT Align(2)
  6 DBM$UWA_B                             345     PIC   OVR   REL   GBL   SHR NOEXE   RD   WRT Align(2)

COMMAND QUALIFIERS

    COBOL /LIST/MAP PARTSS3

    /NOCOPY_LIST  /NOMACHINE_CODE  /NOCROSS_REFERENCE
    /NOANSI_FORMAT  /NOSEQUENCE_CHECK  /MAP=ALPHABETICAL
    /NOTRUNCATE  /NOAUDIT  /NOCONDITIONALS
    /CHECK=(NOPERFORM,NOBOUNDS)  /DEBUG=(NOSYMBOLS,TRACEBACK)
    /WARNINGS=(NOSTANDARD,OTHER,NOINFORMATION)
    /STANDARD=(NOSYNTAX,NOPDP11,NOV3,85)  /NOFIPS
    /LIST  /OBJECT /NODIAGNOSTICS /NOFLAGGER

STATISTICS

    Run Time:           0.92 seconds
    Elapsed Time:       8.90 seconds
    Page Faults:        545
    Dynamic Memory:     465 pages
```

ZK-6435-1-GE

15.30 Examples

This section provides programming examples of how to do the following:

- Populate a database
- Back up a database
- Access and display database information
- Create new record relationships

This chapter also provides an example of how to create a bill of materials and sample runs of some of the programming examples.

15.30.1 Populating a Database

The DBMPARTLD program in Example 15–15 loads a series of sequential data files into the PARTS database. The PARTS database consists of a NEW root file with a default extension of .ROO describing the database instance and a series of .DBS storage files containing the actual data records. PARTS is the schema relative to the current position in CDD/Plus when the program is compiled. As the DBCS inserts the records, it creates set relationships based on the PARTSS1 subschema definitions. In the DB statement PARTS and NEW can be logical names. If PARTS is not a logical name, VAX COBOL appends PARTS to CDD$DEFAULT; for example, CDD$DEFAULT.PARTS. If NEW is not a logical name, the DBCS appends .ROO as the default file type; for example, NEW.ROO.

Example 15–15: Populating a Database

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   DBMPARTLD.
**********************************************************
*                                                        *
* This program loads the PARTS database                  *
*                                                        *
**********************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX.
OBJECT-COMPUTER. VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT MAKE-FILE
                ASSIGN TO "DBM$PARTS:DBMMAKE.DAT".
        SELECT BUY-FILE
                ASSIGN TO "DBM$PARTS:DBMBUY.DAT".
        SELECT VENDOR-FILE
                ASSIGN TO "DBM$PARTS:DBMVENDOR.DAT".
        SELECT EMPLOYEE-FILE
                ASSIGN TO "DBM$PARTS:DBMEMPLOY.DAT".
        SELECT COMPONENT-FILE
                ASSIGN TO "DBM$PARTS:DBMCOMPON.DAT".
        SELECT SUPPLY-FILE
                ASSIGN TO "DBM$PARTS:DBMSUPPLY.DAT".
                SELECT DIVISION-FILE
                ASSIGN TO "DBM$PARTS:DBMSUPER.DAT".
        SELECT RESP-FOR-FILE
                ASSIGN TO "DBM$PARTS:DBMRESPON.DAT".
DATA DIVISION.
SUB-SCHEMA SECTION.
        DB PARTSS1 WITHIN PARTS FOR NEW.
FILE SECTION.
FD      MAKE-FILE
        RECORD VARYING FROM 24 TO 80 CHARACTERS.
01      MAKE-PART-RECORD.
            02 CONTROL-FIELD        PIC X.
            02 PART_ID              PIC X(8).
            02 PART_DESC            PIC X(50).
            02 PART_STATUS          PIC X(1).
            02 PART_PRICE           PIC 9(6)V9(3).
            02 PART_COST            PIC 9(6)V9(3).
            02 PART_SUPPORT         PIC X(2).
01     MAKE-CLASS-RECORD.
            02  CONTROL-FIELD       PIC X.
            02  CLASS_CODE          PIC XX.
            02  CLASS_DESC          PIC X(20).
            02  CLASS_STATUS        PIC X.
FD      BUY-FILE
        RECORD VARYING FROM 24 TO 80 CHARACTERS.
01      BUY-PART-RECORD.
            02 CONTROL-FIELD        PIC X.
            02 PART_ID              PIC X(8).
            02 PART_DESC            PIC X(50).
            02 PART_STATUS          PIC X(1).
            02 PART_PRICE           PIC 9(6)V9(3).
            02 PART_COST            PIC 9(6)V9(3).
            02 PART_SUPPORT         PIC X(2).
```

(continued on next page)

```
01      BUY-CLASS-RECORD.
            02 CONTROL-FIELD        PIC X.
            02  CLASS_CODE          PIC XX.
            02  CLASS_DESC          PIC X(20).
            02  CLASS_STATUS        PIC X.
FD      COMPONENT-FILE
        LABEL RECORDS ARE STANDARD.
01      COMPONENT-RECORD.
            02  COMP_SUB_PART      PIC X(8).
            02  COMP_OWNER_PART    PIC X(8).
            02  COMP_MEASURE       PIC X.
            02  COMP_QUANTITY      PIC 9(5).
FD      VENDOR-FILE
        LABEL RECORDS ARE STANDARD.
01      VENDOR-RECORD.
            02 VEND_ID              PIC X(8).
            02 VEND_NAME            PIC X(40).
            02 VEND_CONTACT         PIC X(30).
            02 VEND_ADD OCCURS 3 TIMES
                                    PIC X(15).
            02 VEND_PHONE           PIC 9(10).
FD      SUPPLY-FILE
        RECORD VARYING FROM 37 TO 64 CHARACTERS.
01      SUPPLY-RECORD.
            02  CONTROL-FIELD     PIC X.
            02  PART-ID           PIC X(8).
            02  VEND-NAME         PIC X(40).
            02  SUP_RATING        PIC X.
            02  SUP_TYPE          PIC X(4).
            02  SUP_LAG_TIME      PIC X(10).
01      QUOTE-RECORD.
            02  CONTROL-FIELD      PIC X.
            02  QUOTE_ID           PIC X(7).
            02  QUOTE_DATE         PIC 9(6).
            02  QUOTE_MIN_ORDER    PIC X(5).
            02  QUOTE_UNIT_PRIC    PIC 9(6)V9(3).
            02  QUOTE_QTY_PRICE    PIC 9(6)V9(3).
FD      EMPLOYEE-FILE
        LABEL RECORDS ARE STANDARD.
01      EMPLOYEE-RECORD.
            02 EMP_ID              PIC 9(5).
            02 EMP_NAME.
                03 EMP_LAST_NAME   PIC X(20).
                03 EMP_FIRST_NAME  PIC X(10).
            02 EMP_PHONE           PIC X(7).
            02 EMP_LOC             PIC X(5).
FD      DIVISION-FILE
        RECORD VARYING FROM 6 TO 26 CHARACTERS.
01      MANAGES-RECORD.
            02  CONTROL-FIELD               PIC X.
            02  GROUP_NAME                  PIC X(20).
            02  EMP_ID                      PIC 9(5).
01      CONSISTS-RECORD.
            02  CONTROL-FIELD               PIC X.
            02  EMP_ID                      PIC 9(5).
FD      RESP-FOR-FILE
        LABEL RECORDS ARE STANDARD.
01      RESP-FOR-RECORD.
            02  EMP_ID                      PIC 9(5).
            02  PART_ID                     PIC X(8).
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
WORKING-STORAGE SECTION.

77      ITEM-USED                PIC X(70).
77      STAT                     PIC 9(9) USAGE COMP.
77      DB-TEMP                  PIC 9(9) USAGE IS COMP.
77      CLASS-COUNT              PIC 999 VALUE IS 0.
77      PART-COUNT               PIC 999 VALUE IS 0.
77      COMPONENT-COUNT          PIC 999 VALUE IS 0.
77      VENDOR-COUNT             PIC 999 VALUE IS 0.
77      SUPPLY-COUNT             PIC 999 VALUE IS 0.
77      QUOTE-COUNT              PIC 999 VALUE IS 0.
77      EMPLOYEE-COUNT           PIC 999 VALUE IS 0.
77      DIVISION-COUNT           PIC 999 VALUE IS 0.

PROCEDURE DIVISION.

DECLARATIVES.
100-DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION ON OTHER.
100-PROCEDURE.
    DISPLAY "DATABASE EXCEPTION CONDITION".
    PERFORM 150-DISPLAY-MESSAGE.

150-DISPLAY-MESSAGE.
*
*  DBM$SIGNAL displays diagnostic messages based on the
*  status code in DB-CONDITION.
*
    CALL "DBM$SIGNAL".
    ROLLBACK.
    STOP RUN.
END DECLARATIVES.

DB-PROCESSING SECTION.

INITIALIZATION-ROUT.
    READY EXCLUSIVE UPDATE.

CONTROL-ROUT.
    OPEN INPUT MAKE-FILE.
    PERFORM MAKE-LOAD THRU MAKE-LOAD-END.
    CLOSE MAKE-FILE.
*    DISPLAY " ".
*    DISPLAY CLASS-COUNT, " CLASS records loaded from MAKE".
*    DISPLAY PART-COUNT, " PART records loaded from MAKE".

    OPEN INPUT BUY-FILE.
    MOVE 0 TO CLASS-COUNT.
    MOVE 0 TO PART-COUNT.
    PERFORM BUY-LOAD THRU BUY-LOAD-END.
    CLOSE BUY-FILE.
*    DISPLAY " ".
*    DISPLAY CLASS-COUNT, " CLASS records loaded from BUY".
*    DISPLAY PART-COUNT, " PART records loaded from BUY".

    OPEN INPUT VENDOR-FILE.
    PERFORM VENDOR-LOAD THRU VENDOR-LOAD-END.
    CLOSE VENDOR-FILE.
*    DISPLAY " ".
*    DISPLAY VENDOR-COUNT, " VENDOR records loaded".
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
    OPEN INPUT COMPONENT-FILE.
    PERFORM COMPONENT-LOAD THRU COMPONENT-LOAD-END.
    CLOSE COMPONENT-FILE.
*    DISPLAY " ".
*    DISPLAY COMPONENT-COUNT, " COMPONENT records loaded".

    OPEN INPUT EMPLOYEE-FILE.
    PERFORM EMPLOYEE-LOAD THRU EMPLOYEE-LOAD-END.
    CLOSE EMPLOYEE-FILE.
*    DISPLAY " ".
*    DISPLAY EMPLOYEE-COUNT, " EMPLOYEE records loaded".

    OPEN INPUT SUPPLY-FILE.
    PERFORM SUPPLY-LOAD THRU SUPPLY-LOAD-END.
    CLOSE SUPPLY-FILE.
*    DISPLAY " ".
*    DISPLAY SUPPLY-COUNT, " SUPPLY records loaded".
*    DISPLAY QUOTE-COUNT, " QUOTE records loaded".

    OPEN INPUT DIVISION-FILE.
    PERFORM DIVISION-LOAD THRU DIVISION-LOAD-END.
    CLOSE DIVISION-FILE.
*    DISPLAY " ".
*    DISPLAY DIVISION-COUNT, " DIVISION records loaded".

    OPEN INPUT RESP-FOR-FILE.
    PERFORM RESP-FOR-LOAD THRU RESP-FOR-LOAD-END.
    CLOSE RESP-FOR-FILE.

    COMMIT.
    STOP RUN.

MAKE-LOAD.
    READ MAKE-FILE AT END GO TO MAKE-LOAD-END.
    IF CONTROL-FIELD OF MAKE-PART-RECORD = "C"
        MOVE CORR MAKE-CLASS-RECORD TO CATEGORY
        STORE CATEGORY WITHIN MAKE
        ADD 1 TO CLASS-COUNT
            ELSE
            MOVE CORR MAKE-PART-RECORD TO PART
            STORE PART WITHIN MAKE
            ADD 1 TO PART-COUNT.
    GO TO MAKE-LOAD.

MAKE-LOAD-END.
    EXIT.

BUY-LOAD.
    READ BUY-FILE AT END GO TO BUY-LOAD-END.
    IF CONTROL-FIELD OF BUY-PART-RECORD = "C"
        MOVE CORR BUY-CLASS-RECORD TO CATEGORY
        STORE CATEGORY WITHIN BUY
        ADD 1 TO CLASS-COUNT
            ELSE
            MOVE CORR BUY-PART-RECORD TO PART
            STORE PART WITHIN BUY
            ADD 1 TO PART-COUNT.
    GO TO BUY-LOAD.

BUY-LOAD-END.
    EXIT.
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
VENDOR-LOAD.
    READ VENDOR-FILE AT END GO TO VENDOR-LOAD-END.
    MOVE VEND_ID OF VENDOR-RECORD TO VEND_ID OF VENDOR.
    MOVE VEND_NAME OF VENDOR-RECORD TO VEND_NAME OF VENDOR.
    MOVE VEND_CONTACT OF VENDOR-RECORD TO VEND_CONTACT OF VENDOR.
    MOVE VEND_ADD (1) TO VEND_ADDRESS (1).
    MOVE VEND_ADD (2) TO VEND_ADDRESS (2).
    MOVE VEND_ADD (3) TO VEND_ADDRESS (3).
    MOVE VEND_PHONE OF VENDOR-RECORD TO VEND_PHONE OF VENDOR.
    STORE VENDOR.
    ADD 1 TO VENDOR-COUNT.
    GO TO VENDOR-LOAD.

VENDOR-LOAD-END.
    EXIT.

COMPONENT-LOAD.
    READ COMPONENT-FILE AT END GO TO COMPONENT-LOAD-END.
    IF COMP_OWNER_PART OF COMPONENT-RECORD =
       COMP_OWNER_PART OF COMPONENT
           GO TO COMPONENT-SUB-LOAD.
    MOVE COMP_OWNER_PART OF COMPONENT-RECORD TO PART_ID OF PART.
    FIND FIRST PART WITHIN ALL_PARTS USING PART_ID OF PART
        AT END DISPLAY PART_ID OF PART,
           "COMP_OWNER_PART does not exist for COMPONENT"
           GO TO COMPONENT-LOAD.

COMPONENT-SUB-LOAD.
    MOVE COMP_SUB_PART OF COMPONENT-RECORD TO PART_ID OF PART.
    FIND FIRST PART WITHIN ALL_PARTS USING PART_ID OF PART
        RETAINING PART_USES
        AT END DISPLAY PART_ID OF PART,
            "COMP_SUB_PART does not exist for COMPONENT"
           GO TO COMPONENT-LOAD.
    MOVE CORR COMPONENT-RECORD TO COMPONENT.
    STORE COMPONENT.
    ADD 1 TO COMPONENT-COUNT.
    GO TO COMPONENT-LOAD.

COMPONENT-LOAD-END.
    EXIT.

EMPLOYEE-LOAD.
    READ EMPLOYEE-FILE AT END GO TO EMPLOYEE-LOAD-END.
    MOVE CORR EMPLOYEE-RECORD TO EMPLOYEE.
    STORE EMPLOYEE.
    ADD 1 TO EMPLOYEE-COUNT.
    GO TO EMPLOYEE-LOAD.

EMPLOYEE-LOAD-EXIT
    EXIT.
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
SUPPLY-LOAD.
    READ SUPPLY-FILE AT END GO TO SUPPLY-LOAD-END.

SUPPLY-LOAD-LOOP.
    IF CONTROL-FIELD OF SUPPLY-RECORD = "S"
        MOVE PART-ID OF SUPPLY-RECORD TO PART_ID OF PART
        FIND FIRST PART WITHIN ALL_PARTS USING PART_ID OF PART
          AT END
            DISPLAY PART_ID OF PART,
                    " PART-ID for SUPPLY does not exist"
            MOVE " " TO CONTROL-FIELD OF SUPPLY-RECORD
            PERFORM BAD-SUPPLY THRU BAD-SUPPLY-END
                    UNTIL CONTROL-FIELD OF SUPPLY-RECORD = "S"
            GO TO SUPPLY-LOAD-LOOP
          END-FIND
        MOVE VEND-NAME OF SUPPLY-RECORD TO VEND_NAME OF VENDOR
        FIND FIRST VENDOR WITHIN ALL_VENDORS USING VEND_NAME OF VENDOR
          AT END
            DISPLAY VEND_NAME OF VENDOR
                    "VEND-NAME for SUPPLY does not exist"
            MOVE " " TO CONTROL-FIELD OF SUPPLY-RECORD
            PERFORM BAD-SUPPLY THRU BAD-SUPPLY-END
                    UNTIL CONTROL-FIELD OF SUPPLY-RECORD = "S"
            GO TO SUPPLY-LOAD-LOOP
          END-FIND
        MOVE CORR SUPPLY-RECORD TO SUPPLY
        STORE SUPPLY
        ADD 1 TO SUPPLY-COUNT
        GO TO SUPPLY-LOAD
    ELSE
        MOVE CORR QUOTE-RECORD TO PR_QUOTE
        STORE PR_QUOTE
        ADD 1 TO QUOTE-COUNT
        GO TO SUPPLY-LOAD.

BAD-SUPPLY.
    READ SUPPLY-FILE AT END GO TO SUPPLY-LOAD-END.
    IF CONTROL-FIELD OF SUPPLY-RECORD = "Q"
        DISPLAY QUOTE_ID OF QUOTE-RECORD, " QUOTE_ID not stored".

BAD-SUPPLY-END.
    EXIT.

SUPPLY-LOAD-END.
    EXIT.

DIVISION-LOAD.
    READ DIVISION-FILE AT END GO TO DIVISION-LOAD-END.
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
DIVISION-LOAD-LOOP.
    IF CONTROL-FIELD OF MANAGES-RECORD = "M"
        MOVE EMP_ID OF MANAGES-RECORD TO EMP_ID OF EMPLOYEE
        FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES
                   USING EMP_ID OF EMPLOYEE
          AT END DISPLAY EMP_ID OF EMPLOYEE,
                " EMP_ID for MANAGES does not exist"
                MOVE " " TO CONTROL-FIELD OF MANAGES-RECORD
                PERFORM BAD-DIVISION THRU BAD-DIVISION-END UNTIL
                CONTROL-FIELD OF MANAGES-RECORD = "M"
                GO TO DIVISION-LOAD-LOOP
          END-FIND
        MOVE CORR MANAGES-RECORD TO WK_GROUP
        STORE WK_GROUP
        ADD 1 TO DIVISION-COUNT
        GO TO DIVISION-LOAD
    ELSE
        MOVE EMP_ID OF CONSISTS-RECORD TO EMP_ID OF EMPLOYEE
        FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING
         EMP_ID OF EMPLOYEE
          AT END DISPLAY EMP_ID OF CONSISTS-RECORD,
            " EMP_ID for CONSISTS_OF does not exist"
            GO TO DIVISION-LOAD
          END-FIND
        CONNECT EMPLOYEE TO CONSISTS_OF
        GO TO DIVISION-LOAD.

BAD-DIVISION.
    READ DIVISION-FILE AT END GO TO DIVISION-LOAD-END.
    IF CONTROL-FIELD OF MANAGES-RECORD = "C"
        DISPLAY EMP_ID OF CONSISTS-RECORD, " EMP_ID not connected".

BAD-DIVISION-END.
    EXIT.

DIVISION-LOAD-END.
    EXIT.

RESP-FOR-LOAD.
    READ RESP-FOR-FILE AT END GO TO RESP-FOR-LOAD-END.

RESP-FOR-LOAD-LOOP.
    MOVE EMP_ID OF RESP-FOR-RECORD TO EMP_ID OF EMPLOYEE.
    FETCH FIRST EMPLOYEE WITHIN ALL_EMPLOYEES
                USING EMP_ID OF EMPLOYEE
      AT END
        DISPLAY EMP_ID OF RESP-FOR-RECORD,
        " EMP_ID for RESPONSIBLE_FOR does not exist"
        GO TO RESP-FOR-LOAD.
```

(continued on next page)

Example 15–15 (Cont.): Populating a Database

```
RESP-PART-LOOP.
    MOVE PART_ID OF RESP-FOR-RECORD TO PART_ID OF PART.
    FIND FIRST PART WITHIN ALL_PARTS USING PART_ID OF PART
      AT END
        DISPLAY PART_ID OF RESP-FOR-RECORD,
        " PART_ID for RESPONSIBLE_FOR does not exist"
        GO TO RESP-FOR-LOAD.
    CONNECT PART TO RESPONSIBLE_FOR.
    READ RESP-FOR-FILE AT END GO TO RESP-FOR-LOAD-END.
    IF EMP_ID OF RESP-FOR-RECORD = EMP_ID OF EMPLOYEE
        GO TO RESP-PART-LOOP
    ELSE
        GO TO RESP-FOR-LOAD-LOOP.
RESP-FOR-LOAD-END.
    EXIT.
```

15.30.2 Backing Up a Database

The PARTSBACK program in Example 15–16 unloads all PARTS database records, independently of their pointers, into a series of sequential data files. It is the first step in restructuring and reorganizing a database. For example, after backing up the database, you can change its contents. You can also create a new version of the database including different keys or new set relationships.

The PARTS database consists of a NEW root file with a default extension of .ROO describing the database instance and a series of .DBS storage files containing the actual data records. PARTS is the schema relative to the current position in CDD/Plus when the program is compiled. In the DB statement, PARTS and NEW can be logical names. If PARTS is not a logical name, VAX COBOL appends PARTS to CDD$DEFAULT; for example, CDD$DEFAULT.PARTS. If NEW is not a logical name, the DBCS appends .ROO as the default file type; for example, NEW.ROO.

Example 15–16: Backing Up a Database

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     PARTSBACK.
*************************************************************
*
*  This program unloads the PARTS database
*
*************************************************************

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX.
OBJECT-COMPUTER.    VAX.
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.
        SELECT MAKE-FILE
                ASSIGN TO "DBM$PARTS:DBMMAKE.DAT".
        SELECT BUY-FILE
               ASSIGN TO "DBM$PARTS:DBMBUY.DAT".
        SELECT VENDOR-FILE
               ASSIGN TO "DBM$PARTS:DBMVENDOR.DAT".
        SELECT EMPLOYEE-FILE
               ASSIGN TO "DBM$PARTS:DBMEMPLOY.DAT".
        SELECT COMPONENT-FILE
               ASSIGN TO "DBM$PARTS:DBMCOMPON.DAT".
        SELECT SUPPLY-FILE
               ASSIGN TO "DBM$PARTS:DBMSUPPLY.DAT".
        SELECT DIVISION-FILE
               ASSIGN TO "DBM$PARTS:DBMSUPER.DAT".
        SELECT RESP-FOR-FILE
               ASSIGN TO "DBM$PARTS:DBMRESPON.DAT".

DATA DIVISION.

SUB-SCHEMA SECTION.
DB PARTSS1 WITHIN PARTS FOR NEW.

FILE SECTION.

FD      MAKE-FILE
        RECORD VARYING FROM 24 TO 80 CHARACTERS.
01      MAKE-PART-RECORD.
        02 CONTROL-FIELD        PIC X.
        02 PART_ID              PIC X(8).
        02 PART_DESC            PIC X(50).
        02 PART_STATUS          PIC X(1).
        02 PART_PRICE           PIC 9(6)V9(3).
        02 PART_COST            PIC 9(6)V9(3).
        02 PART_SUPPORT         PIC X(2).
01      MAKE-CLASS-RECORD.
        02  CONTROL-FIELD       PIC X.
        02  CLASS_CODE          PIC XX.
        02  CLASS_DESC          PIC X(20).
        02  CLASS_STATUS        PIC X.

FD      BUY-FILE
        RECORD VARYING FROM 24 TO 80 CHARACTERS.
01      BUY-PART-RECORD.
        02 CONTROL-FIELD        PIC X.
        02 PART_ID              PIC X(8).
        02 PART_DESC            PIC X(50).
        02 PART_STATUS          PIC X(1).
        02 PART_PRICE           PIC 9(6)V9(3).
        02 PART_COST            PIC 9(6)V9(3).
        02 PART_SUPPORT         PIC X(2).
01      BUY-CLASS-RECORD.
        02 CONTROL-FIELD        PIC X.
        02 CLASS_CODE           PIC XX.
        02 CLASS_DESC           PIC X(20).
        02 CLASS_STATUS         PIC X.
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
FD      COMPONENT-FILE
        LABEL RECORDS ARE STANDARD.
01      COMPONENT-RECORD.
        02  COMP_SUB_PART      PIC X(8).
        02  COMP_OWNER_PART    PIC X(8).
        02  COMP_MEASURE       PIC X.
        02  COMP_QUANTITY      PIC 9(5).

FD      VENDOR-FILE
        LABEL RECORDS ARE STANDARD.
01      VENDOR-RECORD.
        02 VEND_ID                     PIC X(8).
        02 VEND_NAME                   PIC X(40).
        02 VEND_CONTACT                PIC X(30).
        02 VEND_ADDRESS OCCURS 3 TIMES PIC X(15).
        02 VEND_PHONE                  PIC 9(10).

FD      SUPPLY-FILE
        RECORD VARYING FROM 37 TO 64 CHARACTERS.
01      SUPPLY-RECORD.
        02  CONTROL-FIELD      PIC X.
        02  PART-ID            PIC X(8).
        02  VEND-NAME          PIC X(40).
        02  SUP_RATING         PIC X.
        02  SUP_TYPE           PIC X(4).
        02  SUP_LAG_TIME       PIC X(10).
01      QUOTE-RECORD.
        02  CONTROL-FIELD      PIC X.
        02  QUOTE_ID           PIC X(7).
        02  QUOTE_DATE         PIC 9(6).
        02  QUOTE_MIN_ORDER    PIC X(5).
        02  QUOTE_UNIT_PRIC    PIC 9(6)V9(3).
        02  QUOTE_QTY_PRICE    PIC 9(6)V9(3).

FD      EMPLOYEE-FILE
        LABEL RECORDS ARE STANDARD.
01      EMPLOYEE-RECORD.
        02 EMP_ID                      PIC 9(5).
        02 EMP_NAME.
                03 EMP_LAST_NAME       PIC X(20).
                03 EMP_FIRST_NAME      PIC X(10).
        02 EMP_PHONE                   PIC X(7).
        02 EMP_LOC                     PIC X(5).

FD      DIVISION-FILE
        RECORD VARYING FROM 6 TO 26 CHARACTERS.
01      MANAGES-RECORD.
        02  CONTROL-FIELD       PIC X.
        02  GROUP_NAME          PIC X(20).
        02  EMP_ID              PIC 9(5).
01      CONSISTS-RECORD.
        02  CONTROL-FIELD       PIC X.
        02  EMP_ID              PIC 9(5).

FD      RESP-FOR-FILE
        LABEL RECORDS ARE STANDARD.
01      RESP-FOR-RECORD.
        02  EMP_ID             PIC 9(5).
        02  PART_ID            PIC X(8).
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
WORKING-STORAGE SECTION.

77      CLASS-COUNT             PIC 999 VALUE IS 0.
77      PART-COUNT              PIC 999 VALUE IS 0.
77      COMPONENT-COUNT         PIC 999 VALUE IS 0.
77      VENDOR-COUNT            PIC 999 VALUE IS 0.
77      SUPPLY-COUNT            PIC 999 VALUE IS 0.
77      QUOTE-COUNT             PIC 999 VALUE IS 0.
77      EMPLOYEE-COUNT          PIC 999 VALUE IS 0.

PROCEDURE DIVISION.

DECLARATIVES.
100-DATABASE-EXCEPTIONS SECTION.
    USE FOR DB-EXCEPTION ON OTHER.
100-PROCEDURE.
    DISPLAY "DATABASE EXCEPTION CONDITION".
    PERFORM 150-DISPLAY-MESSAGE.

150-DISPLAY-MESSAGE.
*
* DBM$SIGNAL displays diagnostic messages based on the
* status code in DB-CONDITION.
*
    CALL "DBM$SIGNAL".
    ROLLBACK.
    STOP RUN.
END DECLARATIVES.

DB-PROCESSING SECTION.

INITIALIZATION-ROUT.
    READY PROTECTED.

CONTROL-ROUT.
    OPEN OUTPUT COMPONENT-FILE, SUPPLY-FILE.
    OPEN OUTPUT MAKE-FILE.
    PERFORM MAKE-UNLOAD THRU MAKE-UNLOAD-END.
    CLOSE MAKE-FILE.
    DISPLAY " ".
    DISPLAY CLASS-COUNT, " CLASS records unloaded from MAKE".
    DISPLAY PART-COUNT, " PART records unloaded from MAKE".

    OPEN OUTPUT BUY-FILE.
    MOVE 0 TO CLASS-COUNT.
    MOVE 0 TO PART-COUNT.
    PERFORM BUY-UNLOAD THRU BUY-UNLOAD-END.
    CLOSE BUY-FILE, COMPONENT-FILE, SUPPLY-FILE.
    DISPLAY " ".
    DISPLAY CLASS-COUNT, " CLASS records unloaded from BUY".
    DISPLAY PART-COUNT, " PART records unloaded from BUY".
    DISPLAY " ".
    DISPLAY SUPPLY-COUNT, " SUPPLY records unloaded".
    DISPLAY QUOTE-COUNT, " QUOTE records unloaded".
    DISPLAY COMPONENT-COUNT " COMPONENT records unloaded".

    OPEN OUTPUT VENDOR-FILE.
    PERFORM VENDOR-UNLOAD THRU VENDOR-UNLOAD-END.
    CLOSE VENDOR-FILE.
    DISPLAY " ".
    DISPLAY VENDOR-COUNT, " VENDOR records unloaded".
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
    OPEN OUTPUT EMPLOYEE-FILE, RESP-FOR-FILE, DIVISION-FILE.
    PERFORM EMPLOYEE-UNLOAD THRU EMPLOYEE-UNLOAD-END.
    CLOSE EMPLOYEE-FILE, RESP-FOR-FILE, DIVISION-FILE.
    DISPLAY " ".
    DISPLAY EMPLOYEE-COUNT, " EMPLOYEE records unloaded".

    COMMIT.
    STOP RUN.

MAKE-UNLOAD.
    FETCH NEXT CATEGORY WITHIN MAKE
        AT END GO TO MAKE-UNLOAD-END.
    MOVE "C" TO CONTROL-FIELD OF MAKE-CLASS-RECORD.
    MOVE CORR CATEGORY TO MAKE-CLASS-RECORD.
    ADD 1 TO CLASS-COUNT.
    WRITE MAKE-CLASS-RECORD.

MAKE-PART-LOOP.
    FETCH NEXT PART WITHIN CLASS_PART RETAINING REALM
        AT END GO TO MAKE-UNLOAD.
    MOVE "P" TO CONTROL-FIELD OF MAKE-PART-RECORD.
    MOVE CORR PART TO MAKE-PART-RECORD.
    ADD 1 TO PART-COUNT.
    WRITE MAKE-PART-RECORD.
    PERFORM COMPONENT-SUPPLY-UNLOAD THRU
            COMPONENT-SUPPLY-UNLOAD-END.
    GO TO MAKE-PART-LOOP.

MAKE-UNLOAD-END.
    EXIT.

BUY-UNLOAD.
    FETCH NEXT CATEGORY WITHIN BUY
        AT END GO TO BUY-UNLOAD-END.
    MOVE "C" TO CONTROL-FIELD OF BUY-CLASS-RECORD.
    MOVE CORR CATEGORY TO BUY-CLASS-RECORD.
    ADD 1 TO CLASS-COUNT.
    WRITE BUY-CLASS-RECORD.

BUY-PART-LOOP.
    FETCH NEXT PART WITHIN CLASS_PART RETAINING REALM
        AT END GO TO BUY-UNLOAD.
    MOVE "P" TO CONTROL-FIELD OF BUY-PART-RECORD.
    MOVE CORR PART TO BUY-PART-RECORD.
    ADD 1 TO PART-COUNT.
    WRITE BUY-PART-RECORD.
    PERFORM COMPONENT-SUPPLY-UNLOAD THRU
            COMPONENT-SUPPLY-UNLOAD-END.
    GO TO BUY-PART-LOOP.

BUY-UNLOAD-END.
    EXIT.

COMPONENT-SUPPLY-UNLOAD.

COMPONENT-UNLOAD.
    FETCH NEXT COMPONENT WITHIN PART_USES RETAINING REALM
        AT END GO TO SUPPLY-QUOTE-LOOP.
    MOVE CORR COMPONENT TO COMPONENT-RECORD.
    ADD 1 TO COMPONENT-COUNT.
    WRITE COMPONENT-RECORD.
    GO TO COMPONENT-UNLOAD.
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
SUPPLY-QUOTE-LOOP.
    FETCH NEXT WITHIN PART_INFO RETAINING REALM
        AT END GO TO COMPONENT-SUPPLY-UNLOAD-END.
    IF DB-CURRENT-RECORD-NAME = "PR_QUOTE" THEN
        MOVE CORR PR_QUOTE TO QUOTE-RECORD
        MOVE "Q" TO CONTROL-FIELD OF QUOTE-RECORD
        ADD 1 TO QUOTE-COUNT
        WRITE QUOTE-RECORD
        GO TO SUPPLY-QUOTE-LOOP
    ELSE
        MOVE CORR SUPPLY TO SUPPLY-RECORD
        FETCH OWNER WITHIN VENDOR_SUPPLY
        MOVE "S" TO CONTROL-FIELD OF SUPPLY-RECORD
        MOVE VEND_NAME OF VENDOR TO VEND-NAME OF SUPPLY-RECORD
        MOVE PART_ID OF PART TO PART-ID OF SUPPLY-RECORD
        ADD 1 TO SUPPLY-COUNT
        WRITE SUPPLY-RECORD
        GO TO SUPPLY-QUOTE-LOOP.

COMPONENT-SUPPLY-UNLOAD-END.
    EXIT.

VENDOR-UNLOAD.
    FREE CURRENT WITHIN MARKET.

VENDOR-UNLOAD-LOOP.
    FETCH NEXT VENDOR WITHIN MARKET
        AT END GO TO VENDOR-UNLOAD-END.
    ADD 1 TO VENDOR-COUNT.
    MOVE VEND_ID OF VENDOR TO VEND_ID OF VENDOR-RECORD.
    MOVE VEND_NAME OF VENDOR TO VEND_NAME OF VENDOR-RECORD.
    MOVE VEND_CONTACT OF VENDOR TO VEND_CONTACT OF VENDOR-RECORD.
    MOVE VEND_ADDRESS OF VENDOR (1) TO
         VEND_ADDRESS OF VENDOR-RECORD (1).
    MOVE VEND_ADDRESS OF VENDOR (2) TO
         VEND_ADDRESS OF VENDOR-RECORD (2).
    MOVE VEND_ADDRESS OF VENDOR (3) TO
         VEND_ADDRESS OF VENDOR-RECORD (3).
    MOVE VEND_PHONE OF VENDOR TO VEND_PHONE OF VENDOR-RECORD.
    WRITE VENDOR-RECORD.
    GO TO VENDOR-UNLOAD-LOOP.

VENDOR-UNLOAD-END.
    EXIT.

EMPLOYEE-UNLOAD.
    FETCH NEXT EMPLOYEE WITHIN ALL_EMPLOYEES
        AT END GO TO EMPLOYEE-UNLOAD-END.
    MOVE CORR EMPLOYEE TO EMPLOYEE-RECORD.
    ADD 1 TO EMPLOYEE-COUNT.
    WRITE EMPLOYEE-RECORD.

DIVISION-UNLOAD.
    FETCH NEXT WITHIN MANAGES
        AT END GO TO RESP-UNLOAD.
    MOVE EMP_ID OF EMPLOYEE TO EMP_ID OF MANAGES-RECORD.
    MOVE GROUP_NAME OF WK_GROUP TO GROUP_NAME OF MANAGES-RECORD.
    MOVE "M" TO CONTROL-FIELD OF MANAGES-RECORD.
    WRITE MANAGES-RECORD.
```

(continued on next page)

Example 15–16 (Cont.): Backing Up a Database

```
CONSISTS-UNLOAD.
    FETCH NEXT WITHIN CONSISTS_OF RETAINING MANAGES ALL_EMPLOYEES
        AT END GO TO DIVISION-UNLOAD.
    MOVE "C" TO CONTROL-FIELD OF CONSISTS-RECORD.
    MOVE EMP_ID OF EMPLOYEE TO  EMP_ID OF CONSISTS-RECORD.
    WRITE CONSISTS-RECORD.
    GO TO CONSISTS-UNLOAD.

RESP-UNLOAD.
    FETCH CURRENT WITHIN ALL_EMPLOYEES.
RESP-UNLOAD-LOOP.
    FETCH NEXT WITHIN RESPONSIBLE_FOR
        AT END GO TO EMPLOYEE-UNLOAD.
    MOVE PART_ID OF PART TO PART_ID OF RESP-FOR-RECORD.
    MOVE EMP_ID OF EMPLOYEE TO EMP_ID OF RESP-FOR-RECORD.
    WRITE RESP-FOR-RECORD.
    GO TO RESP-UNLOAD-LOOP.
EMPLOYEE-UNLOAD-END.
    EXIT.
```

15.30.3 Accessing and Displaying Database Information

The PARTBOM program in Example 15–17 produces a report of subcomponents (bill of materials) for a part in the PARTS database. Refer to Figure 15–24 for an explanation of the report and Section 15.30.6 for a sample listing.

Example 15–17: Accessing and Displaying Database Information

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PARTBOM.
AUTHOR.  ME.
INSTALLATION. HERE.
DATE-WRITTEN. TODAY.
SECURITY. NONE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX.
OBJECT-COMPUTER. VAX.


DATA DIVISION.
SUB-SCHEMA SECTION.

DB    PARTSS1 WITHIN PARTS FOR NEW.
LD    KEEP-COMPONENT.

WORKING-STORAGE SECTION.

01      INPUT-REC               PIC X(80).

01      INDENT-LEVEL            PIC 9(02)  VALUE 40.
01      END-OF-COLLECTION       PIC 9(01)  VALUE 0.
        88  END-COLLECTION                 VALUE 1.

01      INDENT-TREE.
        02  INDENT-TREE-ARRAY   PIC X(03)  OCCURS 1 TO 40 TIMES
                                DEPENDING ON INDENT-LEVEL.
```

(continued on next page)

Example 15–17 (Cont.): Accessing and Displaying Database Information

```
PROCEDURE DIVISION.

INITIALIZATION.
    READY   MAKE, BUY EXCLUSIVE RETRIEVAL.
    MOVE    ALL "|  " TO INDENT-TREE.

SOLICIT-INPUT.
    MOVE ZERO TO END-OF-COLLECTION.
    DISPLAY    " ".
    DISPLAY    "Enter PART_ID> " WITH NO ADVANCING.
    MOVE    SPACES TO INPUT-REC.
    ACCEPT PART_ID
        AT END GO TO PARTBOM-DONE.
    FETCH    FIRST PART WITHIN ALL_PARTS USING PART_ID
        AT END DISPLAY "***  Part number ",

                                PART_ID, " not found.  ***"
               GO TO SOLICIT-INPUT.
    DISPLAY    " ".
    DISPLAY    " ".
    DISPLAY "+-----------------------------------+".
    DISPLAY "| Parts Bill of Materials Explosion |".
    DISPLAY "|         (COBOL Version)           |".
    DISPLAY "|        Part-id: " PART_ID "       |".
    DISPLAY "+-----------------------------------+".
    DISPLAY " ".
    DISPLAY " ".
    DISPLAY " ".
    DISPLAY    PART_ID, " - ", PART_DESC
    MOVE ZERO TO INDENT-LEVEL.
    FREE ALL FROM KEEP-COMPONENT.
    PERFORM PARTBOM-LOOP THRU PARTBOM-LOOP-EXIT
        UNTIL END-COLLECTION.
    GO TO SOLICIT-INPUT.

PARTBOM-DONE.
    COMMIT.
    DISPLAY " ".
    DISPLAY "END COBOL PARTBOM.".
    STOP RUN.

PARTBOM-LOOP.
    FIND NEXT COMPONENT WITHIN PART_USES
        AT END PERFORM POP-COMPONENT THRU POP-COMPONENT-EXIT
               GO TO PARTBOM-LOOP-EXIT.
    KEEP CURRENT USING KEEP-COMPONENT.
    ADD 1 TO INDENT-LEVEL.
    FIND OWNER PART_USED_ON.
    GET PART_ID, PART_DESC.
    DISPLAY INDENT-TREE, PART_ID, " - ", PART_DESC.

PARTBOM-LOOP-EXIT.
    EXIT.
```

(continued on next page)

Example 15–17 (Cont.): Accessing and Displaying Database Information

```
POP-COMPONENT.
    FIND LAST WITHIN KEEP-COMPONENT
        AT END MOVE 1 TO END-OF-COLLECTION
               GO TO POP-COMPONENT-EXIT.
    FREE LAST WITHIN KEEP-COMPONENT.
    SUBTRACT 1 FROM INDENT-LEVEL.

POP-COMPONENT-EXIT.
    EXIT.
```

15.30.4 PARTBOM Sample Run

Example 15–18 displays a sample run of the PARTBOM program in Example 15–17.

Example 15–18: Sample Run of the PARTBOM Program

```
Enter PARTID> BT163456
                 +-----------------------------------+
                 | Parts Bill of Materials Explosion |
                 |          (COBOL Version)          |
                 |         Part-id: BT163456         |
                 +-----------------------------------+
BT163456 - VT100
|  BU355678 - VT100 NON REFLECTIVE SCREEN
|  BU345670 - TERMINAL TABLE VT100
|  |  AZ345678 - 3/4 INCH SCREWS
|  |  AZ167890 - 1/2 INCH SCREWS
|  |  AZ517890 - 1/4 INCH BOLTS
|  |  AZ012345 - 3 INCH NAILS
|  |  AS234567 - 1/4 INCH TACKS
|  |  AS901234 - 3/8 INCH SCREWS
|  |  AS456789 - 4/5 INCH CLAMP
|  |  AS560890 - 1 INCH CLAMP
|  BU456789 - PLASTIC KEY ALPHA.
|  BU345438 - PLASTIC KEY NUM.
|  BU234567 - VIDEO TUBE
|  |  AZ345678 - 3/4 INCH SCREWS
|  |  AZ789012 - 3/8 INCH BOLTS
|  |  AS234567 - 1/4 INCH TACKS
|  |  AS560890 - 1 INCH CLAMP
|  BU890123 - VT100 HOUSING
|  BU876778 - VT100 SCREEN
|  AZ345678 - 3/4 INCH SCREWS
|  AZ567890 - 1/4 INCH SCREWS
|  AZ789012 - 3/8 INCH BOLTS
|  AS901234 - 3/8 INCH SCREWS
|  AS890123 - 3/4 INCH ELECTRICAL TAPE
Enter PARTID> ctrl/z
END COBOL PARTBOM.
```

15.30.5 Creating Relationships Between Records of the Same Type

The STOOL program in Example 15–19 illustrates how to create a relationship between records of the same type. It loads and connects the parts example discussed in Section 15.18.2.2 and produces a parts breakdown report illustrating the relationships. Section 15.30.6 contains the sample report.

Example 15–19: Creating Relationships Between Records of the Same Type

```
IDENTIFICATION DIVISION.
PROGRAM-ID. STOOL.
DATA DIVISION.
SUB-SCHEMA SECTION.
DB  PARTSS1 WITHIN  PARTS FOR "NEW.ROO".
LD  KEEP-COMPONENT.
WORKING-STORAGE SECTION.
01  DB-ERROR-CHECK       PIC 9.
    88  DB-ERROR         VALUE 1.
    88  DB-OK            VALUE 0.
01  DB-COND              PIC 9(9).
01  DB-ID                PIC 9(4).

PROCEDURE DIVISION.
A000-BEGIN.
    READY USAGE-MODE IS CONCURRENT UPDATE.
    MOVE 0 TO DB-ERROR-CHECK.
    PERFORM B000-STORE-PARTS THROUGH
            B300-BUILD-AND-STORE-STOOL-LEG.
    IF DB-OK PERFORM C000-STORE-COMPONENTS
                     THRU 800-VERIFY-ROUTINE.

A100-EOJ.
*   IF DB-ERROR
    ROLLBACK ON ERROR DISPLAY "Error on ROLLBACK"
             PERFORM 900-DISPLAY-DB-CONDITION
             END-ROLLBACK
    DISPLAY "End of Job".
    STOP RUN.

B000-STORE-PARTS.
    FIND FIRST PART ON ERROR
         DISPLAY "Positioning to first part is unsuccessful"
         PERFORM 900-DISPLAY-DB-CONDITION
         MOVE 1 TO DB-ERROR-CHECK.

B100-BUILD-AND-STORE-STOOL.
    MOVE "SAMP1" TO PART_ID.
    MOVE "STOOL" TO PART_DESC.
    MOVE "G"     TO PART_STATUS.
    MOVE 11      TO PART_PRICE.
    MOVE 6       TO PART_COST.
    MOVE SPACES  TO PART_SUPPORT.
    IF DB-OK STORE PART ON ERROR
          DISPLAY "B100 Error in storing STOOL"
          PERFORM 900-DISPLAY-DB-CONDITION
          MOVE 1 TO DB-ERROR-CHECK.
```

(continued on next page)

Example 15–19 (Cont.): Creating Relationships Between Records of the Same Type

```
B200-BUILD-AND-STORE-STOOL-SEAT.
    MOVE "SAMP2"      TO PART_ID.
    MOVE "STOOL SEAT" TO PART_DESC.
    MOVE "G"          TO PART_STATUS.
    MOVE 3            TO PART_PRICE.
    MOVE 2            TO PART_COST.
    MOVE SPACES       TO PART_SUPPORT.
    IF DB-OK STORE PART ON ERROR
          DISPLAY "B200 Error in storing STOOL SEAT"
          PERFORM 900-DISPLAY-DB-CONDITION
          MOVE 1 TO DB-ERROR-CHECK.

B300-BUILD-AND-STORE-STOOL-LEG.
    MOVE "SAMP3"      TO PART_ID.
    MOVE "STOOL LEGS" TO PART_DESC.
    MOVE "G"          TO PART_STATUS.
    MOVE 2            TO PART_PRICE.
    MOVE 1            TO PART_COST.
    MOVE SPACES       TO PART_SUPPORT.
    IF DB-OK STORE PART ON ERROR
          DISPLAY "B300 Error in storing STOOL LEGS"
          PERFORM 900-DISPLAY-DB-CONDITION
          MOVE 1 TO DB-ERROR-CHECK.

C000-STORE-COMPONENTS.
    MOVE "STOOL" TO PART_DESC.

C100-FIND-STOOL.
    FIND FIRST PART USING PART_DESC ON ERROR
         DISPLAY "C000 Error in finding STOOL"
         PERFORM 900-DISPLAY-DB-CONDITION
         MOVE 1 TO DB-ERROR-CHECK.
    MOVE "STOOL SEAT" TO PART_DESC.

C200-FIND-STOOL-SEAT.
    IF DB-OK
       FIND FIRST PART USING PART_DESC RETAINING PART_USES
         ON ERROR
             DISPLAY "C000 Error in finding STOOL SEAT"
             PERFORM 900-DISPLAY-DB-CONDITION
             MOVE 1 TO DB-ERROR-CHECK.

C300-CONNECT-COMPONENT-1.
    MOVE "SAMP2" TO COMP_SUB_PART.
    MOVE "SAMP1" TO COMP_OWNER_PART.
    MOVE "U"     TO COMP_MEASURE.
    MOVE 1       TO COMP_QUANTITY.
    IF DB-OK
       STORE COMPONENT RETAINING PART_USES
         ON ERROR
             DISPLAY "C000 Error in storing first component"
             PERFORM 900-DISPLAY-DB-CONDITION
             MOVE 1 TO DB-ERROR-CHECK.

C400-FIND-STOOL-LEGS.
    MOVE "STOOL LEGS" TO PART_DESC.
    IF DB-OK
       FIND FIRST PART USING PART_DESC RETAINING PART_USES
         ON ERROR
             DISPLAY "C000 Error in finding STOOL LEGS"
             PERFORM 900-DISPLAY-DB-CONDITION
             MOVE 1 TO DB-ERROR-CHECK.
```

(continued on next page)

Example 15–19 (Cont.): Creating Relationships Between Records of the Same Type

```
C500-CONNECT-COMPONENT-4.
    MOVE "SAMP3" TO COMP_SUB_PART.
    MOVE "SAMP1" TO COMP_OWNER_PART.
    MOVE "U"     TO COMP_MEASURE.
    MOVE 4       TO COMP_QUANTITY.
    IF DB-OK
       STORE COMPONENT
         ON ERROR
             DISPLAY "C000 Error in storing second component"
             PERFORM 900-DISPLAY-DB-CONDITION
             MOVE 1 TO DB-ERROR-CHECK.

800-VERIFY-ROUTINE.
    CALL "PARTBOM".

900-DISPLAY-DB-CONDITION.
    MOVE DB-CONDITION                  TO DB-COND.
    MOVE DB-CURRENT-RECORD-ID          TO DB-ID.
    DISPLAY "DB-CONDITION            - ", DB-COND.
    DISPLAY "DB-CURRENT-RECORD-NAME  - ",
                             DB-CURRENT-RECORD-NAME.
    DISPLAY "DB-CURRENT-RECORD-ID    - ", DB-ID.
    CALL "DBM$SIGNAL".

IDENTIFICATION DIVISION.
PROGRAM-ID.  PARTBOM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT  INPUT-FILE ASSIGN TO "SYS$COMMAND".

DATA DIVISION.
SUB-SCHEMA SECTION.
*  DB PARTSS1 WITHIN  PARTS FOR "NEW.ROO".

FILE SECTION.
FD      INPUT-FILE
        LABEL RECORDS ARE STANDARD
        DATA  RECORD  IS  INPUT-REC.
01      INPUT-REC               PIC X(80).

WORKING-STORAGE SECTION.
01      INDENT-LEVEL            PIC 9(02)  VALUE 40.
01      DBM$_END                PIC 9(09)  COMP
                                VALUE EXTERNAL DBM$_END.
01      END-OF-COLLECTION       PIC 9(01)  VALUE 0.
        88  END-COLLECTION                 VALUE 1.
01      INDENT-TREE.
        02  INDENT-TREE-ARRAY   PIC X(03)
                                OCCURS 1 TO 40 TIMES
                                DEPENDING  ON INDENT-LEVEL.

PROCEDURE DIVISION.

INITIALIZATION.
    OPEN INPUT  INPUT-FILE.
    MOVE ALL "|  " TO INDENT-TREE.
```

(continued on next page)

Example 15–19 (Cont.): Creating Relationships Between Records of the Same Type

```
SOLICIT-INPUT.
    MOVE ZERO TO END-OF-COLLECTION.
    DISPLAY " ".
    DISPLAY "Enter PART_ID> " WITH NO ADVANCING.
    MOVE SPACES TO INPUT-REC.
    READ INPUT-FILE INTO PART_ID
        AT END GO TO PARTBOM-DONE.
    FETCH FIRST PART WITHIN ALL_PARTS USING PART_ID
        AT END DISPLAY "*** Part number ",
                                PART_ID, " not found.  ***"
               GO TO SOLICIT-INPUT.
    DISPLAY    " ".
    DISPLAY    " ".
    DISPLAY
    DISPLAY "+-----------------------------------+".
    DISPLAY "| Parts Bill of Materials Explosion |".
    DISPLAY "|         (COBOL Version)          |".
    DISPLAY "|        Part-id: " PART_ID "      |".
    DISPLAY "+-----------------------------------+".
    DISPLAY " ".
    DISPLAY " ".
    DISPLAY " ".
    DISPLAY    PART_ID, " - ", PART_DESC
    MOVE ZERO TO INDENT-LEVEL.
    FREE ALL FROM KEEP-COMPONENT.
    PERFORM PARTBOM-LOOP THRU PARTBOM-LOOP-EXIT
        UNTIL END-COLLECTION.
    GO TO SOLICIT-INPUT.

PARTBOM-DONE.
    CLOSE INPUT-FILE.
    EXIT PROGRAM.

PARTBOM-LOOP.
    FIND NEXT COMPONENT WITHIN PART_USES
         AT END PERFORM POP-COMPONENT
                       THRU POP-COMPONENT-EXIT
         GO TO PARTBOM-LOOP-EXIT.
    KEEP CURRENT USING KEEP-COMPONENT.
    ADD 1 TO INDENT-LEVEL.
    FIND OWNER PART_USED_ON.
    GET PART_ID, PART_DESC.
    DISPLAY INDENT-TREE, PART_ID, " - ", PART_DESC.

PARTBOM-LOOP-EXIT.
    EXIT.

POP-COMPONENT.
    FIND    LAST WITHIN KEEP-COMPONENT
        AT END MOVE 1 TO END-OF-COLLECTION
               GO TO POP-COMPONENT-EXIT.
    FREE    LAST WITHIN KEEP-COMPONENT.
    SUBTRACT 1 FROM INDENT-LEVEL.

POP-COMPONENT-EXIT.
    EXIT.
END PROGRAM PARTBOM.
END PROGRAM STOOL.
```

15.30.6 STOOL Program Parts Breakdown Report—Sample Run

This is the report output by the STOOL program in Example 15–19.

```
Enter PARTID> SAMP1 RET

+-----------------------------------+
| Parts Bill of Materials Explosion |
|          (COBOL Version)          |
|          Part-id: SAMP1           |
+-----------------------------------+

SAMP1    - STOOL
|  SAMP3    - STOOL LEGS
|  SAMP2    - STOOL SEAT

Enter PARTID> ctrl/z
End of Job
```

15.30.7 Creating New Record Relationships

The PERSONNEL-UPDATE program in Example 15–20 creates the records and relationships described in Section 15.18.2.3. It directly contains two other programs: PROMOTION-UPDATE and PERSONNEL-REPORT. PROMOTION-UPDATE is directly contained by PERSONNEL-UPDATE. It changes the record relationships created by PERSONNEL-UPDATE. PERSONNEL-REPORT is also directly contained by PERSONNEL-UPDATE. It generates one report showing the record relationships just after creation by PERSONNEL-UPDATE and another report showing the new record relationships. PERSONNEL-REPORT is a Report Writer program. Section 15.30.7.1 and Section 15.30.7.2 each contain a report generated by the PERSONNEL-UPDATE program.

Example 15–20: Creating New Record Relationships

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PERSONNEL-UPDATE.

DATA DIVISION.
SUB-SCHEMA SECTION.
DB PARTSS1 WITHIN PARTS FOR "NEW.ROO".
LD KEEPSUPER.
LD KEEP-EMPLOYEE.

WORKING-STORAGE SECTION.
01  ANSWER      PIC X.
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
PROCEDURE DIVISION.
A000-BEGIN.
    READY USAGE-MODE IS UPDATE.
    PERFORM A100-EMPLOYEE-LOAD.
    PERFORM A200-CONNECTING-TO-CONSISTS-OF.
    DISPLAY "Employees and groups are loaded".
    DISPLAY "Personnel Report before update ..."
    CALL "PERSONNEL-REPORT".
    DISPLAY "Press your carriage return key to continue".
    ACCEPT ANSWER.
    CALL "PROMOTION-UPDATE".
    DISPLAY "Promotions completed".
    DISPLAY "Press your carriage return key to continue".
    ACCEPT ANSWER.
    DISPLAY "Personnel Report after update ...".
    CALL "PERSONNEL-REPORT".

A010-EOJ.
    ROLLBACK.
    DISPLAY "End of PERSONNEL-UPDATE".
    STOP RUN.

A100-EMPLOYEE-LOAD.
    MOVE 10500     TO EMP_ID.
    MOVE "HOWELL"  TO EMP_LAST_NAME.
    MOVE "JOHN"    TO EMP_FIRST_NAME.
    MOVE 1111111   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 08400     TO EMP_ID.
    MOVE "NOYCE"   TO EMP_LAST_NAME.
    MOVE "BILL"    TO EMP_FIRST_NAME.
    MOVE 2222222   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 06600     TO EMP_ID.
    MOVE "MOORE"   TO EMP_LAST_NAME.
    MOVE "BRUCE"   TO EMP_FIRST_NAME.
    MOVE 3333333   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 01000     TO EMP_ID.
    MOVE "RAVAN"   TO EMP_LAST_NAME.
    MOVE "JERRY"   TO EMP_FIRST_NAME.
    MOVE 5555555   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 04000     TO EMP_ID.
    MOVE "BURLEW"  TO EMP_LAST_NAME.
    MOVE "THOMAS"  TO EMP_FIRST_NAME.
    MOVE 6666666   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 07000     TO EMP_ID.
    MOVE "NEILS"   TO EMP_LAST_NAME.
    MOVE "ALBERT"  TO EMP_FIRST_NAME.
    MOVE 7777777   TO EMP_PHONE.
    MOVE "N.H."    TO EMP_LOC.
    STORE EMPLOYEE.
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
    MOVE 05000    TO EMP_ID.
    MOVE "KLEIN"  TO EMP_LAST_NAME.
    MOVE "DON"    TO EMP_FIRST_NAME.
    MOVE 8888888  TO EMP_PHONE.
    MOVE "N.H."   TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 02000    TO EMP_ID.
    MOVE "DEANE"  TO EMP_LAST_NAME.
    MOVE "FRANK"  TO EMP_FIRST_NAME.
    MOVE 9999999  TO EMP_PHONE.
    MOVE "N.H."   TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 01400    TO EMP_ID.
    MOVE "RILEY"  TO EMP_LAST_NAME.
    MOVE "GEORGE" TO EMP_FIRST_NAME.
    MOVE 1234567  TO EMP_PHONE.
    MOVE "N.H."   TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 05500    TO EMP_ID.
    MOVE "BAKER"  TO EMP_LAST_NAME.
    MOVE "DOUGH"  TO EMP_FIRST_NAME.
    MOVE 7654321  TO EMP_PHONE.
    MOVE "N.H."   TO EMP_LOC.
    STORE EMPLOYEE.

    MOVE 07400    TO EMP_ID.
    MOVE "FIFER"  TO EMP_LAST-NAME.
    MOVE "MIKE"   TO EMP_FIRST_NAME.
    MOVE 1212121  TO EMP_PHONE.
    MOVE "N.H."   TO EMP_LOC.
    STORE EMPLOYEE.

A200-CONNECTING-TO-CONSISTS-OF.
    MOVE 10500 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    MOVE "A" TO GROUP_NAME.
    STORE WK_GROUP.

    MOVE 08400 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 06600 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 08400 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    MOVE "B1" TO GROUP_NAME.
    STORE WK_GROUP.

    MOVE 01000 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 04000 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 07000 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
    MOVE 06600 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    MOVE "B2" TO GROUP_NAME.
    STORE WK_GROUP.

    MOVE 01400 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 02000 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 05000 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 05500 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

    MOVE 07400 TO EMP_ID.
    FIND FIRST EMPLOYEE WITHIN ALL_EMPLOYEES USING EMP_ID.
    CONNECT EMPLOYEE TO CONSISTS_OF.

IDENTIFICATION DIVISION.
PROGRAM-ID. PROMOTION-UPDATE.

PROCEDURE DIVISION.
A000-BEGIN.
    MOVE "A" TO GROUP_NAME.
*
* The next statement makes HOWELL’s GROUP "A" record current
*
    FIND FIRST WK_GROUP USING GROUP_NAME.
*
* The next two statements fetch KLEIN using EMP_ID.
* The RETAINING clause keeps the WK_GROUP record "A"
* as current of the CONSISTS_OF set. This allows the program
* to connect KLEIN to the correct occurrence of WK_GROUP.
* A fetch to KLEIN without the RETAINING clause makes KLEIN
* current of CONSISTS_OF thus destroying the pointer to the
* WK_GROUP record "A".
*
    MOVE 05000 TO EMP_ID.
    FETCH FIRST EMPLOYEE USING EMP_ID RETAINING CONSISTS_OF.
*
* The next statement disconnects KLEIN from the WK_GROUP "B1"
* record and connects him to the current WK_GROUP "A" record.
*
    RECONNECT EMPLOYEE WITHIN CONSISTS_OF.
*
* The next two sentences create and store a WK_GROUP record.
* Because KLEIN is current of EMPLOYEE, a STORE WK_GROUP
* automatically connects WK_GROUP as a member of the MANAGES
* set owned by KLEIN, and makes "B3" current of the MANAGES
* and CONSISTS_OF sets.
*
    MOVE "B3" TO WK_GROUP.
    STORE WK_GROUP.
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
*
* The next two statements fetch NEILS and retain WK_GROUP
* "B3" as current of CONSISTS_OF.
*
    MOVE 7000 TO EMP_ID.
    FETCH FIRST EMPLOYEE USING EMP_ID RETAINING CONSISTS_OF.
*
* The next statement disconnects NEILS from WK_GROUP "B1"
* record and reconnects him to the WK_GROUP "B3" record.
* It also retains "B3" as current of CONSISTS_OF. This
* maintains the pointer at "B3" allowing the program to
* reassign RILEY to KLEIN.
*
    RECONNECT EMPLOYEE WITHIN CONSISTS_OF RETAINING CONSISTS_OF.
*
* The next three statements fetch RILEY, disconnect him from
* "B2" and reconnect him to "B3".
*
    MOVE 01400 TO EMP_ID.
    FETCH FIRST EMPLOYEE USING EMP_ID RETAINING CONSISTS_OF.
    RECONNECT EMPLOYEE WITHIN CONSISTS_OF.

END PROGRAM PROMOTION-UPDATE.

IDENTIFICATION DIVISION.
PROGRAM-ID. PERSONNEL-REPORT.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PERSONNEL-REPORT-FILE ASSIGN TO "TT:".

DATA DIVISION.
FILE SECTION.
FD  PERSONNEL-REPORT-FILE
    VALUE OF ID IS "PERSONNEL.LIS"
    REPORT IS PERSONNEL-LISTING.

WORKING-STORAGE SECTION.
01  CONTROL-FIELDS.
    02  MANAGER-NAME      PIC X(20).
    02  MANAGES-GROUP     PIC XX.
    02  SUPERVISOR-NAME   PIC X(20).
    02  SUPERVISES-GROUP  PIC XX.
    02  EMPLOYEE-NUMBER   PIC XXXXX.
    02  EMPLOYEE-NAME     PIC X(20).
REPORT SECTION.
RD  PERSONNEL-LISTING
    PAGE LIMIT IS 66
    HEADING       1
    FIRST DETAIL  3
    LAST DETAIL   60
    CONTROLS ARE  MANAGES-GROUP
                  SUPERVISES-GROUP.
01  TYPE IS PAGE HEADING.
    02  LINE 1 COLUMN 22
               PIC X(16) VALUE "EMPLOYEE LISTING".
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
01  MANAGER-CONTROL TYPE IS CONTROL HEADING MANAGES-GROUP.
    02  LINE IS PLUS 1.
        03  COLUMN 16 PIC X(17)
                      VALUE "MANAGER OF GROUP ".
        03  COLUMN 33 PIC XX
                      SOURCE MANAGES-GROUP.
        03  COLUMN 35 PIC XXXX
                      VALUE "IS: ".
        03  COLUMN 39 PIC X(20)
                      SOURCE MANAGER-NAME.
01  GROUP-CONTROL TYPE IS CONTROL HEADING SUPERVISES-GROUP.
    02  LINE IS PLUS 1.
        03  COLUMN 3  PIC XXXXXXX
                      VALUE "GROUP: ".
        03  COLUMN 10 PIC XX
                      SOURCE SUPERVISES-GROUP.
    02  LINE IS PLUS 1.
        03  COLUMN 3  PIC X(15)
                      VALUE IS "SUPERVISOR IS: ".
        03  COLUMN 18 PIC X(20)
                      SOURCE IS SUPERVISOR-NAME.
    02  LINE IS PLUS 2.
        03  COLUMN 3  PIC X(6)
                      VALUE "GROUP ".
        03  COLUMN 9  PIC XX
                      SOURCE IS SUPERVISES-GROUP.
        03  COLUMN 12 PIC X(9)
                      VALUE "EMPLOYEES".
        03  COLUMN 24 PIC X(15)
                      VALUE "EMPLOYEE NUMBER".
        03  COLUMN 43 PIC X(13)
                      VALUE "EMPLOYEE NAME".
01  EMPLOYEE-LINE TYPE IS DETAIL.
    02  LINE IS PLUS 1.
        03  COLUMN 28 PIC XXXXX SOURCE IS EMPLOYEE-NUMBER.
        03  COLUMN 44 PIC X(20) SOURCE IS EMPLOYEE-NAME.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT PERSONNEL-REPORT-FILE.
    INITIATE PERSONNEL-LISTING.
    PERFORM A100-GET-THE-BOSS THROUGH A700-DONE-THE-BOSS.
    TERMINATE PERSONNEL-LISTING.
    CLOSE PERSONNEL-REPORT-FILE.
    EXIT PROGRAM.

A100-GET-THE-BOSS.
    MOVE 10500 TO EMP_ID.
    FETCH FIRST EMPLOYEE USING EMP_ID.
    MOVE EMP_LAST_NAME TO MANAGER-NAME.
    FETCH FIRST WK_GROUP WITHIN MANAGES.
    MOVE GROUP_NAME TO MANAGES-GROUP.

A200-GET-SUPERVISORS.
    FETCH NEXT EMPLOYEE WITHIN CONSISTS_OF
               AT END GO TO A700-DONE-THE-BOSS.
    MOVE EMP_LAST_NAME TO SUPERVISOR-NAME.
    KEEP CURRENT USING KEEPSUPER.
    FETCH NEXT WK_GROUP WITHIN MANAGES.
    MOVE GROUP_NAME TO SUPERVISES-GROUP.
    PERFORM A500-GET-EMPLOYEES THROUGH A600-DONE-EMPLOYEES.
    GO TO A200-GET-SUPERVISORS.
```

(continued on next page)

Example 15–20 (Cont.): Creating New Record Relationships

```
A500-GET-EMPLOYEES.
    FETCH NEXT EMPLOYEE WITHIN CONSISTS_OF
               AT END GO TO A510-FIND-CURRENT-SUPER.
    MOVE EMP_LAST_NAME TO EMPLOYEE-NAME.
    MOVE EMP_ID        TO EMPLOYEE-NUMBER.
    GENERATE EMPLOYEE-LINE.
    GO TO A500-GET-EMPLOYEES.

A510-FIND-CURRENT-SUPER.
    FIND FIRST WITHIN KEEPSUPER.
    FREE ALL FROM KEEPSUPER.

A600-DONE-EMPLOYEES.
    EXIT.

A700-DONE-THE-BOSS.
    EXIT.

END PROGRAM PERSONNEL-REPORT.

END PROGRAM PERSONNEL-UPDATE.
```

15.30.7.1 PERSONNEL-UPDATE Sample Run—Listing Before Promotion

This sample report, created by the preceding PERSONNEL-UPDATE program, corresponds to the data in Figure 15–26.

Example 15–21: Sample Run of PERSONNEL-UPDATE Before Promotion

```
                  EMPLOYEE LISTING

            MANAGER OF GROUP A IS: HOWELL
GROUP B2
SUPERVISOR IS: MOORE

GROUP B2 EMPLOYEES   EMPLOYEE NUMBER    EMPLOYEE NAME
                         05500           BAKER
                         02000           DEANE
                         07400           FIFER
                         05000           KLEIN
                         01400           RILEY
GROUP B1
SUPERVISOR IS: NOYCE

GROUP B1 EMPLOYEES   EMPLOYEE NUMBER    EMPLOYEE NAME
                         04000           BURLEW
                         07000           NEILS
                         01000           RAVAN
```

15.30.7.2 PERSONNEL-UPDATE Sample Run—Listing After Promotion

This sample report, created by PERSONNEL-UPDATE in Section 15.30.7, corresponds to the data in Figure 15–27.

Example 15–22: Sample Run of PERSONNEL-UPDATE After Promotion

```
                  EMPLOYEE LISTING

            MANAGER OF GROUP A IS: HOWELL
GROUP B3
SUPERVISOR IS: KLEIN

GROUP B3 EMPLOYEES   EMPLOYEE NUMBER    EMPLOYEE NAME
                         07000           NEILS
                         01400           RILEY
GROUP B2
SUPERVISOR IS: MOORE
GROUP B2 EMPLOYEES   EMPLOYEE NUMBER    EMPLOYEE NAME
                         05500           BAKER
                         02000           DEANE
                         07400           FIFER
GROUP B1
SUPERVISOR IS: NOYCE

GROUP B1 EMPLOYEES   EMPLOYEE NUMBER    EMPLOYEE NAME
                         04000           BURLEW
                         01000           RAVAN
```

Chapter 16

Producing Printed Reports with VAX COBOL

This chapter discusses how to produce a printed report using VAX COBOL. It addresses the following topics:

- How to design a report
- The components of a report
- Methods of reporting accumulation and control totals
- Methods of programming your VAX COBOL report
- Modes of printing a report

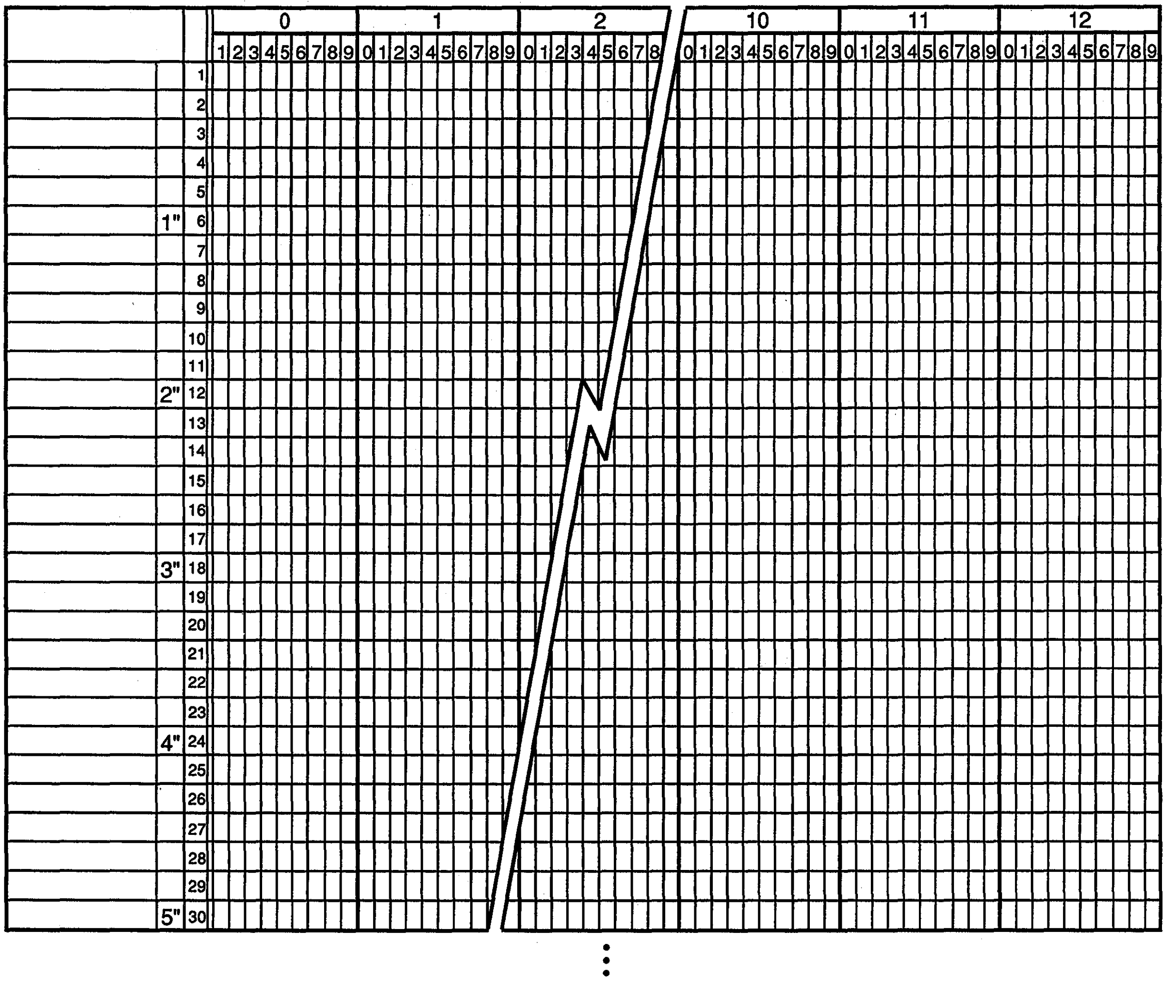
In addition, this chapter explains the use of the VAX COBOL Report Writer, a feature that assists you in formatting and producing reports.

16.1 Designing the Report

Whether you are producing a report for yourself or for a customer, you must begin by designing the report. The design of a report is dictated by the data you must include in the report. If you have a general idea of what the report is to contain, you can produce a rough outline using a report layout worksheet.

To create the worksheet, either use an online text editor or draw a layout worksheet like the one displayed in Figure 16–1.

Figure 16–1: Sample Layout Worksheet



ZK-6077-GE

The layout worksheet in Figure 16–1 has 132 characters on a line and 60 lines on a page. When you outline your worksheet, include specifics such as page headings, rows and columns, and column sizes.

Section 16.2 discusses other report components that you must plan for when you design a report. Note that you can use your worksheet later when you write the VAX COBOL program that produces the report.

16.2 Components of a Report

A description of the seven components of a report follows. The numbers in the following list correspond to the circled numbers in Figure 16–2:

❶ Report Heading (RH)—The report heading consists of information printed before the main body of a report. It can be printed on a separate page, or as the first page heading, with the remaining page headings abbreviated to save paper. The report heading can include information such as handling and distribution instructions. It can also include the selection criteria, sort order, and assumptions made when creating the report.

❷ Page Heading (PH)—The page heading consists of information printed on the top one or more lines of every page in the report. It usually names and dates the report, gives the report page number, and produces a title for each column of information in the detail line.

❸ Control Heading (CH)—The control heading consists of one or more lines of information identifying the beginning of a new logical area on a page.

❹ Detail Lines (DL)—The detail consists of one or more lines of the primary data of the report.

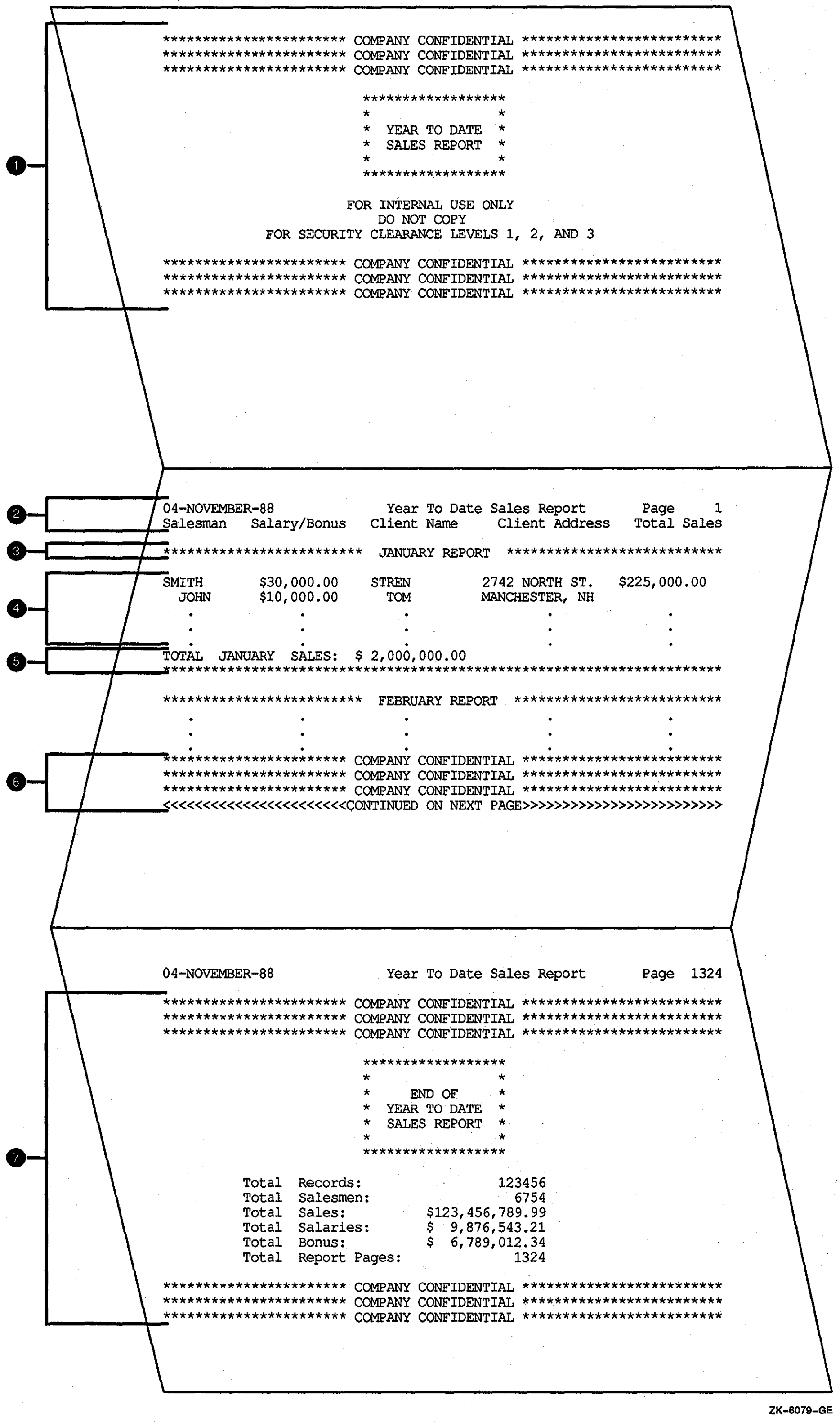
❺ Control Footing (CF)—The control footing consists of one or more lines of information identifying the end of a logical area. The control footing can contain one or more totals and an accompanying message.

❻ Page Footing (PF)—The page footing consists of one or more lines of information at the bottom of each page.

❼ Report Footing (RF)—The report footing consists of information printed after the main body of the report. It can be continued on the same page of the report body, or it can be on a separate page. It may contain information such as hash or control totals. A report footing is a convenient place to print run-time statistics, such as the number of records read and written for each file. It can also provide warning messages, such as when a table is close to overflowing.

It is suggested that all reports have an END OF REPORT message or other indicator at the end of the report, so that you can tell at a glance that you have all the pages. (The consecutive page numbers tell if a page is missing, but they do not indicate which page is the last.)

Figure 16–2: Components of a Report



```
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************

                         ******************
                         *                *
                         *  YEAR TO DATE  *
                         *  SALES REPORT  *
                         *                *
                         ******************

                       FOR INTERNAL USE ONLY
                           DO NOT COPY
             FOR SECURITY CLEARANCE LEVELS 1, 2, AND 3

*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
```

```
04-NOVEMBER-88              Year To Date Sales Report       Page     1
Salesman   Salary/Bonus  Client Name     Client Address   Total Sales
************************* JANUARY REPORT ***************************

SMITH       $30,000.00    STREN        2742 NORTH ST.   $225,000.00
  JOHN      $10,000.00     TOM         MANCHESTER, NH
   .             .           .               .               .
   .             .           .               .               .
TOTAL  JANUARY  SALES:  $ 2,000,000.00
**********************************************************************

************************* FEBRUARY REPORT **************************
   .             .           .               .               .
   .             .           .               .               .
   .             .           .               .               .
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
<<<<<<<<<<<<<<<<<<<<<<<CONTINUED ON NEXT PAGE>>>>>>>>>>>>>>>>>>>>>>>>>
```

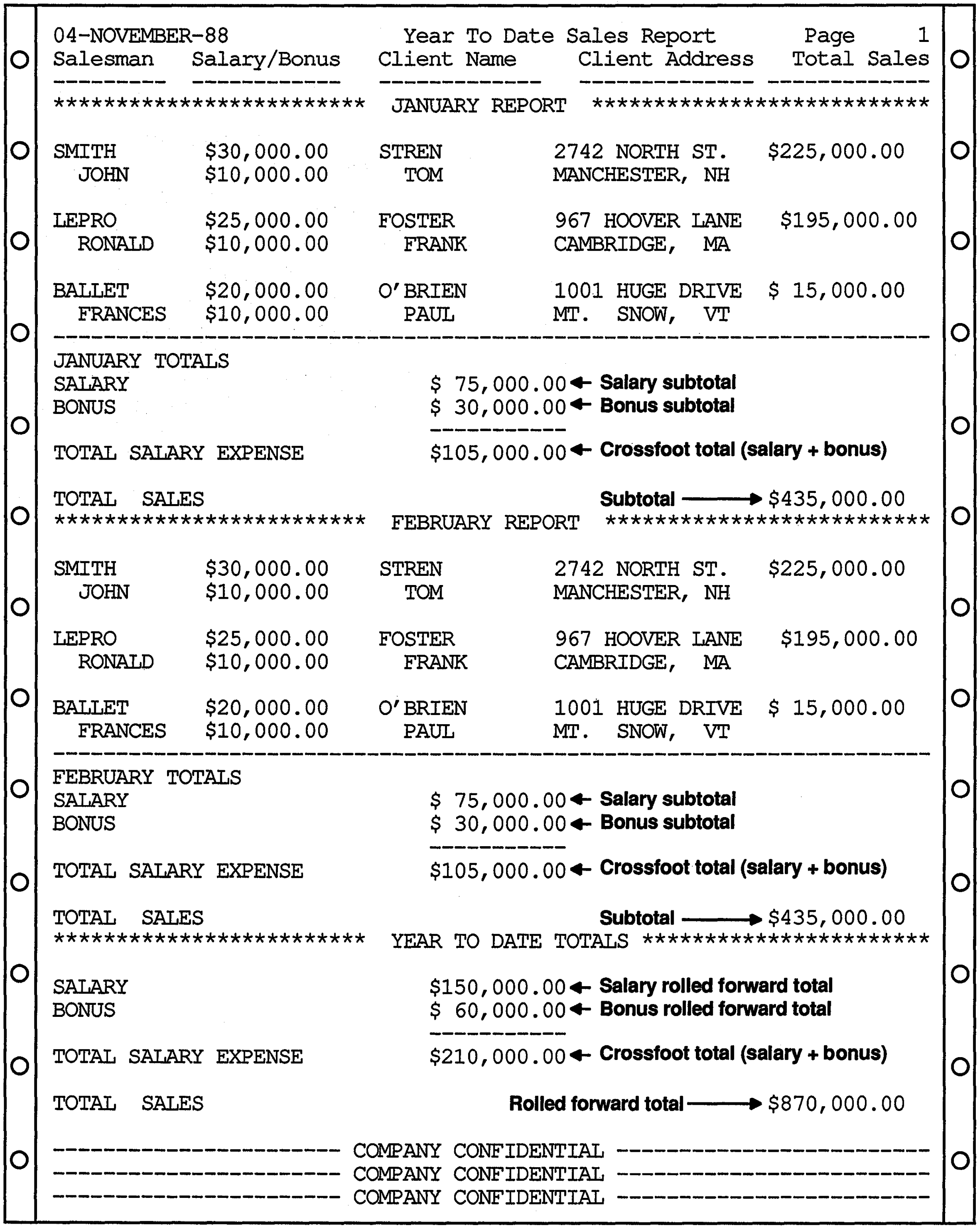
```
04-NOVEMBER-88              Year To Date Sales Report       Page  1324
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************

                         ******************
                         *                *
                         *     END OF     *
                         *  YEAR TO DATE  *
                         *  SALES REPORT  *
                         *                *
                         ******************

          Total  Records:                123456
          Total  Salesmen:                 6754
          Total  Sales:           $123,456,789.99
          Total  Salaries:        $  9,876,543.21
          Total  Bonus:           $  6,789,012.34
          Total  Report Pages:             1324

*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
*********************** COMPANY CONFIDENTIAL *************************
```

16.3 Accumulating and Reporting Totals

Your program can report three types of totals in the control footings and report footings of your report:

- Subtotals—Subtotaling is the process of summing a detail item from each detail line. For example, in Figure 16–3, Salary, Bonus, and Total Sales are subtotaled. To get the first salary subtotal for January on page 1 ($75,000.00), the program must add each salesman's salary ($30,000+$25,000+$20,000). After printing the salary total, the program must zero the total to begin subtotaling for the next month.
- Crossfoot Totals—Crossfooting is the process of summing subtotals from a common group of totals. For example, in Figure 16–3, TOTAL SALARY EXPENSE is crossfooted by adding TOTAL SALARY and TOTAL BONUS. To get the first TOTAL SALARY EXPENSE crossfoot total for the January report, the program must add the salary subtotal and the bonus subtotal before the program clears the subtotals.
- Rolled Forward Totals—Rolling-forward is the process of summing either subtotals or crossfoot totals. For example, in Figure 16–3, the YEAR TO DATE TOTALS at the bottom of page 1 are rolled forward from both the JANUARY and FEBRUARY totals. The program computes the salary and bonus YEAR TO DATE TOTALS from the previous salary and bonus subtotals. It computes the total salary expense figure from the previous total salary expense crossfoot totals.

Figure 16–3: Subtotals, Crossfoot Totals, and Rolled Forward Totals



```
04-NOVEMBER-88              Year To Date Sales Report       Page     1
Salesman   Salary/Bonus   Client Name     Client Address   Total Sales
---------  ------------   -------------   --------------  ------------
*************************  JANUARY REPORT  ***************************

SMITH        $30,000.00    STREN         2742 NORTH ST.    $225,000.00
  JOHN       $10,000.00      TOM         MANCHESTER, NH

LEPRO        $25,000.00    FOSTER        967 HOOVER LANE    $195,000.00
  RONALD     $10,000.00      FRANK       CAMBRIDGE,  MA

BALLET       $20,000.00    O'BRIEN       1001 HUGE DRIVE   $ 15,000.00
  FRANCES    $10,000.00      PAUL        MT.  SNOW,  VT
----------------------------------------------------------------------
JANUARY TOTALS
SALARY                         $ 75,000.00 ← Salary subtotal
BONUS                          $ 30,000.00 ← Bonus subtotal
                               -----------
TOTAL SALARY EXPENSE           $105,000.00 ← Crossfoot total (salary + bonus)

TOTAL  SALES                               Subtotal ──→ $435,000.00
*************************  FEBRUARY REPORT  **************************

SMITH        $30,000.00    STREN         2742 NORTH ST.    $225,000.00
  JOHN       $10,000.00      TOM         MANCHESTER, NH

LEPRO        $25,000.00    FOSTER        967 HOOVER LANE    $195,000.00
  RONALD     $10,000.00      FRANK       CAMBRIDGE,  MA

BALLET       $20,000.00    O'BRIEN       1001 HUGE DRIVE   $ 15,000.00
  FRANCES    $10,000.00      PAUL        MT.  SNOW,  VT
----------------------------------------------------------------------
FEBRUARY TOTALS
SALARY                         $ 75,000.00 ← Salary subtotal
BONUS                          $ 30,000.00 ← Bonus subtotal
                               -----------
TOTAL SALARY EXPENSE           $105,000.00 ← Crossfoot total (salary + bonus)

TOTAL  SALES                               Subtotal ──→ $435,000.00
*************************  YEAR TO DATE TOTALS  **********************

SALARY                         $150,000.00 ← Salary rolled forward total
BONUS                          $ 60,000.00 ← Bonus rolled forward total
                               -----------
TOTAL SALARY EXPENSE           $210,000.00 ← Crossfoot total (salary + bonus)

TOTAL  SALES                    Rolled forward total ──→ $870,000.00

----------------------- COMPANY CONFIDENTIAL ------------------------
----------------------- COMPANY CONFIDENTIAL ------------------------
----------------------- COMPANY CONFIDENTIAL ------------------------
```

ZK–6080–GE

16.4 The Logical Page and the Physical Page

A physical page is the paper page printed by your printer.

A logical page is conceptual, consisting of a page body and optionally a top margin, footing, and bottom margin. Figure 16–4 and Figure 16–7 illustrate the logical page structure for the conventional file report and linage-file report, respectively.

The number of lines on a logical page is defined by the number of lines on the target physical page. Thus, the number of lines determines the size of the logical page. When you design a report, you must choose those lines within the logical page that are to be page headers (PH), control headers (CH), detail lines (DL), control footings (CF), and page footings (PF). Once the framework of the logical page is defined, your program must stay within those bounds; otherwise, the printed report may not contain the correct information.

You can program two types of reports: a conventional file report or a linage-file report. Section 16.5 and Section 16.5.1 discuss these reports in detail.

16.5 Programming the Conventional VAX COBOL Report

A conventional file report has sequential organization and access mode, contains variable-length with fixed control records, and consists of one or more logical pages.

To program a conventional report, you should understand how to do the following:

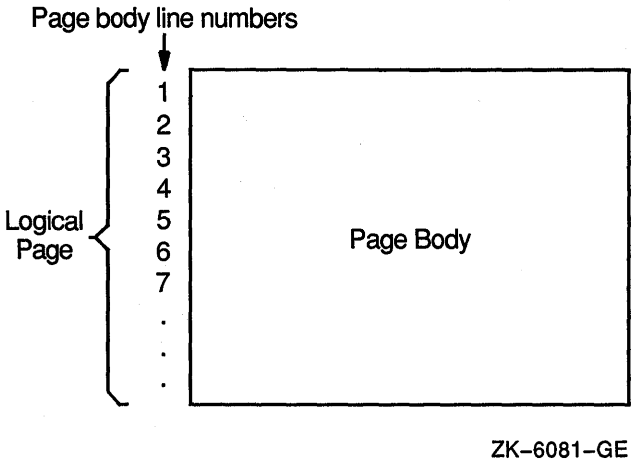
- Define the logical page
- Advance to the next logical page
- Program for the page-overflow condition
- Use a line counter

The following sections discuss these topics in detail. Additionally, Section 16.5.5 contains an example of a VAX COBOL program that produces a conventional file report.

16.5.1 Defining the Logical Page in a Conventional Report

Your program specifies the format of your report. Using the report layout worksheet you created, you can write a VAX COBOL program that defines the logical page area for a conventional report. Figure 16–4 shows the logical page area for a conventional report. The conventional report logical page area consists of the page areas discussed in Section 16.4.

Figure 16–4: Logical Page Area for a Conventional Report



Once you have defined the logical page, you must handle vertical spacing, horizontal spacing, and the number of lines that appear on each page so that you can advance to the next logical page. The following sections discuss these subjects.

16.5.2 Controlling the Spacing in a Conventional Report

To control the horizontal spacing on a logical page, define every report item from your report layout worksheet in the Working-Storage Section of your VAX COBOL program.

To control the vertical spacing on a logical page, use the WRITE statement. The WRITE statement controls whether one or more lines are skipped before or after your program writes a line of the report. For example, to print a line before advancing five lines, use the following:

```
WRITE... BEFORE ADVANCING 5 LINES.
```

To print a line after advancing two lines, use the following:

```
WRITE... AFTER ADVANCING 2 LINES.
```

16.5.3 Advancing to the Next Logical Page in a Conventional Report

To advance to the next logical page and position the printer to the page heading area, you must be able to track the number of lines that your program writes on a page. The VAX COBOL compiler lets you control the number of lines written on a page with the WRITE statement.

The WRITE statement must appear in your Procedure Division and it should contain either the AFTER ADVANCING PAGE or BEFORE ADVANCING PAGE clause. Example 16–2 demonstrates the use of the WRITE statement with the AFTER ADVANCING PAGE clause.

The next two sections discuss how to handle a page-overflow condition and how to use a line counter to keep track of the number of lines your program writes on a logical page.

16.5.3.1 Programming for the Page-Overflow Condition in a Conventional Report

A page-overflow condition occurs when your program writes more lines than the logical page can accommodate. This normal condition lets your program know when to execute its top-of-page routines. Top-of-page routines should contain WRITE statements with either the AFTER ADVANCING PAGE or BEFORE ADVANCING PAGE clause.

These statements determine when a report's logical page is full, and when the program prints the last line on a logical page (if you do not want to use all the lines on a page). Example 16–1 shows two methods that check for the page-overflow condition:

- Paragraph A100-FIRST-REPORT-ROUTINES checks for a full page after it writes a report line. If the page-overflow condition exists, A901-HEADER-ROUTINE executes.
- Paragraph A500-SECOND-REPORT-ROUTINES checks if more than 50 lines exist on the current logical page. If more than 50 lines exist, A902-HEADER-ROUTINE executes.

In either case, the AFTER ADVANCING PAGE clause in the A901-HEADER-ROUTINE and A902-HEADER-ROUTINE paragraphs generates the characters needed for the printer to position itself at the top of the next page heading area.

Example 16–1: Checking for the Page-Overflow Condition

```
PROCEDURE DIVISION.
A000-BEGIN.
     .
     .
     .
A100-FIRST-REPORT-ROUTINES.
*
* A901-HEADER-ROUTINE executes whenever the number of lines written exceeds
* the number of lines on the 66-line default logical page.
*
    WRITE A-LINE1 AFTER ADVANCING 2 LINES.
    ADD 2 TO REPORT1-LINE-COUNT.
    IF REPORT1-LINE-COUNT > 65 PERFORM A901-HEADER-ROUTINE.
     .
     .
     .
A500-SECOND-REPORT-ROUTINES.
*
* This routine uses only the first 50 lines of the 66-line report.
*
    WRITE A-LINE2 AFTER ADVANCING 2 LINES.
    ADD 2 TO REPORT2-LINE-COUNT.
    IF REPORT2-LINE-COUNT IS GREATER THAN 50
                               PERFORM A902-HEADER-ROUTINE.
     .
     .
     .
```

(continued on next page)

Example 16–1 (Cont.): Checking for the Page-Overflow Condition

```
A901-HEADER-ROUTINE.
    WRITE A-LINE1 FROM REPORT1-HEADER-LINE-1 AFTER ADVANCING PAGE.
    MOVE 0 TO REPORT1-LINE-COUNT.
    ADD 1 TO REPORT1-LINE-COUNT.
      .
      .
      .
A902-HEADER-ROUTINE.
    WRITE A-LINE2 FROM REPORT2-HEADER-LINE-1 AFTER ADVANCING PAGE.
    MOVE 0 TO REPORT2-LINE-COUNT.
    ADD 1 TO REPORT2-LINE-COUNT.
      .
      .
      .
```

Although the WRITE statement allows you to check for a page-overflow condition, you can also use a line counter that tracks the number of lines that appear on a page. Section 16.5.3.2 discusses this in more detail.

16.5.3.2 Using a Line Counter

A line counter is another method of tracking the number of lines that appear on a page. If you define a line counter in the Working-Storage Section of your program, each time a line is written or skipped the line counter value is incremented by one.

Your program should contain a routine that checks the line counter value before it writes or skips the next line. If the value is less than the limit you have set, it writes or skips. If the value equals or exceeds the limit you have set, the program executes header routines that allow it to advance to the next logical page.

16.5.4 Printing the Conventional Report

When you are ready to print your report, you must ensure that your system's line printer can accommodate the page size or form of your report. If the printer uses a different page size or form, contact your system manager. The system manager can change the page or form size to accommodate your report.

Section 16.7 discusses the different modes for printing a report.

16.5.5 A Conventional File Report Example

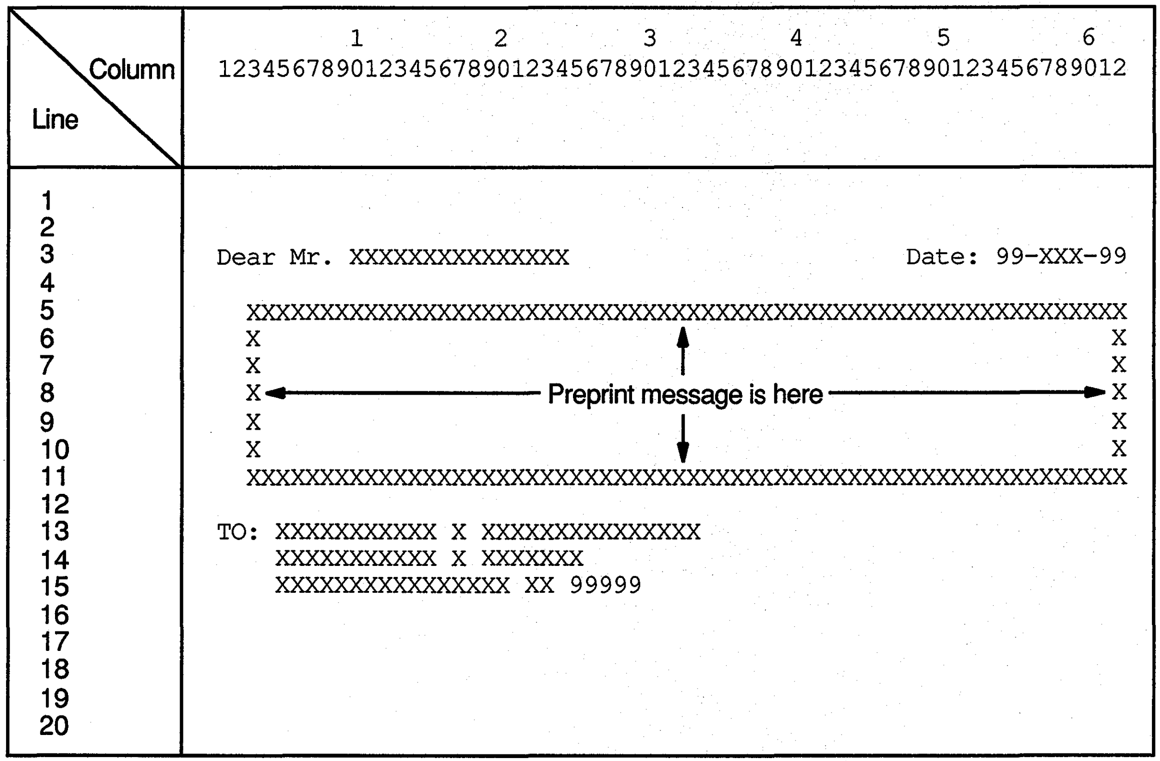
Example 16–2 shows a VAX COBOL program that produces two reports from the same input file.

The first report, Figure 16–5, is a preprinted form letter that can be inserted into a business envelope. This report has a logical page length of 20 lines and a width of 80 characters. Note that this report uses only the first 15 lines on the page. Because this is a preprinted form, the program supplies only the following information:

- The date for line 3
- The customer's name for lines 3 and 13

- The customer's address for lines 14 and 15

Figure 16–5: A 20-Line Logical Page

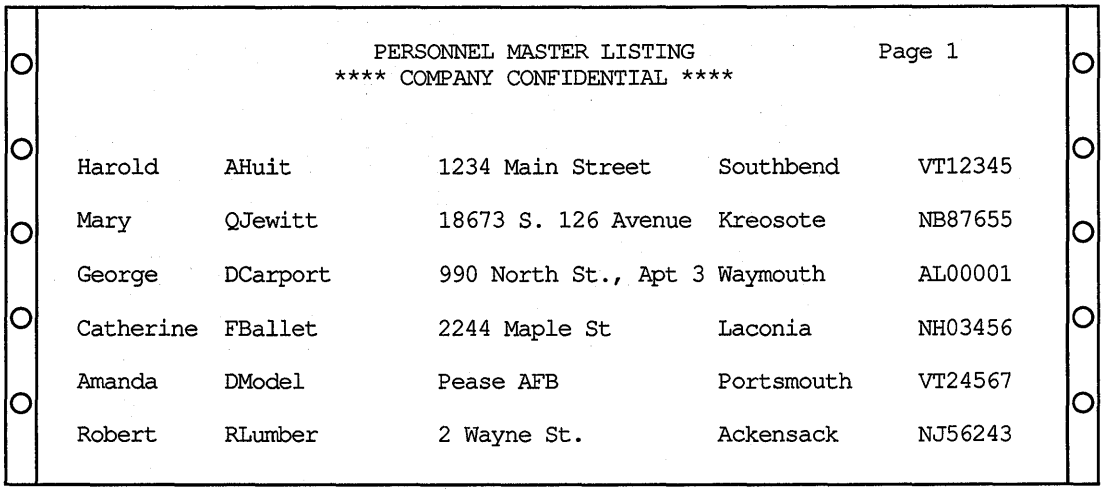


```
     Column          1         2         3         4         5         6
Line        12345678901234567890123456789012345678901234567890123456789012

 1
 2
 3          Dear Mr. XXXXXXXXXXXXXXX                       Date: 99-XXX-99
 4
 5            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 6            X                                                          X
 7            X                                                          X
 8            X<------------ Preprint message is here ------------------>X
 9            X                                                          X
10            X                                                          X
11            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
12
13          TO: XXXXXXXXXXX X XXXXXXXXXXXXXXX
14              XXXXXXXXXXX X XXXXXXX
15              XXXXXXXXXXXXXXXX XX 99999
16
17
18
19
20
```

ZK–6082–GE

The second report, Figure 16–6, is a double-spaced master listing of all input records. While this report's logical page is identical to the default logical page for the system (in this case, 66 vertical lines and 132 horizontal characters), this report uses only the first 55 lines on the page. Both reports are output to a disk for later printing.

Figure 16–6: A Double-Spaced Master Listing



```
                PERSONNEL MASTER LISTING          Page 1
              **** COMPANY CONFIDENTIAL ****


Harold     AHuit          1234 Main Street    Southbend      VT12345

Mary       QJewitt        18673 S. 126 Avenue Kreosote       NB87655

George     DCarport       990 North St., Apt 3 Waymouth      AL00001

Catherine  FBallet        2244 Maple St       Laconia        NH03456

Amanda     DModel         Pease AFB           Portsmouth     VT24567

Robert     RLumber        2 Wayne St.         Ackensack      NJ56243
```

ZK-6083-GE

Example 16–2: Page Advancing and Line Skipping

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REP01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE   ASSIGN TO "REPIN.DAT".
    SELECT FORM1-REPORT ASSIGN TO "FORM1.DAT".
    SELECT FORM2-REPORT ASSIGN TO "FORM2.DAT".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST                     PIC X(10).
        03  I-MID                       PIC X.
        03  I-LAST                      PIC X(15).
    02  I-ADDRESS.
        03  I-STREET                    PIC X(20).
        03  I-CITY                      PIC X(15).
        03  I-STATE                     PIC XX.
        03  I-ZIP                       PIC 99999.
FD  FORM1-REPORT.
01  FORM1-PRINT-LINE                    PIC X(80).
FD  FORM2-REPORT.
01  FORM2-PRINT-LINE                    PIC X(80).
WORKING-STORAGE SECTION.
01  END-OF-FILE                         PIC  X     VALUE SPACE.
01  MAX-LINES-ON-FORM2                  PIC  99    VALUE 55.
01  FORM2-LINE-COUNTER                  PIC  99    VALUE 00.
01  PAGE-NO                             PIC  99999 VALUE 0.
```

(continued on next page)

Example 16–2 (Cont.): Page Advancing and Line Skipping

```
01  FORM1-LINE-3.
    02                                  PIC X(9)    VALUE SPACES.
    02  FORM1-LAST                      PIC X(15).
01  FORM1-LINE-13.
    02                                  PIC X(4)    VALUE SPACES.
    02  FORM1-NAME                      PIC X(26).
01  FORM1-LINE-14.
    02                                  PIC X(4)    VALUE SPACES.
    02  FORM1-STREET                    PIC X(20).
01  FORM1-LINE-15.
    02                                  PIC X(4)    VALUE SPACES.
    02  FORM1-CITY                      PIC X(15).
    02                                  PIC X       VALUE SPACE.
    02  FORM1-STATE                     PIC XX.
    02                                  PIC X       VALUE SPACE.
    02  FORM1-ZIP                       PIC 99999.
01  FORM2-HEADER-1.
    02              PIC X(15) VALUE SPACES.
    02              PIC X(30) VALUE "   PERSONNEL MASTER LISTING   ".
    02              PIC X(10) VALUE SPACES.
    02              PIC XXXXX VALUE "Page ".
    02  F2H-PAGE    PIC ZZZZZ.
01  FORM2-HEADER-2.
    02              PIC X(15) VALUE SPACES.
    02              PIC X(30) VALUE "**** COMPANY CONFIDENTIAL ****".

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT FORM1-REPORT
                FORM2-REPORT.
    PERFORM A900-PRINT-HEADERS-ROUTINE.
    PERFORM A100-PRINT-REPORTS UNTIL END-OF-FILE = "Y".
    CLOSE INPUT-FILE
          FORM1-REPORT
          FORM2-REPORT.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-PRINT-REPORTS.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       PERFORM A200-PRINT-REPORTS.
A200-PRINT-REPORTS.
    IF FORM2-LINE-COUNTER IS GREATER THAN MAX-LINES-ON-FORM2
       PERFORM A900-PRINT-HEADERS-ROUTINE.
    WRITE FORM2-PRINT-LINE FROM INPUT-RECORD
                           AFTER ADVANCING 2 LINES.
    ADD 2 TO FORM2-LINE-COUNTER.
    MOVE I-LAST      TO FORM1-LAST.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-3
                           AFTER ADVANCING 3 LINES.
    MOVE I-NAME      TO FORM1-NAME.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-13
                           AFTER ADVANCING 10 LINES.
    MOVE I-STREET    TO FORM1-STREET.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-14.
    MOVE I-CITY      TO FORM1-CITY.
    MOVE I-STATE     TO FORM1-STATE.
    MOVE I-ZIP       TO FORM1-ZIP.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-15.
```

(continued on next page)

Example 16–2 (Cont.): Page Advancing and Line Skipping

```
A900-PRINT-HEADERS-ROUTINE.
*
* This routine generates a form feed, writes two lines,
* skips two lines, then resets the line counter to 4 to
* indicate used lines on the current logical page.
* Line 5 on this page is the next print line.
*
    ADD 1 TO PAGE-NO.
    MOVE PAGE-NO TO F2H-PAGE.
    WRITE FORM2-PRINT-LINE FROM FORM2-HEADER-1
                        AFTER ADVANCING PAGE.
    WRITE FORM2-PRINT-LINE FROM FORM2-HEADER-2
                        BEFORE ADVANCING 2.
    MOVE 4 TO FORM2-LINE-COUNTER.
```

16.6 Programming the Linage-File VAX COBOL Report

A linage-file report has sequential organization and access mode, contains variable-length with fixed control records, and consists of one or more logical pages. Unlike the conventional VAX COBOL report, however, you can use the LINAGE clause to do the following:

- Define the number of lines on the logical page
- Divide the logical page into sections

Additionally, a linage-file report has a LINAGE-COUNTER special register assigned to it that monitors the number of lines written to the current logical page.

To program a linage report, you should understand how to do the following:

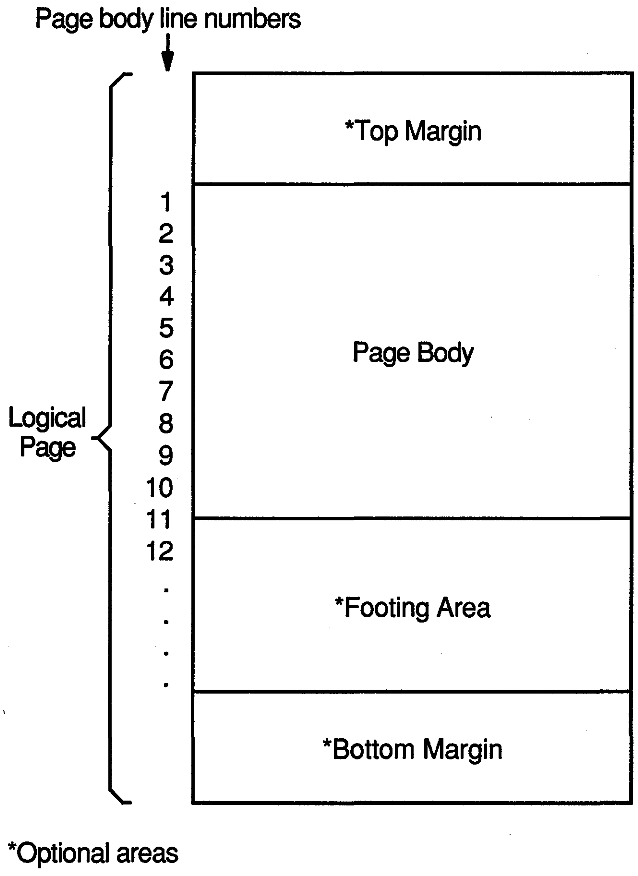
- Define the logical page with the LINAGE clause
- Use the LINAGE-COUNTER special register
- Advance to the next logical page
- Program for the page-overflow condition

The following sections discuss these topics in detail. Example 16–4 shows an example of a linage-file program.

16.6.1 Defining the Logical Page in a Linage-File Report

Your program specifies the format of your report. Using the report layout worksheet you created, you can write a VAX COBOL program that defines the logical page area and divides the page into logical page sections for a linage-file report. Figure 16–7 shows the logical page area and the four divisions of a linage-file report.

Figure 16–7: Logical Page Areas for a Linage File Report



ZK-6084-GE

To define the number of lines on a logical page and to divide it into logical page sections, you must include the LINAGE clause as a File Description entry in your program. The LINAGE clause lets you specify the size of the logical page's top and bottom margins and the line where the footing area begins in the page body.

For example, to define how many lines you want your program to skip at the top or bottom of the logical page, use the LINAGE clause with either the LINES AT TOP or the LINES AT BOTTOM phrase. To define a footing area within the logical page, use the LINAGE clause with the WITH FOOTING phrase.

The LINES AT TOP phrase positions the printer on the first print line in the page body. The LINES AT BOTTOM phrase positions the printer at the top of the next logical page once the current page body is complete. The WITH FOOTING phrase defines a footing area in the logical page that controls page-overflow conditions. Additionally, you can insert specific text, such as footnotes or page numbers, on the bottom lines of your logical page.

In addition to defining the logical page area and the number of lines that appear on a page, you must be prepared to handle vertical spacing, horizontal spacing, logical page advancement, and page-overflow. The following sections discuss these topics in detail.

16.6.2 Controlling the Spacing in a Linage-File Report

To control the horizontal spacing on a logical page, define every report item from your report layout worksheet in the Working-Storage Section of your VAX COBOL program.

To control the vertical spacing on a logical page, use the WRITE statement. The WRITE statement controls whether one or more lines is skipped before or after your program writes a line of the report. For example, to print a line before advancing five lines, use the following:

```
WRITE... BEFORE ADVANCING 5 LINES.
```

To print a line after advancing two lines, use the following:

```
WRITE... AFTER ADVANCING 2 LINES.
```

16.6.3 Using the LINAGE-COUNTER

The LINAGE-COUNTER special register is one method of tracking the number of lines that your program writes on a logical page. When you use the LINAGE-COUNTER special register, each time a line is written or skipped, the register is incremented by 1.

Before the program writes a new line, it checks the LINAGE-COUNTER value to see if the current logical page can accept the new line. If the value equals the maximum number of lines for the page body, the compiler positions the pointer on the first print line of the next page body. The compiler automatically resets this register to 1 each time your program begins a new logical page.

If you choose not to use the LINAGE-COUNTER register, you can advance to the next logical page using the WRITE statement, as explained in Section 16.6.4.

16.6.4 Advancing to the Next Logical Page in a Linage-File Report

Linage-files automatically advance to the next logical page when the LINAGE-COUNTER value equals the number of lines on the logical page. However, VAX COBOL also lets your program control logical page advancement with the WRITE statement.

To manually advance to the next logical page from any line in the current page body and position the printer on the first print line of the next page body, your program must include the WRITE statement with either the BEFORE ADVANCING PAGE clause or the AFTER ADVANCING PAGE clause. For an example of the WRITE statement, see Section 16.6.7.

Section 16.6.5 discusses how to handle a page-overflow condition.

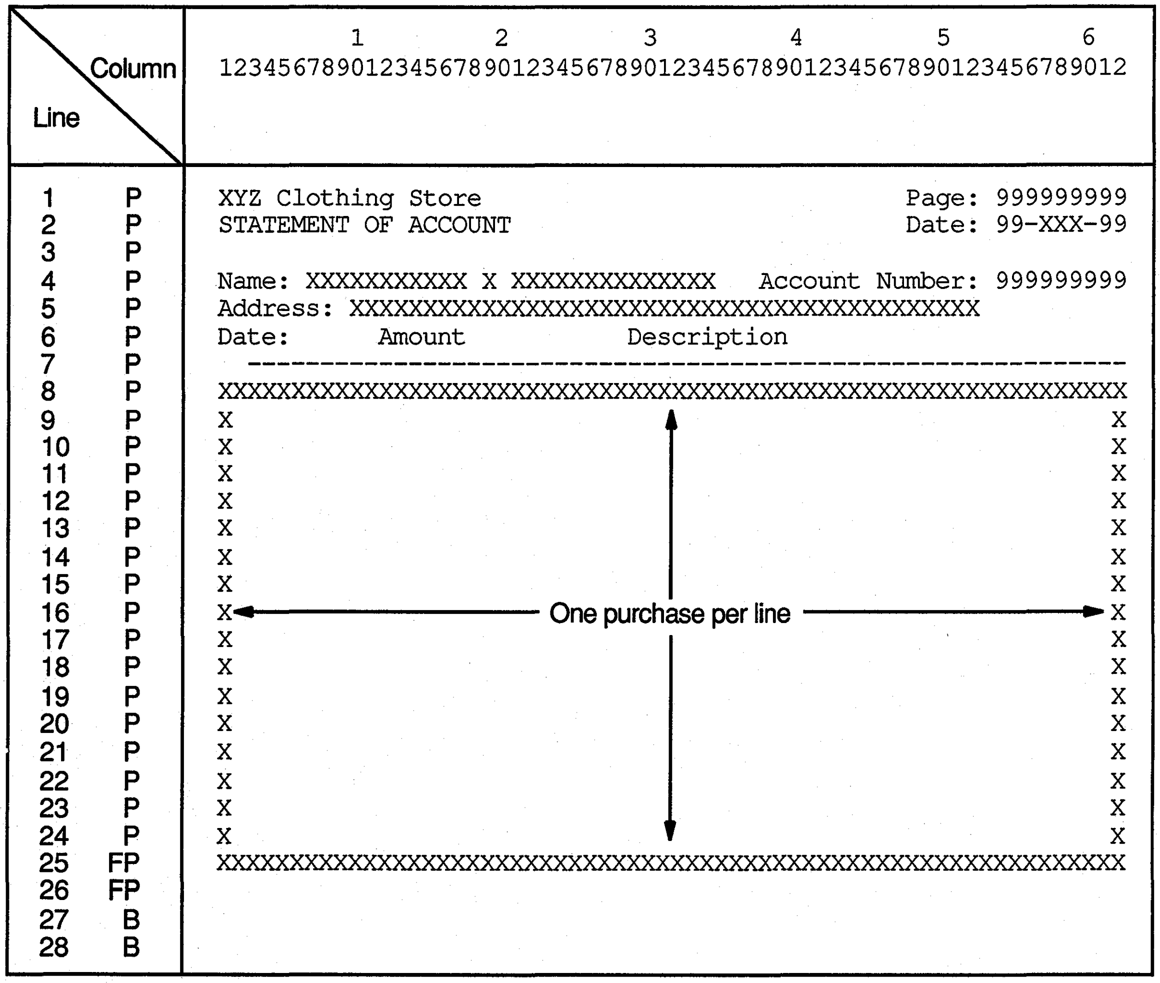
16.6.5 Programming for the Page-Overflow Condition

A page-overflow condition occurs when your program writes more lines than the logical page can accommodate. Although the compiler automatically advances to the next logical page when you use the LINAGE-COUNTER register, header information is not printed, and the overflow lines begin the next logical page.

If you want your program to advance to the next page and print page headers on the new page when the page is full, you should include routines in your program that limit the number of lines on each logical page.

Example 16–3 demonstrates how to include these routines in your program using the logical page shown in Figure 16–8.

Figure 16–8: A 28-Line Logical Page



```
        Column           1         2         3         4         5         6
                12345678901234567890123456789012345678901234567890123456789012
Line

 1   P          XYZ Clothing Store                               Page: 999999999
 2   P          STATEMENT OF ACCOUNT                             Date: 99-XXX-99
 3   P
 4   P          Name: XXXXXXXXXXX X XXXXXXXXXXXXXX   Account Number: 999999999
 5   P          Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 6   P          Date:      Amount           Description
 7   P          ------------------------------------------------------------
 8   P          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 9   P          X                                                            X
10   P          X                                                            X
11   P          X                                                            X
12   P          X                                                            X
13   P          X                                                            X
14   P          X                                                            X
15   P          X                                                            X
16   P          X<------------------ One purchase per line ----------------->X
17   P          X                                                            X
18   P          X                                                            X
19   P          X                                                            X
20   P          X                                                            X
21   P          X                                                            X
22   P          X                                                            X
23   P          X                                                            X
24   P          X                                                            X
25   FP         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
26   FP
27    B
28    B
```

Legend: T = Top margin = line 00
P = Page body = lines 01–26
F = Footing area = lines 25–26
B = Bottom margin = lines 27–28

ZK–6085–GE

In Figure 16–8, each detail line of the report represents a separate purchase at the XYZ Clothing Store. Each page can contain from 1 to 18 purchase lines. Each customer can have an unlimited number of purchases. A total of purchases for each customer is to appear on line 25 of that customer's last statement page. Headers appear on the top of each page.

The input file, INPUT.DAT, consists of individual purchase records sorted in ascending order by customer account number and purchase date. In Example 16–3, the LINAGE clause defines a footing area so the program can check for a page-overflow condition. When the condition is detected, the program executes its header routine to print lines 1 to 7.

Example 16–3: Checking for Page-Overflow on a 28-Line Logical Page

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPOVF.
* Print this report: PRINT REPORT.DAT/NOFEED
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "INPUT.DAT".
    SELECT REPORT-FILE ASSIGN TO "REPORT.DAT".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST          PIC X(10).
        03  I-MID            PIC X.
        03  I-LAST           PIC X(15).
    02  I-ADDRESS.
        03  I-STREET         PIC X(20).
        03  I-CITY           PIC X(15).
        03  I-STATE          PIC XX.
        03  I-ZIP            PIC 99999.
    02  I-ACCOUNT-NUMBER     PIC X(9).
    02  I-PURCHASE-DATE      PIC XXXXXX.
    02  I-PURCHASE-AMOUNT    PIC S9(6)V99.
    02  I-PURCHASE-DESCRIP   PIC X(20).
FD  REPORT-FILE
    LINAGE IS 26 LINES
           WITH FOOTING AT 25
           LINES AT BOTTOM  2.
01  PRINT-LINE               PIC X(80).
WORKING-STORAGE SECTION.
01  HEAD-1.
    02  H1-LC    PIC 99.
    02  FILLER   PIC X(20) VALUE "XYZ Clothing Store  ".
    02  FILLER   PIC X(25) VALUE SPACES.
    02  FILLER   PIC X(6)  VALUE "Page: ".
    02  H1-PAGE  PIC Z(9).
01  HEAD-2.
    02  H2-LC    PIC 99.
    02  FILLER   PIC X(20) VALUE "STATEMENT OF ACCOUNT".
    02  FILLER   PIC X(25) VALUE SPACES.
    02  FILLER   PIC X(6)  VALUE "Date: ".
    02  H2-DATE  PIC X(9).

01  HEAD-3.
    02  H3-LC    PIC 99.
    02  FILLER   PIC X(6)  VALUE "Name: ".
    02  H3-FNAME PIC X(10).
    02  FILLER   PIC X     VALUE SPACE.
    02  H3-MNAME PIC X.
    02  FILLER   PIC X     VALUE SPACE.
    02  H3-LNAME PIC X(15).
    02  FILLER   PIC X(17) VALUE " Account Number: ".
    02  H3-NUM   PIC Z(9).
```

(continued on next page)

Example 16–3 (Cont.): Checking for Page-Overflow on a 28-Line Logical Page

```
01  HEAD-4.
    02  H4-LC    PIC 99.
    02  FILLER   PIC X(9)  VALUE "Address: ".
    02  H4-STRT  PIC X(20).
    02  FILLER   PIC X     VALUE SPACE.
    02  H4-CITY  PIC X(15).
    02  FILLER   PIC X     VALUE SPACE.
    02  H4-STATE PIC XX.
    02  FILLER   PIC X     VALUE SPACE.
    02  H4-ZIP   PIC 99999.
01  HEAD-5.
    02  H5-LC    PIC 99.
    02  FILLER   PIC X(4)  VALUE "Date".
    02  FILLER   PIC X(7)  VALUE SPACES.
    02  FILLER   PIC X(6)  VALUE "Amount".
    02  FILLER   PIC X(10) VALUE SPACES.
    02  FILLER   PIC X(11) VALUE "Description".
01  HEAD-6       PIC X(61) VALUE ALL "-".
01  DETAIL-LINE.
    02  DET-LC   PIC 99.
    02  DL-DATE  PIC X(9).
    02  FILLER   PIC X     VALUE SPACE.
    02  DL-AMT   PIC $ZZZ,ZZZ.99-.
    02  FILLER   PIC X     VALUE SPACE.
    02  DL-DESC  PIC X(20).
01  TOTAL-LINE.
    02  TOT-LC   PIC 99.
    02  FILLER   PIC X(25) VALUE "Total purchases to date: ".
    02  TL       PIC $ZZZ,ZZZ,ZZZ.99-.
01  TOTAL-PURCHASES        PIC S9(9)V99.
01  PAGE-NUMBER            PIC S9(9).
01  HOLD-I-ACCOUNT-NUMBER  PIC X(9)   VALUE IS LOW-VALUES.
01  END-OF-FILE            PIC X      VALUE IS "N".
01  THESE-MANY             PIC 99     VALUE IS 1.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    DISPLAY " Enter date--DD-MMM-YY:".
    ACCEPT H2-DATE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A050-WRAP-UP.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END-OF-JOB".
    STOP RUN.
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE
                           PERFORM A400-PRINT-TOTALS
                           MOVE HIGH-VALUES TO I-ACCOUNT-NUMBER.
    DISPLAY INPUT-RECORD.
    IF END-OF-FILE NOT = "Y"
       AND I-ACCOUNT-NUMBER NOT = HOLD-I-ACCOUNT-NUMBER
              PERFORM A200-NEW-CUSTOMER.
    IF END-OF-FILE NOT = "Y"
       AND I-ACCOUNT-NUMBER = HOLD-I-ACCOUNT-NUMBER
              PERFORM A300-PRINT-DETAIL-LINE.
    MOVE I-ACCOUNT-NUMBER TO HOLD-I-ACCOUNT-NUMBER.
```

(continued on next page)

Example 16–3 (Cont.): Checking for Page-Overflow on a 28-Line Logical Page

```
A200-NEW-CUSTOMER.
    IF HOLD-I-ACCOUNT-NUMBER = LOW-VALUES
           PERFORM A600-SET-UP-HEADERS
           PERFORM A500-PRINT-HEADERS
           PERFORM A300-PRINT-DETAIL-LINE
       ELSE
           PERFORM A400-PRINT-TOTALS
           PERFORM A600-SET-UP-HEADERS
           PERFORM A500-PRINT-HEADERS
           PERFORM A300-PRINT-DETAIL-LINE.
A300-PRINT-DETAIL-LINE.
    MOVE I-PURCHASE-DATE    TO DL-DATE.
    MOVE I-PURCHASE-AMOUNT  TO DL-AMT.
    MOVE I-PURCHASE-DESCRIP TO DL-DESC.
    WRITE PRINT-LINE FROM DETAIL-LINE
                      AT END-OF-PAGE PERFORM A500-PRINT-HEADERS.
    ADD I-PURCHASE-AMOUNT TO TOTAL-PURCHASES.
A400-PRINT-TOTALS.
    MOVE TOTAL-PURCHASES TO TL.
    COMPUTE THESE-MANY = 25 - LINAGE-COUNTER.
    WRITE PRINT-LINE FROM TOTAL-LINE AFTER ADVANCING THESE-MANY LINES.
    MOVE 0 TO TOTAL-PURCHASES.
A500-PRINT-HEADERS.
    ADD 1 TO PAGE-NUMBER.
    MOVE PAGE-NUMBER TO H1-PAGE.
    WRITE PRINT-LINE FROM HEAD-1 AFTER ADVANCING PAGE.
    WRITE PRINT-LINE FROM HEAD-2.
    MOVE SPACES TO PRINT-LINE.
    WRITE PRINT-LINE.
    WRITE PRINT-LINE FROM HEAD-3.
    WRITE PRINT-LINE FROM HEAD-4.
    WRITE PRINT-LINE FROM HEAD-5.
    WRITE PRINT-LINE FROM HEAD-6.
A600-SET-UP-HEADERS.
    MOVE I-FIRST          TO H3-FNAME.
    MOVE I-MID            TO H3-MNAME.
    MOVE I-LAST           TO H3-LNAME.
    MOVE I-ACCOUNT-NUMBER TO H3-NUM.
    MOVE I-STREET         TO H4-STRT.
    MOVE I-CITY           TO H4-CITY.
    MOVE I-STATE          TO H4-STATE.
    MOVE I-ZIP            TO H4-ZIP.
```

16.6.6 Printing a Linage-File Report

The default PRINT command inserts a page ejection when a form nears the end of a page. Therefore, when the default PRINT command refers to a linage-file report, it can change the report's page spacing.

To print a linage-file report, use the /NOFEED qualifier with the DCL PRINT command as follows:

```
PRINT report-file-specification/NOFEED
```

The LINAGE clause causes a VAX COBOL report file to be in print-file format. (See Chapter 8 for more information.) When a WRITE statement positions the file to the top of the next logical page, the device is positioned by line spacing rather than by page ejection or form feed.

For more information on printing your report, see Section 16.7.

16.6.7 A Linage-File Report Example

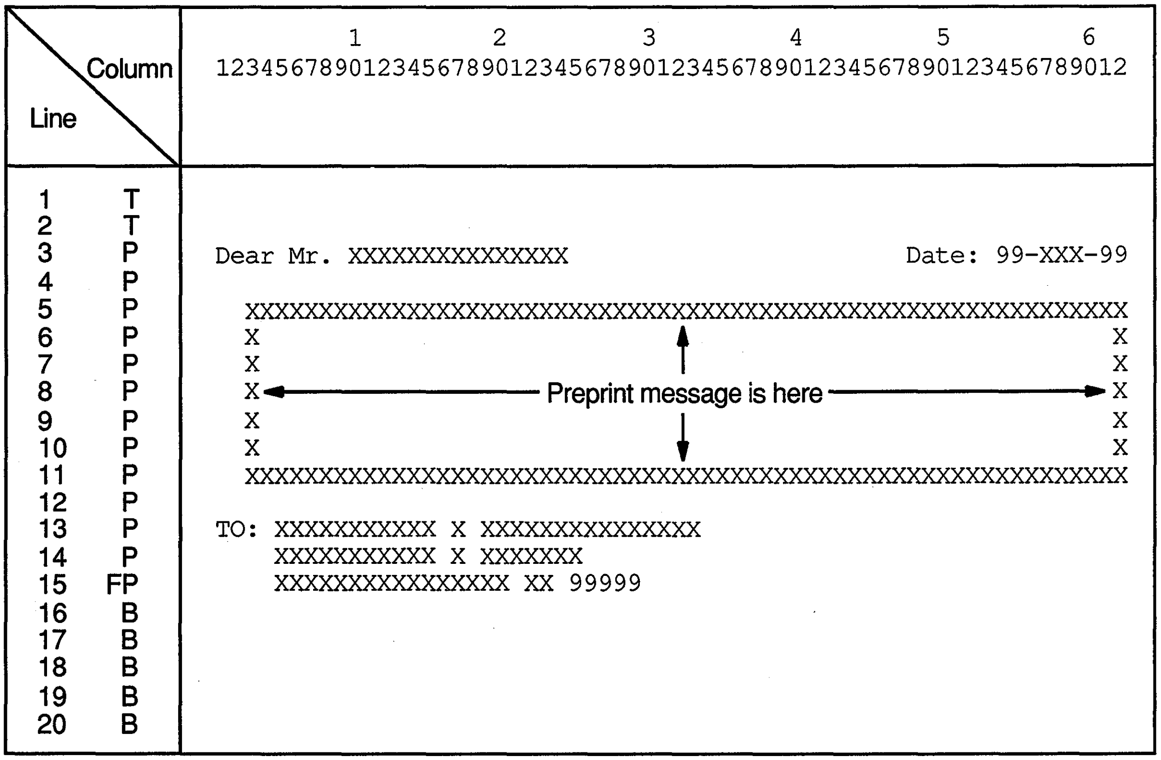
Example 16–4 shows a VAX COBOL program that produces a linage-file report.

The LINAGE clause in the following File Description entry defines the logical page areas shown in Figure 16–9:

```
FD  MINIF1-REPORT
    LINAGE IS 13 LINES
                       LINES AT TOP      2
                       LINES AT BOTTOM   5.
```

Figure 16–9 shows a 20-line logical page that includes a top margin (T), a page body (P), a footing area (F), and a bottom margin (B).

Figure 16–9: A 20-Line Logical Page



```
 Column            1         2         3         4         5         6
 Line     12345678901234567890123456789012345678901234567890123456789012

  1  T
  2  T
  3  P    Dear Mr. XXXXXXXXXXXXXXX                          Date: 99-XXX-99
  4  P
  5  P      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  6  P      X                                                          X
  7  P      X                                                          X
  8  P      X                                                          X
  9  P      X                                                          X
 10  P      X                                                          X
 11  P      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 12  P
 13  P    TO: XXXXXXXXXXX X XXXXXXXXXXXXXXX
 14  P        XXXXXXXXXXX X XXXXXXX
 15  FP       XXXXXXXXXXXXXXXX XX 99999
 16  B
 17  B
 18  B
 19  B
 20  B
```

Legend: T = Top margin = lines 1 and 2
P = Page body = lines 3 through 15
F = Footing area = line 15
B = Bottom margin = lines 16 through 20

ZK–6086–GE

The first line to which the logical page can be positioned is the third line on the page; this is the first print line. The page-overflow condition occurs when a WRITE statement causes the LINAGE-COUNTER value to equal 15. Line 15 is the last line on the page on which text can be written. The page advances to the next logical page when a WRITE statement causes the LINAGE-COUNTER value to exceed 15. The pointer is then positioned on the first print line of the next logical page.

Example 16–4: Programming a 20-Line Logical Page Defined by the LINAGE Clause

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPLINAG.
*  Print the report - PRINT MINIF1.DAT/NOFEED
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE   ASSIGN TO "REPIN.DAT".
    SELECT MINIF1-REPORT ASSIGN TO "MINIF1.DAT".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST                      PIC X(10).
        03  I-MID                        PIC X.
        03  I-LAST                       PIC X(15).
    02  I-ADDRESS.
        03  I-STREET                     PIC X(20).
        03  I-CITY                       PIC X(15).
        03  I-STATE                      PIC XX.
        03  I-ZIP                        PIC 99999.
FD  MINIF1-REPORT
    LINAGE IS 13 LINES
           LINES AT TOP    2
           LINES AT BOTTOM 5.
01  MINIF1-PRINT-LINE                    PIC X(80).
WORKING-STORAGE SECTION.
01  END-OF-FILE                          PIC  X     VALUE SPACE.
01  LINE-UP-OK                           PIC  X     VALUE SPACE.
01  MINIF1-LINE-3.
    02  FILLER                           PIC X(9)   VALUE SPACES.
    02  MINIF1-LAST                      PIC X(15).
    02  FILLER                           PIC X(23)  VALUE SPACES.
    02  FILLER                           PIC X(6)   VALUE "Date: ".
    02  MINIF1-DATE                      PIC 99/99/99.
01  MINIF1-LINE-13.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-NAME                      PIC X(26).
01  MINIF1-LINE-14.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-STREET                    PIC X(20).
01  MINIF1-LINE-15.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-CITY                      PIC X(15).
    02  FILLER                           PIC X      VALUE SPACE.
    02  MINIF1-STATE                     PIC XX.
    02  FILLER                           PIC X      VALUE SPACE.
    02  MINIF1-ZIP                       PIC 99999.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT MINIF1-REPORT.
    ACCEPT MINIF1-DATE FROM DATE.
    PERFORM A300-FORM-LINE-UP UNTIL LINE-UP-OK = "Y".
    OPEN INPUT  INPUT-FILE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A010-WRAP-UP.
    CLOSE INPUT-FILE
          MINIF1-REPORT.
    DISPLAY "END OF JOB".
    STOP RUN.
```

(continued on next page)

Example 16–4 (Cont.): Programming a 20-Line Logical Page Defined by the LINAGE Clause

```
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       PERFORM A200-PRINT-REPORT.
A200-PRINT-REPORT.
    MOVE I-LAST          TO MINIF1-LAST.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-3 BEFORE ADVANCING 1 LINE.
    MOVE SPACES TO MINIF1-PRINT-LINE.
    WRITE MINIF1-PRINT-LINE AFTER ADVANCING 9 LINES.
    MOVE I-NAME          TO MINIF1-NAME.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-13 BEFORE ADVANCING 1 LINE.
    MOVE I-STREET        TO MINIF1-STREET.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-14 BEFORE ADVANCING 1 LINE.
    MOVE I-CITY          TO MINIF1-CITY.
    MOVE I-STATE     TO MINIF1-STATE.
    MOVE I-ZIP       TO MINIF1-ZIP.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-15 BEFORE ADVANCING 1 LINE.
A300-FORM-LINE-UP.
    MOVE ALL "X" TO INPUT-RECORD.
    PERFORM A200-PRINT-REPORT 3 TIMES.
    DISPLAY "Is Alignment OK? (Y/N): " WITH NO ADVANCING.
    ACCEPT LINE-UP-OK.
```

16.7 Modes for Printing Reports

Either your VAX COBOL program can allocate a printer directly and immediately produce the report, or it can spool the report to a mass storage device for printing later. Section 16.7.1 and Section 16.7.2 describe these two modes of printing. Note that spooling the report to a mass storage device makes better use of system resources than allocating a printer directly.

16.7.1 Directly Allocating a Printer

To directly allocate a printer, your VAX COBOL program must include the printer's device name in the file specification for the report file as follows:

```
SELECT REPORT-FILE ASSIGN TO "LP:".
```

Directly allocating the printer has the following advantages:

- Results are immediate.
- Numbers on a preprinted form are associated with a record in a file (as in the case of payroll checks). For example, as the operator opens each box of forms and mounts them in the printer (or remounts them if a paper jam occurs), your program can request and accept the starting number from each new box of forms. If the program then outputs a record for each printed form and includes the form number in the record, you establish an immediate audit trail.

Directly allocating the printer has the following disadvantages:

- Either you must wait until all printer requests from the system spooler are completed, or you must change job priorities.

- You tie up the printer for as long as your job runs. If your program does computations and runs for a long time, you could significantly reduce your installation's pages-printed-per-day production schedule.
- You do not have a backup report file in the event of power failure or other unforeseen circumstances. Therefore, if your job fails, you must begin again.

16.7.2 Spooling to a Mass Storage Device

To spool your report to a mass storage device (such as a disk or magnetic tape) for later printing, your VAX COBOL program must include a file specification in the report file section. For example, to spool JAN28P.DAT to the device DB1:, you would include the following code in your program:

```
SELECT REPORT-FILE ASSIGN TO "DB1:JAN28P".
```

Spooling to a mass storage device has the following advantages:

- You can run your job at any time regardless of other printer activity and printer status.
- Your application program does not make immediate resource demands on the printer.
- You can schedule the printing based on production and shop requirements, and print the file according to your priority needs.
- You optimize use of the printer. Spooling results in printing the maximum number of lines per minute.
- You have a backup of the file.

Spooling to a mass storage device has the following disadvantages:

- You do not see immediate results.
- It is difficult and expensive to input preprinted form numbers (for example, check numbers) from your forms into your report file.

16.8 Programming the Report Writer Report

Report Writer allows you to describe the physical appearance of a report's format. To do this, you specify the Report Writer statements that describe the report's contents and control in the Report Section of the Data Division. These statements replace many complex, detailed procedures that you would otherwise have to include in a conventional or linage-file report.

The following sections explain how to produce a report with the Report Writer. These sections discuss how to do the following:

- Use the REPORT clause
- Define the Report Section and the report file
- Define the Report Writer logical page
- Specify multiple reports
- Define and increment totals
- Process a Report Writer report
- Select a Report Writer type

Detailed examples using Report Writer are documented in Section 16.9.

16.8.1 Using the REPORT Clause in the File Section

To create a report with Report Writer, you must write a report to a specific file. That file is described by a File Description (FD) entry; however, unlike a conventional or linage-file report, your FD entry for a Report Writer file must contain the REPORT clause, and you must assign a name for each report in the REPORT clause.

For instance, in the following example, the File Description on the left does not specify Report Writer; however, the example on the right correctly shows a Report Writer File Section entry:

```
FD  SALES-REPORT                       FD  SALES-REPORT
     .                                  .
     .                                  .
     .                                  .
01  PRINT-AREA     PIC X(133).         REPORT IS MASTER-LIST.
```

To completely describe the report that you specify in the REPORT clause, you must define a Report Section. Section 16.8.2 discusses the Report Section.

16.8.2 Defining the Report Section and the Report File

A Report Section provides specific information about the report that is specified with the REPORT clause. Each report listed in the File Section must be defined in the Report Section.

To define a report, use a Report Description (RD) entry followed by one or more Report Group Description entries (01-level) in the Report Section. For example:

```
FILE SECTION.

FD  SALES-REPORT
    REPORT IS MASTER-LIST.
     .
     .
     .
REPORT SECTION.

RD  MASTER-LIST
    PAGE LIMIT IS   66
      HEADING        1
      FIRST DETAIL  13
      LAST DETAIL   30
      FOOTING       50.
```

The RD supplies information about the format of the printed page and the organization of the subdivisions (see Section 16.8.4).

16.8.3 Defining a Report Writer Logical Page with the PAGE Clause

To define the logical page for a Report Writer report, you use the PAGE clause. This clause enables you to specify the number of lines on a page and the format of that page. For example, the PAGE clause allows you to specify where the heading, detail, and footing appear on the printed page. If you want to use vertical formatting, you must use the PAGE clause.

The RD entry example in Section 16.8.2 contains the following PAGE clause information:

RD Entry Line	Meaning	
PAGE LIMIT IS	66	Maximum number of lines per page is 66
HEADING	1	Line number on which the first report heading (RH) or page heading (PH) should print on each page
FIRST DETAIL	13	First line number on which a control heading (CH), detail (DE), or control footing (CF) should print on a page
LAST DETAIL	30	Last line number on which a CH or DE can print on a page
FOOTING	50	Last line number on which a control footing (CF) can print on a page (if specified, page footing (PF) and report footing (RF) report groups follow the line number shown in FOOTING)

The PAGE LIMIT clause line numbers are in ascending order and must not exceed the number specified in the PAGE LIMIT clause (in this example, 66 lines).

Section 16.8.4 describes report group entries in more detail.

16.8.4 Describing Report Group Description Entries

In a Report Writer program, report groups are the basic elements that make up the logical page. There are seven types of report groups, which consist of one or more report lines printed as a complete unit (for example, a page heading). Each report line can be subdivided into data items or fields.

The seven types of report groups are as follows:

Report Group Type	Description
REPORT HEADING	Prints a title or any other information that pertains to the entire report
PAGE HEADING	Prints a page heading and column headings
CONTROL HEADING	Prints a heading when a control break occurs
DETAIL	Prints the primary data of the report
CONTROL FOOTING	Prints totals when a control break occurs
PAGE FOOTING	Prints totals or comments at the bottom of each page
REPORT FOOTING	Prints trailer information for the report

A Report Writer program can include both printable report groups and null report groups. Null report groups are groups that do not print but are used for control breaks.

Figure 16–10 shows the report group presentation order found on a logical page. You must code at least one DETAIL report group (printable or null) in your program to produce a report. All other report groups are optional. Note that you can code a report group by using the abbreviations shown in Figure 16–10.

Figure 16–10: Presentation Order for a Logical Page

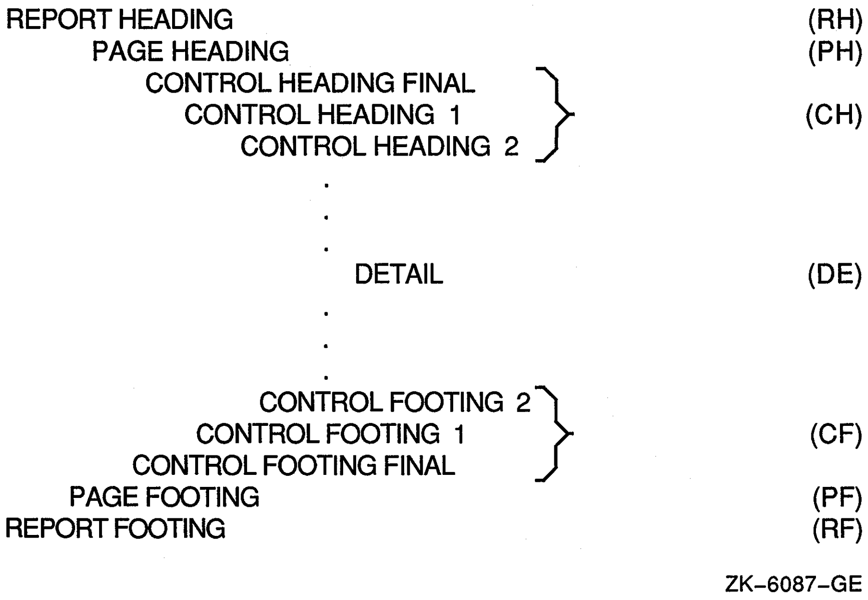
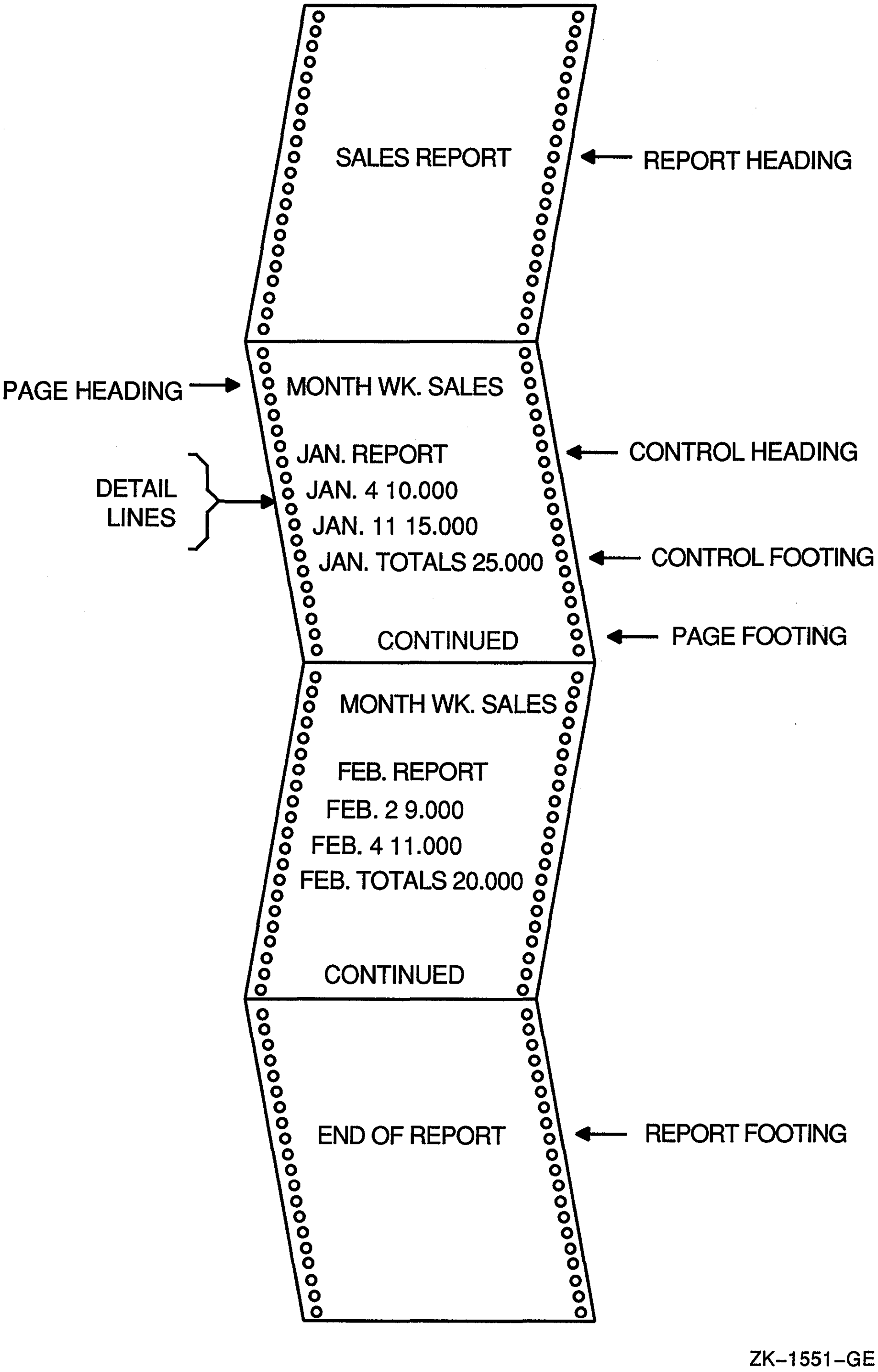


Figure 16–11 shows a report that uses all seven of the report groups listed in the preceding table.

Figure 16–11: Sample Report Using All Seven Report Groups



To code report groups, you use an 01-level entry to describe the physical and logical characteristics of the report group and the Report Writer TYPE clause to indicate the type of the report group. The TYPE clause can be preceded by a user-defined report group name. The CONTROL HEADING and FOOTING report groups use data names that are also specified as CONTROL clause names in the Report Description entry (see Section 16.8.10 for CONTROL clause information).

The following example shows how to use the TYPE and CONTROL clauses:

```
DATA DIVISION.

REPORT SECTION.

01    REPORT-HEADER TYPE IS REPORT HEADING.
01    PAGE-HEADER TYPE IS PAGE HEADING.
01    CONTROL-HEADER TYPE IS CONTROL HEADING CONTROL-NAME-1.
01    DETAIL-LINE TYPE IS DETAIL.
01    CONTROL-FOOTER TYPE IS CONTROL FOOTING CONTROL-NAME-2.
01    PAGE-FOOTER TYPE IS PAGE FOOTING.
01    REPORT-FOOTER TYPE IS REPORT FOOTING.
```

16.8.5 Vertical Spacing for the Logical Page

You use the LINE clause for positioning vertical lines within a report group or for indicating vertical line space between two report groups. The LINE clause indicates the start of an absolute print line (a specific line on a page) or where a relative print line (an increment to the last line printed) is to print on the page. You can use this clause with all report groups.

In the following example, the LINE clause indicates that this report group begins on absolute line number 5 on a page. LINE IS 7 indicates that this report group has a second line of data found on absolute line number 7. Absolute line numbers must be specified in ascending order.

```
01    PAGE-HEADER TYPE IS PAGE HEADING.
      02  LINE IS 5.
      .
      .
      .
      02  LINE IS 7.
```

In the following example the term PLUS in the LINE clause indicates that DETAIL-LINE prints two lines after the last line of the previous report group. If you used a CONTROL HEADING report group that ended on line 20 before DETAIL-LINE, then DETAIL-LINE would print beginning on line 22.

```
01    DETAIL-LINE TYPE IS DETAIL.
      02  LINE PLUS 2.
```

In the following example the LINE clause specifies that the REPORT FOOTING report group prints on line 32 of the next page:

```
01  REPORT-FOOTER TYPE IS REPORT FOOTING.
    02  LINE IS 32 ON NEXT PAGE.
```

You can code NEXT PAGE only for CONTROL HEADING, DETAIL, CONTROL FOOTING, and REPORT FOOTING groups, and only in the first LINE clause in that report group entry.

Within the report group, absolute line numbers must be in ascending order (although not consecutive) and must precede all relative line numbers.

You can use the NEXT GROUP clause instead of the LINE clause to control line spacing. In NEXT GROUP clause, you specify the amount of vertical line space you want following one report group and before the next. You use this clause in the report group that will have the space following it, as shown in the following example:

```
01    CONTROL-HEADER TYPE IS CONTROL HEADING CONTROL-NAME-1
      NEXT GROUP PLUS 4.

01    DETAIL-LINE TYPE IS DETAIL.
```

This example indicates relative line use. The report group (DETAIL) immediately following this CONTROL HEADING report group will print on the fourth line after the CH's last print line.

You can also specify absolute line spacing with the NEXT GROUP clause. An absolute line example—NEXT GROUP IS 10—places the next report group on line 10 of the page. In addition you can use NEXT GROUP NEXT PAGE, which causes a page-eject to occur before the NEXT GROUP report group prints.

NEXT GROUP can be coded only for REPORT HEADING, CONTROL HEADING, DETAIL, CONTROL FOOTING, and PAGE FOOTING report groups, and only at the 01 level.

A PAGE FOOTING report group must not specify the NEXT PAGE phrase of the NEXT GROUP clause.

Both the LINE and NEXT GROUP clauses must adhere to the page parameters specified in the PAGE clause in the RD entry.

In addition, the Report Writer facility keeps track of the number of lines printed or skipped on each page by using the LINE-COUNTER. LINE-COUNTER references a special register that the compiler generates for each Report Description entry in the Report Section. The Report Writer maintains the value of LINE-COUNTER and uses this value to determine the vertical positioning of a report.

16.8.6 Horizontal Spacing for the Logical Page

The COLUMN NUMBER clause defines the horizontal location of items within a report line.

You use the COLUMN NUMBER clause only at the elementary level. This clause must appear in or be subordinate to an entry that contains a LINE NUMBER clause. Within the description of a report line, the COLUMN NUMBER clauses must show values in ascending column order. Column numbers must be positive integer literals with values from 1 to the maximum number of print positions on the printer. For example:

```
01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 1.
        02 COLUMN 1     PIC X(15)          SOURCE LAST-NAME.
        02 COLUMN 17    PIC X(10)          SOURCE FIRST-NAME.
        02 COLUMN 28    PIC XX             SOURCE MIDDLE-INIT.
        02 COLUMN 40    PIC X(20)          SOURCE ADDRESS.
        02 COLUMN 97    PIC $$$,$$$,$$$.99 SOURCE INVOICE-SALES.
```

Omitting the COLUMN clause creates a null (nonprinting) report item. Null report items are used to accumulate totals and force control breaks as described in Section 16.8.4.

The following example shows the use of a COLUMN NUMBER clause in a LINE clause:

```
02  LINE 15 COLUMN 1 PIC X(12) VALUE "SALES TOTALS".
```

The previous example results in the following output:

```
                1         2         3         4
column 1234567890123456789012345678901234567890
       SALES TOTALS
```

In the next example, the COLUMN NUMBER clauses are subordinate to a LINE NUMBER clause:

```
02  LINE 5 ON NEXT PAGE.
   03     COLUMN 1  PIC X(12)       VALUE "(Cust-Number".
   03     COLUMN 14 PIC 9999        SOURCE CUST-NUM.
   03     COLUMN 18 PIC X           VALUE ")".
   03     COLUMN 20 PIC X(15)       VALUE "TOTAL PURCHASES".
   03     COLUMN 36 PIC $$$$,$$$.99 SUM TOT-PURCHS.
```

The previous example produces the following output:

```
                1         2         3         4
column 1234567890123456789012345678901234567890123456
       (Cust-Number 1234) TOTAL PURCHASES   $1,432.99
```

16.8.7 Assigning a Value in a Print Line

In a Report Writer program, one way you specify a value for an item is to use the VALUE clause. This clause designates that the data item has a constant literal value. You often use this clause with REPORT HEADING and PAGE HEADING report groups, because the data in these groups is usually constant, as shown in the following example:

```
01      TYPE IS PAGE HEADING.
        02    LINE  5.
           03 COLUMN 1
                 PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
           03 COLUMN 40
                 PIC X(5)  VALUE "SALES".
```

The previous example results in the following output:

Output:

```
                1         2         3         4         5
column 12345678901234567890123456789012345678901234567890
       CUSTOMER MASTER FILE REPORT            SALES
```

16.8.8 Defining the Source for a Print Field

To assign a variable value to an item in a Report Writer program, you use the SOURCE clause.

The SOURCE clause, written in the Report Section, is analogous to the MOVE statement.

The clause names a data item that is moved to a specified position on the print line. Before an item that contains a SOURCE clause is printed, the Report Writer moves the value in the field named in the SOURCE clause into the print line at the print position specified by the COLUMN clause, as shown in the following example. Any data editing specified by the PICTURE clause is performed before the data is moved to the print line.

```
01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 1.
        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT.
        02 COLUMN 35    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 55    PIC X(20) SOURCE CITY.
        02 COLUMN 75    PIC XX    SOURCE STATE.
        02 COLUMN 78    PIC 99999 SOURCE ZIP.
```

You can also code a SOURCE clause with PAGE-COUNTER or LINE-COUNTER as its operand, as the following example shows. PAGE-COUNTER references a special register created by the compiler for each Report Description entry in the Report Section. This counter automatically increments by 1 each time the Report Writer executes a page advance. The use of PAGE-COUNTER eliminates Procedure Division statements you normally would write to explicitly count pages, as shown in the following example:

```
01      TYPE IS PAGE HEADING.
        02      LINE 5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 52
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 57
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
```

This example produces the following output:

```
                  1         2         3         4         5         6
   column 123456789012345678901234567890123456789012345678901234567890
          CUSTOMER MASTER FILE REPORT                        PAGE    9
```

16.8.9 Specifying Multiple Reports

To include two or more reports in one file, you specify multiple identifiers in the REPORTS clause and provide multiple RDs in the Report Section.

To identify the lines of two or more reports in one file, you use the CODE clause, as shown in the following example:

```
FILE SECTION.
FD   REPORT-FILE
     REPORTS ARE REPORT1
                 REPORT2
                 REPORT3.
REPORT SECTION.
RD   REPORT1...
     CODE"AA".

RD   REPORT2...
     CODE"BB".

RD   REPORT3...
     CODE"CC".
```

The CODE clause specifies a 2-character nonnumeric literal that identifies each print line as belonging to a specific report. When the CODE clause is specified, the literal is automatically placed in the first two character positions of each Report Writer logical record. Note that if the clause is specified for any report in a file, it must be used for all reports in that file.

16.8.10 Generating and Controlling Report Headings and Footings

When you write a report that has control headings and/or footings, you must use the CONTROL clause to create control levels that determine subsequent headings and totals.

The CONTROL clause, found in the RD entry, names data items that indicate when control breaks occur. The CONTROL clause specifies the data items in major to minor order. You must define these CONTROL data items, or control names, in the Data Division, and reference them in the appropriate CONTROL HEADING and FOOTING report groups.

When the value of a control name changes, a control break occurs. The Report Writer only acknowledges this break when you execute a GENERATE or TERMINATE statement for the report, which causes the information related to that CONTROL report group to be printed.

In the following example, the report defines two control totals (MONTH-CONTRL and WEEK-CONTRL) in the CONTROL clause. The source of these control totals is in an input file named IN-FILE. The file is sorted in ascending sequence by MONTH-CONTRL and WEEK-CONTRL. The Report Writer automatically monitors these fields in the input file for any changes. If a new record contains different data than the previous record read, Report Writer triggers a control break.

```
FD     IN-FILE.
01     INPUT-RECORD.
       02  MONTH-CONTRL     PIC...
       02  ...
       02  ...
       02  WEEK-CONTRL      PIC...
FD     REPORT-FILE   REPORT IS SALES-REPORT.
       .
       .
       .
REPORT SECTION.
RD     SALES-REPORT.
       CONTROLS ARE MONTH-CONTRL, WEEK-CONTRL.
01     DETAIL-LINE TYPE IS DETAIL.

01     TYPE IS CONTROL FOOTING MONTH-CONTRL.

01     TYPE IS CONTROL FOOTING WEEK-CONTRL.
```

In this example, if the value in WEEK-CONTRL changes, a break occurs and Report Writer processes the CONTROL FOOTING WEEK-CONTRL report group. If the value in MONTH-CONTRL changes, a break occurs and Report Writer processes both CONTROL FOOTING report groups, because a break in any control field implies a break in all lower-order control fields as well.

The same process occurs if you include similar CONTROL HEADING report groups. However, CONTROL HEADING control breaks occur from a break to minor levels, while CONTROL FOOTING control breaks occur from a break to major levels.

The following example demonstrates the use of FINAL, a special control field that names the most major control field. You specify FINAL once, in the CONTROL clause, as the most major control level. When you code FINAL, a FINAL control break and subsequent FINAL headings and footings occur during program execution: once at the beginning of the report (as part of the report group, CONTROL HEADING FINAL), before the first detail line is printed; and once at the end of the report (as part of the report group, CONTROL FOOTING FINAL), after the last detail line is printed.

```
01   TYPE CONTROL FOOTING FINAL.
     02  LINE 58.
         04  COLUMN 1 PIC X(32) VALUE
            "TOTAL SALES FOR YEAR-TO-DATE WAS".
         04  COLUMN 45 PIC 9(6).99 SOURCE TOTAL-SALES.
```

This example produces the following output:

```
                1         2         3         4         5
column 1234567890123456789012345678901234567890123456789012345
       TOTAL SALES FOR YEAR-TO-DATE WAS            953208.90
```

16.8.11 Defining and Incrementing Totals

In addition to using either the VALUE or SOURCE clause to assign a value to a report item, you can use the SUM clause to accumulate values of report items. This clause establishes a sum counter that is automatically summed during the processing of the report. You code a SUM clause only in a TYPE CONTROL FOOTING report group.

The identifiers of the SUM clause are either elementary numeric data items not in the Report Section or other sum counters in the Report Section that are at the same or lower level in the control hierarchy of the report, as specified in the CONTROL clause.

The SUM clause provides three forms of sum accumulation: subtotaling, crossfooting, and rolling-forward. See Section 16.3 for further details.

16.8.11.1 Subtotaling

In subtotaling, the SUM clause references elementary numeric data items that appear in the File or Working-Storage Sections and then generates sums of those items.

In the following example, EACH-WEEK represents a CONTROL clause name. COST represents a numeric data item in the File Section that indicates weekly expenses for a company. DAY and MONTH indicate the particular day and month.

```
01   TYPE CONTROL FOOTING EACH-WEEK.
     02  LINE PLUS 2.
        03  COLUMN 1    PIC IS X(30)
            VALUE IS "TOTAL EXPENSES FOR WEEK/ENDING".
        03  COLUMN 33   PIC IS X(4)   SOURCE IS MONTH.
        03  COLUMN 39   PIC IS X(2)   SOURCE IS DAY.
        03  WEEK-AMT  COLUMN 45
                       PIC ZZ9.99 SUM COST.
```

This example produces the following subtotal output:

```
                1         2         3         4         5
column 12345678901234567890123456789012345678901234567890
       TOTAL EXPENSES FOR WEEK/ENDING  JULY  02    799.23
```

When the value of EACH-WEEK changes, a control break occurs that causes this TYPE CONTROL FOOTING report group to print. The value of the sum counter is edited according to the PIC clause accompanying the SUM clause. Then the sum lines are printed in the location specified by the items' LINE and COLUMN clauses.

16.8.11.2 Crossfooting

In crossfooting, the SUM clause adds all the sum counters in the same CONTROL FOOTING report group and automatically creates another sum counter.

In the following example, the CONTROL FOOTING group shows both subtotaling (SALES-1) and crossfooting (SALES-2):

```
01  TYPE DETAIL LINE PLUS 1.
    05  COLUMN 15 PIC 999.99 SOURCE BRANCH1-SALES.
    05  COLUMN 25 PIC 999.99 SOURCE BRANCH2-SALES.

01  TYPE CONTROL FOOTING BRANCH-TOTAL LINE PLUS 2.
    05  SALES-1  COLUMN 15 PIC 999.99 SUM BRANCH1-SALES.
    05  SALES-2  COLUMN 25 PIC 999.99 SUM BRANCH2-SALES.
    05  SALES-TOT  COLUMN 50 PIC 999.99 SUM SALES-1, SALES-2.
```

The SALES-1 sum contains the total of the BRANCH1-SALES column and the SALES-2 sum contains the total of the BRANCH2-SALES column (both sums are subtotals). SALES-TOT contains the sum of SALES-1 and SALES-2; it is a crossfooting.

The crossfooting ouput is as follows:

```
                1         2         3         4         5         6
column 123456789012345678901234567890123456789012345678901234567890
                     125.00    300.00                   425.00
```

16.8.11.3 Rolling-Forward

When rolling totals forward, the SUM clause adds a sum counter from a lower-level CONTROL FOOTING report group to a sum counter in a higher-level footing group. The control logic and necessary control hierarchy for rolling counters forward begins in the CONTROL clause.

In the following example, WEEK-AMT is a sum counter found in the lower-level CONTROL FOOTING group, EACH-WEEK. This sum counter is named in the SUM clause in the higher-level CONTROL FOOTING report group, EACH-MONTH. The value of each WEEK-AMT sum is added to the higher-level counter just before the lower-level CONTROL FOOTING group is printed.

```
RD   EXPENSE-FILE.
     .
     .
     .
     CONTROLS ARE EACH-MONTH, EACH-WEEK.
01   TYPE CONTROL FOOTING EACH-WEEK.
     02  LINE PLUS 2.
        03  COLUMN 1            PIC IS X(30)
              VALUE IS "TOTAL EXPENSES FOR WEEK/ENDING".
        03  COLUMN 33           PIC IS X(9)    SOURCE IS MONTH.
        03  COLUMN 42           PIC IS X(2)    SOURCE IS DAY.
        03  WEEK-AMT  COLUMN 45 PIC ZZ9.99     SUM COST.

01   TYPE CONTROL FOOTING EACH-MONTH.
     02  LINE PLUS 2.
        03  COLUMN 10  PIC X(18)  VALUE IS "TOTAL EXPENSES FOR".
        03  COLUMN 29  PIC X(9)   SOURCE MONTH.
        03  COLUMN 50  PIC ZZ9.99 SUM WEEK-AMT.
```

The following output is a result of rolling the totals forward:

```
                1         2         3         4         5
column 123456789012345678901234567890123456789012345678901234567890123456789012345
       TOTAL EXPENSES FOR DECEMBER             379.19
```

16.8.11.4 RESET Option

When a CONTROL FOOTING group is printed, the SUM counter in that group is automatically reset to zero. If you want to specify when a SUM counter is reset to zero, use the RESET phrase. RESET names a data item in a higher-level CONTROL FOOTING that will cause the SUM counter to be reset to zero. RESET is used only with a SUM clause.

The following example sums SALES, resetting the counter to zero only when it encounters a new year (YEAR). This prevents the sum from being reset to zero when a new month causes a control break, giving a running total of the months within the year.

```
RD   SALES-REPORT.
     .
     .
     .
     CONTROLS ARE YEAR, EACH-MONTH, EACH-WEEK.

     .
     .
     .
01   TYPE CONTROL FOOTING EACH-MONTH
     02  COLUMN 10   PIC ZZ9.99  SUM SALES RESET ON YEAR.
```

16.8.11.5 UPON Option

Another SUM option is the UPON phrase. This phrase allows selective subtotaling for the DETAIL Report Group named in the phrase. When you use the UPON phrase, you cannot reference the sum counter in the SUM clause. You can use any File or Working-Storage Section elementary numeric data item.

When you code the UPON option with the SUM clause, the value of the data items of the SUM clause will be added whenever the TYPE DETAIL report group you name in the UPON option is generated.

```
WORKING-STORAGE SECTION.
     .
     .
     .
01   WORK-AREA.
     .
     .
     .
     03  ADD-COUNTER            PIC 9     VALUE 1.
REPORT SECTION.
     .
     .
     .
01   FIRST-DETAIL-LINE TYPE IS DETAIL LINE IS PLUS 2.
     .
     .
     .
01   TYPE IS CONTROL FOOTING FINAL.
     05  LINE IS PLUS 3.
     .
     .
     .
     05  LINE PLUS 2.
         10  COLUMN 5           PIC Z(3)9  SUM ADD-COUNTER
                                           UPON FIRST-DETAIL-LINE.
```

In the preceding example, the value of ADD-COUNTER is added to the CONTROL FOOTING FINAL counter every time the FIRST-DETAIL-LINE report group is generated.

16.8.12 Restricting Print Items

In a Report Writer program, the GROUP INDICATE clause eliminates repeated information from report detail lines by allowing an elementary item in a DETAIL report group to be printed only the first time after a control or page break. The following example illustrates the use of this clause:

```
01  DETAIL-LINE TYPE DETAIL LINE PLUS 1.

    05  COLUMN 1 GROUP INDICATE PIC X(6) VALUE "SALES:".
*        (prints only the first time after a control or page break)

    05  COLUMN 10 PIC X(10) SOURCE BRANCH.
*          (prints each time)
```

These statements produce the following lines:

```
SALES:   BRANCH-A

         BRANCH-B

         BRANCH-C
```

The next two examples are nearly identical programs; the only difference is the use of the GROUP INDICATE clause in the second example.

The following program does not contain a GROUP INDICATE clause:

```
01   DETAIL-LINE TYPE IS DETAIL
                 LINE IS PLUS 1.
     02  COLUMN 1  PIC X(15)
                 SOURCE A-NAME.
     02  COLUMN 20 PIC 9(6)
                 SOURCE A-REG-NO.
```

It produces the following output:

```
         1         2         3
123456789012345678901234567890
Name               Registration
                   Number

Rolans  R.         123456
Rolans  R.         123456
Rolans  R.         123456
Vencher R.         654321
Vencher R.         654321
Vencher R.         654321
Vencher R.         654321
Anders J.          987654
Anders J.          987654
Anders J.          987654
```

The following example contains a GROUP INDICATE clause:

```
01   DETAIL-LINE TYPE IS DETAIL
                 LINE IS PLUS 1.
     02  COLUMN 1  PIC X(15)
                 SOURCE A-NAME
                 GROUP INDICATE.
     02  COLUMN 20  PIC 9(6)
                 SOURCE A-REG-NO.
```

With the GROUP INDICATE clause, the program produces the following output:

```
         1         2         3
123456789012345678901234567890
Name               Registration
                   Number

Rolans  R.         123456
                   123456
                   123456
Vencher R.         654321
                   654321
                   654321
                   654321
Anders J.          987654
                   987654
                   987654
```

16.8.13 Processing a Report Writer Report

In a Report Writer program you usually use the following five statements:

- INITIATE
- GENERATE
- TERMINATE
- USE BEFORE REPORTING
- SUPPRESS

You must use the INITIATE, GENERATE, and TERMINATE statements. The USE BEFORE REPORTING and the SUPPRESS statements are optional.

Before any Report Writer statement is executed, the report file must be open.

16.8.13.1 Initiating the Report

The INITIATE statement begins the report processing and is executed before any GENERATE or TERMINATE statements. The report name used in this statement is specified in the RD entry in the Report Section and in the REPORT clause of the FD entry for the file to which the report is written.

INITIATE sets PAGE-COUNTER to 1, LINE-COUNTER to zero, and all SUM counters to zero.

```
PROCEDURE DIVISION.
          .
          .
          .
MAIN SECTION.
000-START.
          OPEN INPUT CUSTOMER-FILE.
          OPEN OUTPUT PRINTER-FILE.
          .
          .
          .
          INITIATE MASTER-LIST.
```

A second INITIATE statement for the same report must not be executed until a TERMINATE statement for the report has been executed (see Section 16.8.13.4).

16.8.13.2 Generating a Report Writer Report

The GENERATE statement prints the report.

You can produce either detail or summary reports depending on the GENERATE identifier. If you code the name of a DETAIL report group with GENERATE, you create a detail report; if you code a report name with GENERATE, you create a summary report.

16.8.13.3 Automatic Operations of the GENERATE Statement

When the first GENERATE statement is executed, the following report groups are printed, if they are specified in the program:

- REPORT HEADING report group
- PAGE HEADING report group
- CONTROL HEADING report group
- For detail reporting, the specified TYPE DETAIL report group

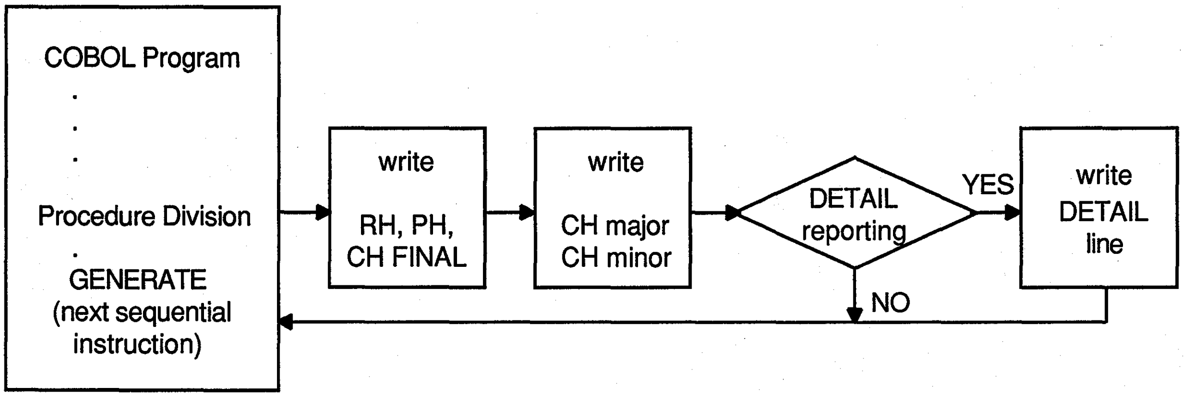
A USE BEFORE REPORTING declarative can also execute just before the associated report group is produced.

NOTE

> Figure 16–12 and Figure 16–13 illustrate the major flow of operations, but do not cover all possible operations associated with a GENERATE statement.

Figure 16–12 shows the sequence of operations for the first GENERATE statement.

Figure 16–12: First GENERATE Statement



ZK–1552–GE

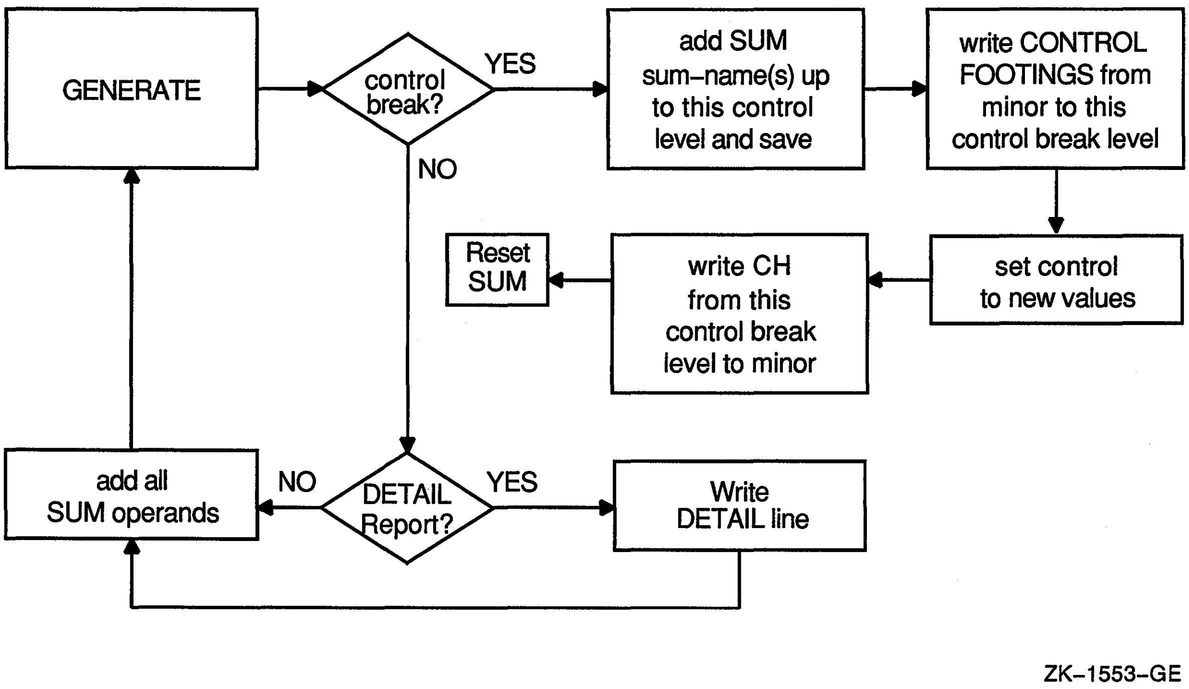
For subsequent GENERATE statements in the program, the following operations take place:

- Any USE BEFORE REPORTING declaratives execute just before the associated report group is produced.
- Any specified control breaks occur.

- CONTROL FOOTING and CONTROL HEADING report groups print after the specified control breaks occur.
- In a detail report, the TYPE DETAIL report groups print.
- SUM operands are incremented.
- Sum counters are reset as specified.

Figure 16–13 shows the sequence of operations for all GENERATE statements except the first. See Figure 16–12 for a comparison with the sequence of operations for the first GENERATE statement.

Figure 16–13: Subsequent GENERATE Statements


GENERATE
control break?
YES
NO
add SUM sum-name(s) up to this control level and save
write CONTROL FOOTINGS from minor to this control break level
set control to new values
write CH from this control break level to minor
Reset SUM
add all SUM operands
NO
DETAIL Report?
YES
Write DETAIL line
ZK-1553-GE

16.8.13.4 Ending Report Writer Processing

The TERMINATE statement completes the processing of a report.

Like INITIATE, the TERMINATE statement report name is specified in the RD entry in the Report Section and in the REPORT clause of the FD entry for the file to which the report is written.

When the TERMINATE statement is executed, breaks occur for all control fields, and all control footings are written; any page footings and report footings are also written.

```
PROCEDURE DIVISION.
.
.
.
300-END-OF-FILE.
    TERMINATE MASTER-LIST.
    CLOSE CUSTOMER-FILE, PRINTER-FILE.
    STOP RUN.
```

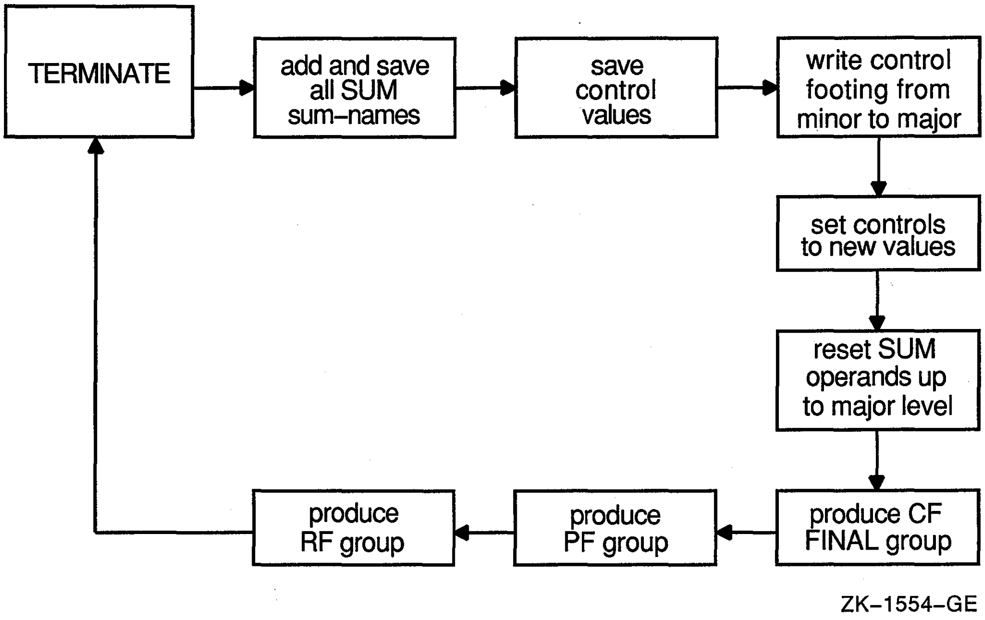
If no GENERATE statement has been executed for the report, the TERMINATE statement does not produce any report groups.

A second TERMINATE statement for the report must not be executed before a second INITIATE statement for the report has been executed.

The TERMINATE statement does not close the report file; a CLOSE statement must be executed after the TERMINATE statement.

Figure 16–14 shows the sequence of operations for TERMINATE.

Figure 16–14: TERMINATE Statement


TERMINATE
add and save all SUM sum–names
save control values
write control footing from minor to major
set controls to new values
reset SUM operands up to major level
produce CF FINAL group
produce PF group
produce RF group
ZK–1554–GE

16.8.13.5 Applying the USE BEFORE REPORTING Statement

In a COBOL program, you specify a Declarative section to define procedures that supplement the standard procedures of the program. Note that in a Report Writer program, you can specify the USE BEFORE REPORTING statement. This USE BEFORE REPORTING statement gives you more control over the data to be printed in a Report Writer program.

The USE BEFORE REPORTING statement:

- Allows you to define declarative procedures
- Causes those procedures to be executed just before a specified report group is printed (this specified report group name is written with the USE statement)

- Lets you modify the data to be printed (for example, where simple sum operations must be augmented by more complex operations involving multiplication, division, and subtraction)
- Lets you suppress printing the report group

The following example indicates that the phrase BEGINNING-OF-REPORT is to be displayed just before the REPORT HEADING group named REPORT-HEADER; the phrase END-OF-REPORT is to be displayed just before the REPORT FOOTING group called REPORT-FOOTER.

```
PROCEDURE DIVISION.
DECLARATIVES.
BOR SECTION.
     USE BEFORE REPORTING REPORT-HEADER.
BOR-A.
     DISPLAY "BEGINNING-OF-REPORT".
EOR SECTION.
     USE BEFORE REPORTING REPORT-FOOTER.
EOR-A.
     DISPLAY "END-OF-REPORT".
END DECLARATIVES.
```

Note that you cannot use INITIATE, GENERATE, or TERMINATE in a Declarative procedure.

16.8.13.6 Suppressing a Report Group

You can also use the SUPPRESS statement in a USE BEFORE REPORTING procedure to suppress the printing of a report group. For example, you can suppress printing of an unnecessary total line, such as a line for a monthly sales total that has only one sale or a line of zeros.

The SUPPRESS statement nullifies any NEXT GROUP and LINE clauses, but leaves the LINE-COUNTER value unchanged.

Note that the SUPPRESS statement applies only to that particular instance of the report group; that group will be printed the next time unless the SUPPRESS statement is executed again.

The SUPPRESS statement has no effect on sum counters.

16.8.14 Selecting a Report Writer Report Type

You can print two types of reports using the Report Writer feature. In a detail report, you print primary data information as well as totals. In a summary report, you print only control heading and footing information (such as report data headings and totals) and exclude detail input record information.

Section 16.9 provides examples of detail and summary reports.

16.8.14.1 Detail Reporting

In detail reporting, at least one printable TYPE DETAIL report group must be specified. A GENERATE statement produces the specified TYPE DETAIL report group and performs all the automatic operations of the Report Writer as specified in the report group entries (see Section 16.8.13.3).

In the following example, DETAIL-LINE is the name of the DETAIL report group. When this GENERATE statement executes, a detail report is printed.

```
200-READ-MASTER.
    READ CUSTOMER-FILE AT END MOVE HIGH-VALUES TO NAME.
    IF NAME NOT = HIGH-VALUES GENERATE DETAIL-LINE.
```

16.8.14.2 Summary Reporting

In summary reporting, the GENERATE statement performs all of the automatic operations of the Report Writer, but does not produce any TYPE DETAIL report groups.

A report name references the name of an RD entry. If MASTER-LIST is an RD entry, then GENERATE MASTER-LIST produces HEADING and FOOTING report groups (in the order defined), but omits DETAIL report group lines.

16.9 Report Writer Examples

This section provides you with the input data and sample reports produced by five Report Writer programs. Each sample report has a program summary section that describes the Report Writer features used in that program; you can examine the summary and output to determine the usage of Report Writer features. Note that each sample report is followed by the program that was used to generate it.

Also, many of the report pages in Reports 2 through 5 have been compressed into fewer pages than you would normally find. For example, a report title page is typically found on a separate page.

NOTE

> The Report Writer produces a report file in print-file format. When the Report Writer positions the file at the top of the next logical page, it positions the pointer by line spacing, rather than page ejection or form feed.
>
> The default VMS PRINT command inserts a form-feed character when a form is within four lines of the bottom. Therefore, when the default PRINT command refers to a Report Writer file, unexpected page spacing can result.
>
> The /NOFEED file qualifier of the PRINT command suppresses the insertion of form-feed characters and prints Report Writer files correctly. Consequently, you should use the /NOFEED qualifier when you use the Report Writer to print a report.

16.9.1 Input Data

The following records are used for the programs in this section.

```
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Adam          Harold    B980 Main Street     Nashua             NH0310223410000002210089002068
Albert        Robert    S100 Meadow Lane     Gardner            MA0142012340000003610090002068
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Allan         David     L10 Wonder Lane      Merrimack          NH0301467800000001241010002068
Amos          James     A71 State Rd         East Westminster   MA0147312341000006410009002068
Amico         Art       A31 Athens Road      Nashua             NH0306089000000007123407002068
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Ames          Alice     J40 Center Road      Nashua             NH0306078900000007100000002078
Alwin         Tom       F400 High Street     Princeton          NJ1234112341000008700001703078
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Berger        Tom       H700 McDonald Lane   Merrimack          NH0306012341000010123416002068
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Ames          Alice     J40 Center Road      Nashua             NH0306078900000007100000002078
Carter        Winston   R123 Timpany Street  Brookline          NH0307823416000011234167602078
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Carroll       Alice     L192 Lewis Road      London             NH0341611117000012167890002078
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Hemingway     Joe       E10 Cuba Street      Westminster        MA0147312341000013876900002078
Cooper        Frank     J300 Mohican Avenue  Mohawk             MA0148034167000014341678002078
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Dickens       Arnold    C100 Bleak Street    Gardner            MA0144090000000011123416702078
Thoreaux      Ralph     H800 Emerson Street  Walden             MA0141641678000016000060002078
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
Williams      Samuel    T310 England Road    Worcester          MA0140012341000017789000002078
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Ames          Alice     J40 Center Road      Nashua             NH0306078900000007100000002078
Dickinson     Rose      E21 Depot Road       Amherst            MA0142341678000019666889002078
Frost         Alfred    R123 Amherst Street  Merrimack          NH0306012341000020111149002078
Alexander     Greg      T317 Narrows Road    Westminster        MA0147334160000004100007102068
Abbott        John      B12 Pleasant Street  Nashua             NH0310212340000001100009007018
```

16.9.2 REPORT1—Detail Report Program

REPORT1 uses the PAGE HEADING, DETAIL, and CONTROL FOOTING report groups and produces a detail report—CUSTMAST1.LIS.

To get CUSTMAST1.LIS, you use the following commands:

```
$ COBOL REPORT1

$ LINK REPORT1

$ RUN REPORT1

$ PRINT/NOFEED CUSTMAST1.LIS
```

The program (REPORT1) in Example 16–5 produces the output shown in Figure 16–15 (CUSTMAST1.LIS).

Example 16–5: Sample Program 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOMER-FILE ASSIGN TO "SYS$DISK:".
        SELECT SORT-FILE     ASSIGN TO "SYS$DISK:".
        SELECT SORTED-FILE   ASSIGN TO "SYS$DISK:".
        SELECT PRINTER-FILE  ASSIGN TO "SYS$DISK:".
DATA DIVISION.
FILE SECTION.
SD      SORT-FILE.
01      SORTED-CUSTOMER-MASTER-FILE.
        02  SORT-NAME                   PIC X(26).
        02                              PIC X(71).
FD      CUSTOMER-FILE
        VALUE OF ID IS "CUSTMAST.DAT".
01      CUSTOMER-MASTER-FILE            PIC X(97).
FD      SORTED-FILE
        VALUE OF ID IS "SORTEDMAS.DAT".

01      CUSTOMER-MASTER-FILE.
        02  NAME.
                03  LAST-NAME           PIC X(15).
                03  FIRST-NAME          PIC X(10).
                03  MIDDLE-INIT         PIC X.
        02  ADDRESS                     PIC X(20).
        02  CITY                        PIC X(20).
        02  STATE                       PIC XX.
        02  ZIP                         PIC 99999.
        02  SALESMAN-NUMBER             PIC 99999.
        02  INVOICE-DATA.
                03  INVOICE-NUMBER      PIC 999999.
                03  INVOICE-SALES       PIC S9(5)V99.
                03  INVOICE-DATE.
                        04  INV-DAY     PIC 99.
                        04  INV-MO      PIC 99.
                        04  INV-YR      PIC 99.

FD      PRINTER-FILE
        VALUE OF ID IS "CUSTMAST1.LIS"
        REPORT IS MASTER-LIST.

WORKING-STORAGE SECTION.

01      UNEDITED-DATE.
        02  UE-YEAR     PIC 99.
        02  UE-MONTH    PIC 99.
        02  UE-DAY      PIC 99.
        02  FILLER      PIC X(6).

01      ONE-COUNT       PIC 9 VALUE 1.
```

(continued on next page)

Example 16–5 (Cont.): Sample Program 1

```
REPORT SECTION.

RD      MASTER-LIST
        PAGE LIMIT IS 66
        HEADING      1
        FIRST DETAIL 13
        LAST DETAIL  55
        CONTROL IS FINAL.
01      TYPE IS PAGE HEADING.
        02      LINE  5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 100
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 105
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
        02      LINE  7.
                03      COLUMN  1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
        02      LINE  8.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 10
                        PIC X(4) VALUE "NAME".
                03      COLUMN 29
                        PIC X VALUE "|".
                03      COLUMN  43
                        PIC X(7) VALUE "ADDRESS".
                03      COLUMN  81
                        PIC X VALUE "|".
                03      COLUMN  91
                        PIC X(7) VALUE "INVOICE".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  9.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE 10.
                03      COLUMN  1
                        PIC X(6) VALUE "| LAST".
                03      COLUMN  16
                        PIC X(7) VALUE "| FIRST".
                03      COLUMN  26
                        PIC X(4) VALUE "|MI|".
                03      COLUMN  35
                        PIC X(6) VALUE "STREET".
                03      COLUMN  48
                        PIC X VALUE "|".
                03      COLUMN 52
                        PIC X(4) VALUE "CITY".
```

(continued on next page)

Example 16–5 (Cont.): Sample Program 1

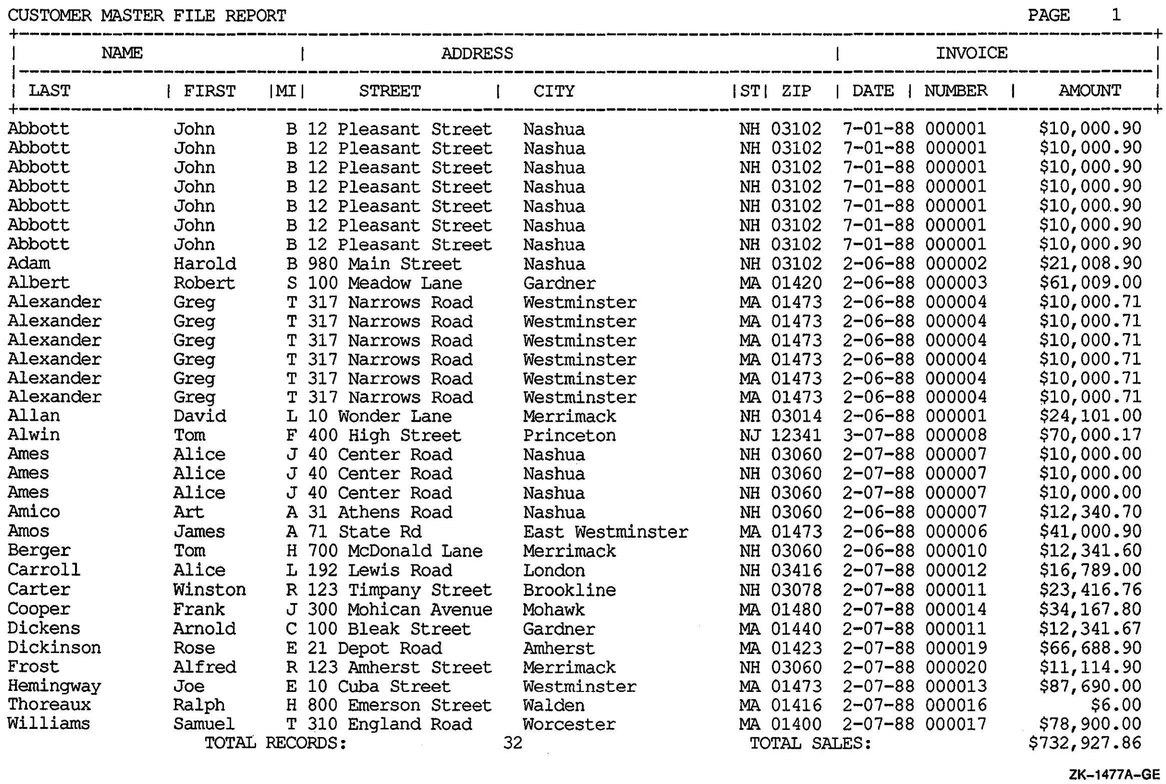
```
                03      COLUMN 71
                        PIC X VALUE "|".
                03      COLUMN 72
                        PIC X(2) VALUE "ST".
                03      COLUMN 74
                        PIC X VALUE "|".
                03      COLUMN 76
                03      COLUMN 81
                        PIC X VALUE "|".
                03      COLUMN 83
                        PIC X(4) VALUE "DATE".
                03      COLUMN 88
                        PIC X VALUE "|".
                03      COLUMN 90
                        PIC X(6) VALUE "NUMBER".
                03      COLUMN 98
                        PIC X VALUE "|".
                03      COLUMN 103
                        PIC X(6) VALUE "AMOUNT".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  11.
                03      COLUMN 1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 1.

        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT.
        02 COLUMN 30    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 51    PIC X(20) SOURCE CITY.
        02 COLUMN 72    PIC XX    SOURCE STATE.
        02 COLUMN 75    PIC 99999 SOURCE ZIP.
        02 COLUMN 81    PIC Z9    SOURCE INV-DAY.
        02 COLUMN 83    PIC X     VALUE "-".
        02 COLUMN 84    PIC 99    SOURCE INV-MO.
        02 COLUMN 86    PIC X     VALUE "-".
        02 COLUMN 87    PIC 99    SOURCE INV-YR.
        02 COLUMN 90    PIC 9(6)  SOURCE INVOICE-NUMBER.
        02 COLUMN 97    PIC $$$,$$$,$$$.99-
                                  SOURCE INVOICE-SALES.
        02 DETAIL-COUNT PIC S9(10) SOURCE ONE-COUNT.
        02 INV-AMOUNT   PIC S9(9)V99 SOURCE INVOICE-SALES.
01      FINAL-FOOTING TYPE IS CONTROL FOOTING FINAL
                      LINE PLUS 5
                      NEXT GROUP NEXT PAGE.
        02      COLUMN  20 PIC X(17) VALUE "TOTAL RECORDS: ".
        02 FDC  COLUMN  40 PIC ZZZ,ZZZ,ZZ9 SUM ONE-COUNT.
        02      COLUMN  73 PIC X(15) VALUE "TOTAL SALES: ".
        02 FIA  COLUMN  93 PIC $$$,$$$,$$$,$$$.99- SUM INVOICE-SALES.
```

(continued on next page)

Example 16–5 (Cont.): Sample Program 1

```
PROCEDURE DIVISION.
000-DO-SORT.
        SORT SORT-FILE ON ASCENDING KEY SORT-NAME
             WITH DUPLICATES IN ORDER
             USING  CUSTOMER-FILE
             GIVING SORTED-FILE.
        DISPLAY "END OF SORT".

050-START.
        OPEN INPUT  SORTED-FILE.
        OPEN OUTPUT PRINTER-FILE.
        ACCEPT UNEDITED-DATE FROM DATE.
        INITIATE MASTER-LIST.
        PERFORM 200-READ-MASTER UNTIL NAME = HIGH-VALUES.
100-END-OF-FILE.
        TERMINATE MASTER-LIST.
        CLOSE SORTED-FILE, PRINTER-FILE.
        STOP RUN.
200-READ-MASTER.
        READ SORTED-FILE AT END MOVE HIGH-VALUES TO NAME.
        IF NAME NOT = HIGH-VALUES GENERATE DETAIL-LINE.
```

Figure 16–15: CUSTMAST1.LIS



```
CUSTOMER MASTER FILE REPORT                                                                         PAGE    1
+---------------------------------------------------------------------------------------------------------------+
|        NAME                |            ADDRESS                                |        INVOICE               |
|---------------------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|    STREET        |  CITY                |ST| ZIP  | DATE | NUMBER  |    AMOUNT   |
+---------------------------------------------------------------------------------------------------------------+
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $10,000.90
Adam            Harold     B 980 Main Street      Nashua               NH 03102  2-06-88 000002      $21,008.90
Albert          Robert     S 100 Meadow Lane      Gardner              MA 01420  2-06-88 000003      $61,009.00
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $10,000.71
Allan           David      L 10 Wonder Lane       Merrimack            NH 03014  2-06-88 000001      $24,101.00
Alwin           Tom        F 400 High Street      Princeton            NJ 12341  3-07-88 000008      $70,000.17
Ames            Alice      J 40 Center Road       Nashua               NH 03060  2-07-88 000007      $10,000.00
Ames            Alice      J 40 Center Road       Nashua               NH 03060  2-07-88 000007      $10,000.00
Ames            Alice      J 40 Center Road       Nashua               NH 03060  2-07-88 000007      $10,000.00
Amico           Art        A 31 Athens Road       Nashua               NH 03060  2-06-88 000007      $12,340.70
Amos            James      A 71 State Rd          East Westminster     MA 01473  2-06-88 000006      $41,000.90
Berger          Tom        H 700 McDonald Lane    Merrimack            NH 03060  2-06-88 000010      $12,341.60
Carroll         Alice      L 192 Lewis Road       London               NH 03416  2-07-88 000012      $16,789.00
Carter          Winston    R 123 Timpany Street   Brookline            NH 03078  2-07-88 000011      $23,416.76
Cooper          Frank      J 300 Mohican Avenue   Mohawk               MA 01480  2-07-88 000014      $34,167.80
Dickens         Arnold     C 100 Bleak Street     Gardner              MA 01440  2-07-88 000011      $12,341.67
Dickinson       Rose       E 21 Depot Road        Amherst              MA 01423  2-07-88 000019      $66,688.90
Frost           Alfred     R 123 Amherst Street   Merrimack            NH 03060  2-07-88 000020      $11,114.90
Hemingway       Joe        E 10 Cuba Street       Westminster          MA 01473  2-07-88 000013      $87,690.00
Thoreaux        Ralph      H 800 Emerson Street   Walden               MA 01416  2-07-88 000016           $6.00
Williams        Samuel     T 310 England Road     Worcester            MA 01400  2-07-88 000017      $78,900.00
                   TOTAL RECORDS:               32                      TOTAL SALES:                $732,927.86
```

ZK-1477A-GE

16.9.3 REPORT2—Detail Report Program

Example 16–6 (REPORT2) is a Report Writer program that uses the REPORT HEADING, PAGE HEADING, DETAIL, CONTROL FOOTING, and REPORT FOOTING report groups and produces a detail report—CUSTMAST2.LIS (shown in Figure 16–16). The output includes both subtotals and rolling-forward totals.

Example 16–6: Sample Program 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOMER-FILE ASSIGN TO "SYS$DISK:".
        SELECT SORT-FILE     ASSIGN TO "SYS$DISK:".
        SELECT SORTED-FILE   ASSIGN TO "SYS$DISK:".
        SELECT PRINTER-FILE  ASSIGN TO "SYS$DISK:".
DATA DIVISION.
FILE SECTION.

SD      SORT-FILE.
01      SORTED-CUSTOMER-MASTER-FILE.
        02  SORT-NAME                   PIC X(26).
        02                              PIC X(71).

FD      CUSTOMER-FILE
        VALUE OF ID IS "CUSTMAST.DAT".
01      CUSTOMER-MASTER-FILE            PIC X(97).

FD      SORTED-FILE
        VALUE OF ID IS "SORTEDMAS.DAT".
01      CUSTOMER-MASTER-FILE.
        02  NAME.
                03  LAST-NAME           PIC X(15).
                03  FIRST-NAME          PIC X(10).
                03  MIDDLE-INIT         PIC X.
        02  ADDRESS                     PIC X(20).
        02  CITY                        PIC X(20).
        02  STATE                       PIC XX.
        02  ZIP                         PIC 99999.
        02  SALESMAN-NUMBER             PIC 99999.
        02  INVOICE-DATA.
               03  INVOICE-NUMBER       PIC 999999.
               03  INVOICE-SALES        PIC S9(5)V99.
               03  INVOICE-DATE.
                      04  INV-DAY       PIC 99.
                      04  INV-MO        PIC 99.
                      04  INV-YR        PIC 99.
FD      PRINTER-FILE
        VALUE OF ID IS "CUSTMAST2.LIS"
        REPORT IS MASTER-LIST.
WORKING-STORAGE SECTION.
01      UNEDITED-DATE.
        02  UE-YEAR     PIC 99.
        02  UE-MONTH    PIC 99.
        02  UE-DAY      PIC 99.
        02  FILLER      PIC X(6).

01      ONE-COUNT       PIC 9 VALUE 1.
```

(continued on next page)

Example 16–6 (Cont.): Sample Program 2

```
REPORT SECTION.
RD      MASTER-LIST
        PAGE LIMIT IS 66
        HEADING       1
        FIRST DETAIL  13
        LAST DETAIL   55
        CONTROLS ARE FINAL
                      NAME.
01      REPORT-HEADER TYPE IS REPORT HEADING NEXT GROUP NEXT PAGE.
        02      LINE 24.
                03      COLUMN 45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN 45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN 57
                        PIC X   VALUE "-".
                03      COLUMN 58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN 60
                        PIC X   VALUE "-".
                03      COLUMN 61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN 75
                        PIC X   VALUE "*".
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30.
                03      COLUMN  45
                        PIC X(31) VALUE "*         Report 2            *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X(31) VALUE "*       Detail Report         *".
        02      LINE 32.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
```

(continued on next page)

Example 16–6 (Cont.): Sample Program 2

```
        02      LINE 33.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 34.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 35.
                03      COLUMN  45
                        PIC X(31) VALUE ALL "*".
01      TYPE IS PAGE HEADING.
        02      LINE  5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 100
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 105
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
        02      LINE  7.
                03      COLUMN  1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
        02      LINE  8.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 10
                        PIC X(4) VALUE "NAME".
                03      COLUMN  29
                        PIC X VALUE "|".
                03      COLUMN 43
                        PIC X(7) VALUE "ADDRESS".
                03      COLUMN  81
                        PIC X VALUE "|".
                03      COLUMN  91
                        PIC X(7) VALUE "INVOICE".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  9.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "|".
```

(continued on next page)

Example 16–6 (Cont.): Sample Program 2

```
        02      LINE  10.
                03      COLUMN  1
                        PIC X(6) VALUE "| LAST".
                03      COLUMN 16
                        PIC X(7) VALUE "| FIRST".
                03      COLUMN 26
                        PIC X(4) VALUE "|MI|".
                03      COLUMN  35
                        PIC X(6) VALUE "STREET".
                03      COLUMN 48
                        PIC X VALUE "|".
                03      COLUMN 52
                        PIC X(4) VALUE "CITY".
                03      COLUMN  71
                        PIC X VALUE "|".
                03      COLUMN 72
                        PIC X(2) VALUE "ST".
                03      COLUMN 74
                        PIC X VALUE "|".
                03      COLUMN 76
                        PIC X(3) VALUE "ZIP".
                03      COLUMN 81
                        PIC X VALUE "|".
                03      COLUMN 83
                        PIC X(4) VALUE "DATE".
                03      COLUMN 88
                        PIC X VALUE "|".
                03      COLUMN  90
                        PIC X(6) VALUE "NUMBER".
                03      COLUMN 98
                        PIC X VALUE "|".
                03      COLUMN 103
                        PIC X(6) VALUE "AMOUNT".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  11.
                03      COLUMN 1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 2.
        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT.
        02 COLUMN 30    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 51    PIC X(20) SOURCE CITY.
        02 COLUMN 72    PIC XX    SOURCE STATE.
        02 COLUMN 75    PIC 99999 SOURCE ZIP.
        02 COLUMN 81    PIC Z9    SOURCE INV-DAY.
        02 COLUMN 83    PIC X     VALUE "-".
        02 COLUMN 84    PIC 99    SOURCE INV-MO.
        02 COLUMN 86    PIC X     VALUE "-".
        02 COLUMN 87    PIC 99    SOURCE INV-YR.
        02 COLUMN 90    PIC 9(6)  SOURCE INVOICE-NUMBER.
        02 COLUMN 97    PIC $$$,$$$,$$$.99-
                                  SOURCE INVOICE-SALES.
        02 DETAIL-COUNT PIC S9(10) SOURCE ONE-COUNT.
        02 INV-AMOUNT   PIC S9(9)V99 SOURCE INVOICE-SALES.
```

(continued on next page)

Example 16–6 (Cont.): Sample Program 2

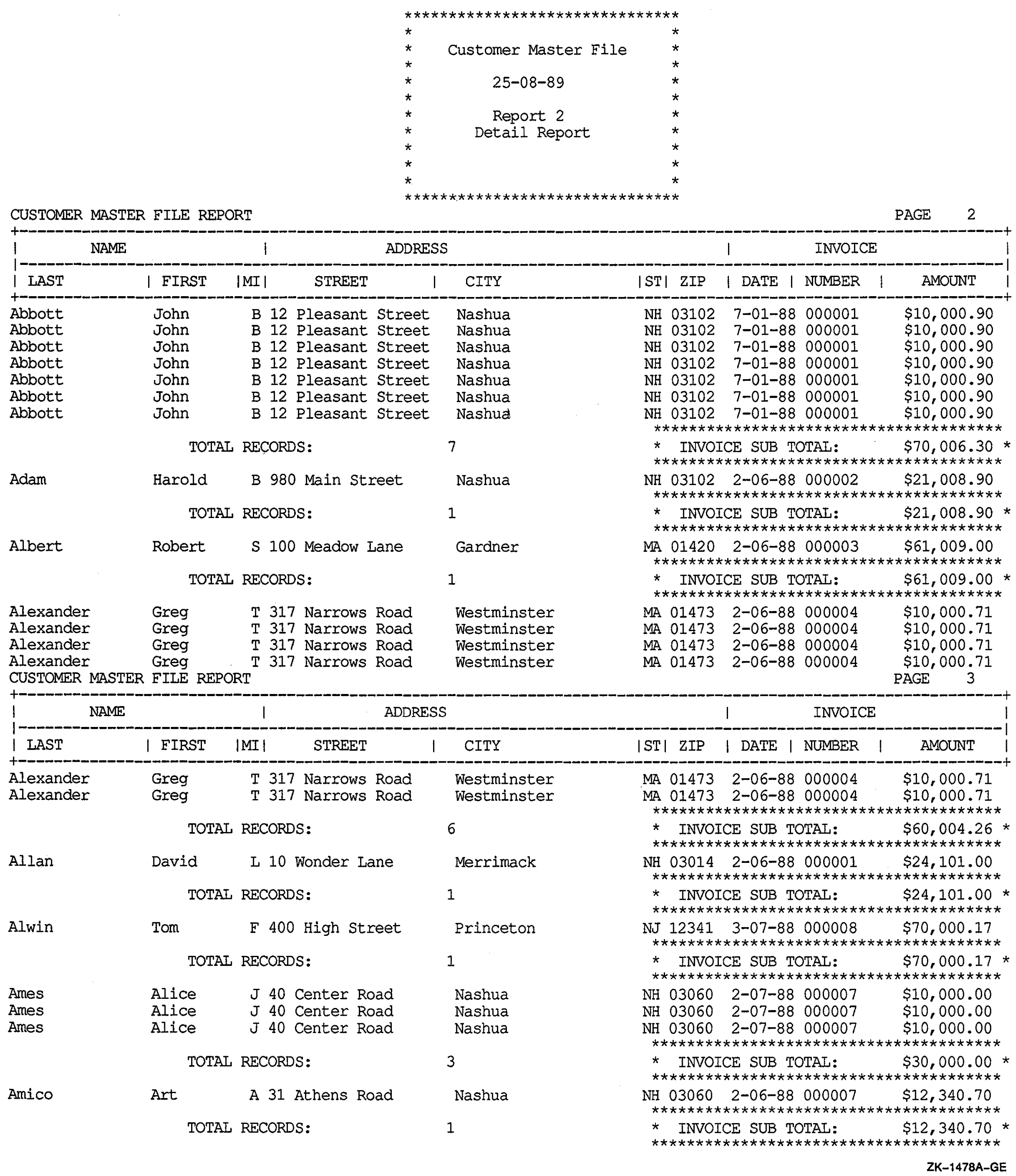
```
01      TYPE IS CONTROL FOOTING NAME
                NEXT GROUP IS PLUS 2.
        02      LINE IS PLUS 2.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  20  PIC X(17) VALUE " TOTAL RECORDS: ".
                03 IDC  COLUMN  40  PIC ZZZ,ZZZ,ZZ9 SUM ONE-COUNT.
                03      COLUMN  73  PIC X(22) VALUE "*  INVOICE SUB TOTAL: ".
                03 IIA  COLUMN  97  PIC $$$,$$$,$$$.99- SUM INVOICE-SALES.
                03      COLUMN  112 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".
01      FINAL-FOOTING TYPE IS CONTROL FOOTING FINAL
                      NEXT GROUP NEXT PAGE.
        02      LINE IS PLUS 2.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  14 PIC X(21) VALUE "GRAND TOTAL RECORDS: ".
                03 FDC  COLUMN  40 PIC ZZZ,ZZZ,ZZ9 SUM IDC.
                03      COLUMN  70 PIC X(24) VALUE "*  GRAND TOTAL INVOICES:".
                03 FIA  COLUMN  95 PIC $,$$$,$$$,$$$.99- SUM IIA.
                03      COLUMN  112 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".
01      REPORT-FOOTER TYPE IS REPORT FOOTING.
        02      LINE 24  ON NEXT PAGE COLUMN  45
                         PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN  45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN  55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN  57
                        PIC X   VALUE "-".
                03      COLUMN  58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN  60
                        PIC X   VALUE "-".
                03      COLUMN  61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN  75
                        PIC X VALUE "*".
```

(continued on next page)

Example 16–6 (Cont.): Sample Program 2

```
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30 COLUMN  45
                        PIC X(31) VALUE "*       End of Report 2       *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 32 COLUMN  45
                        PIC X(31) VALUE ALL "*".
PROCEDURE DIVISION.
DECLARATIVES.
BOR SECTION.
        USE BEFORE REPORTING REPORT-HEADER.
BOR-A.
        DISPLAY "BEGINNING-OF-REPORT".
EOR SECTION.
        USE BEFORE REPORTING REPORT-FOOTER.
EOR-A.
        DISPLAY "END-OF-REPORT".
END DECLARATIVES.
MAIN SECTION.
000-DO-SORT.
        SORT SORT-FILE ON ASCENDING KEY SORT-NAME
           WITH DUPLICATES IN ORDER
           USING CUSTOMER-FILE
           GIVING SORTED-FILE.
000-START.
        OPEN INPUT  SORTED-FILE.
        OPEN OUTPUT PRINTER-FILE.
        ACCEPT UNEDITED-DATE FROM DATE.
        INITIATE MASTER-LIST.
        PERFORM 200-READ-MASTER UNTIL NAME = HIGH-VALUES.
100-END-OF-FILE.
        TERMINATE MASTER-LIST.
        CLOSE SORTED-FILE, PRINTER-FILE.
        STOP RUN.
200-READ-MASTER.
        READ SORTED-FILE AT END MOVE HIGH-VALUES TO NAME.
        IF NAME NOT = HIGH-VALUES GENERATE DETAIL-LINE.
```

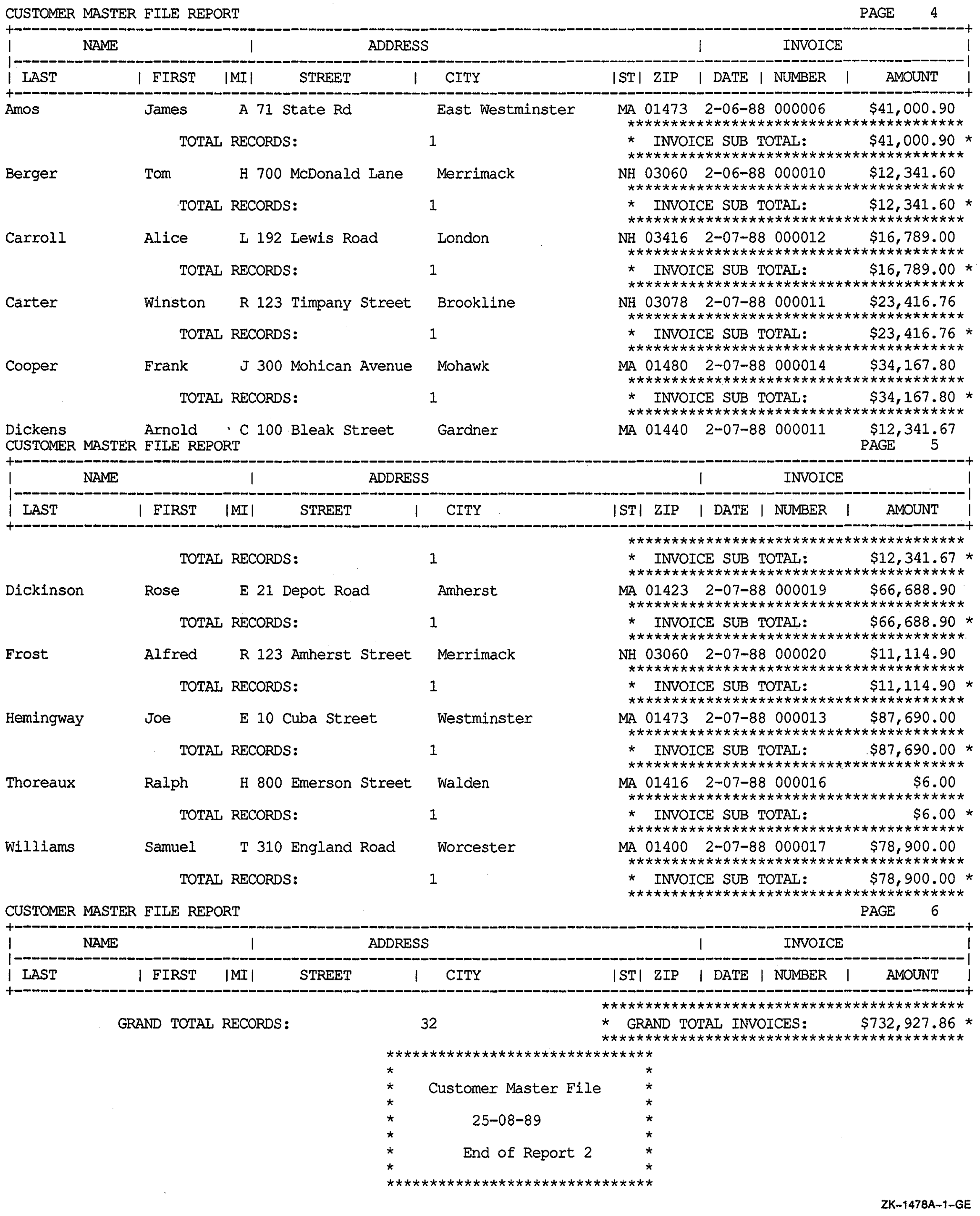
Figure 16–16: CUSTMAST2.LIS



```
                                        ******************************
                                        *                            *
                                        *    Customer Master File    *
                                        *                            *
                                        *          25-08-89          *
                                        *                            *
                                        *          Report 2          *
                                        *        Detail Report       *
                                        *                            *
                                        *                            *
                                        *                            *
                                        ******************************
CUSTOMER MASTER FILE REPORT                                                                    PAGE    2
+------------------------------------------------------------------------------------------------------+
|         NAME              |                ADDRESS                          |          INVOICE        |
|------------------------------------------------------------------------------------------------------|
| LAST           | FIRST   |MI|     STREET        |  CITY            |ST| ZIP  | DATE | NUMBER  |   AMOUNT   |
+------------------------------------------------------------------------------------------------------+
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
Abbott          John       B 12 Pleasant Street   Nashua              NH 03102  7-01-88 000001     $10,000.90
                                                                       **************************************
                    TOTAL RECORDS:          7                          *  INVOICE SUB TOTAL:      $70,006.30 *
                                                                       **************************************
Adam            Harold     B 980 Main Street      Nashua              NH 03102  2-06-88 000002     $21,008.90
                                                                       **************************************
                    TOTAL RECORDS:          1                          *  INVOICE SUB TOTAL:      $21,008.90 *
                                                                       **************************************
Albert          Robert     S 100 Meadow Lane      Gardner             MA 01420  2-06-88 000003     $61,009.00
                                                                       **************************************
                    TOTAL RECORDS:          1                          *  INVOICE SUB TOTAL:      $61,009.00 *
                                                                       **************************************
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
CUSTOMER MASTER FILE REPORT                                                                    PAGE    3
+------------------------------------------------------------------------------------------------------+
|         NAME              |                ADDRESS                          |          INVOICE        |
|------------------------------------------------------------------------------------------------------|
| LAST           | FIRST   |MI|     STREET        |  CITY            |ST| ZIP  | DATE | NUMBER  |   AMOUNT   |
+------------------------------------------------------------------------------------------------------+
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
Alexander       Greg       T 317 Narrows Road     Westminster         MA 01473  2-06-88 000004     $10,000.71
                                                                       **************************************
                    TOTAL RECORDS:          6                          *  INVOICE SUB TOTAL:      $60,004.26 *
                                                                       **************************************
Allan           David      L 10 Wonder Lane       Merrimack           NH 03014  2-06-88 000001     $24,101.00
                                                                       **************************************
                    TOTAL RECORDS:          1                          *  INVOICE SUB TOTAL:      $24,101.00 *
                                                                       **************************************
Alwin           Tom        F 400 High Street      Princeton           NJ 12341  3-07-88 000008     $70,000.17
                                                                       **************************************
                    TOTAL RECORDS:          1                          *  INVOICE SUB TOTAL:      $70,000.17 *
                                                                       **************************************
Ames            Alice      J 40 Center Road       Nashua              NH 03060  2-07-88 000007     $10,000.00
Ames            Alice      J 40 Center Road       Nashua              NH 03060  2-07-88 000007     $10,000.00
Ames            Alice      J 40 Center Road       Nashua              NH 03060  2-07-88 000007     $10,000.00
                                                                       **************************************
                    TOTAL RECORDS:          3                          *  INVOICE SUB TOTAL:      $30,000.00 *
                                                                       **************************************
Amico           Art        A 31 Athens Road       Nashua              NH 03060  2-06-88 000007     $12,340.70
                                                                       **************************************
                    TOTAL RECORDS:          1                          *  INVOICE SUB TOTAL:      $12,340.70 *
                                                                       **************************************
```

ZK-1478A-GE

(continued on next page)



```
CUSTOMER MASTER FILE REPORT                                                                 PAGE    4
+-----------------------------------------------------------------------------------------------------+
|        NAME                |                ADDRESS                         |        INVOICE        |
|-----------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|     STREET      |  CITY                |ST| ZIP  | DATE | NUMBER |   AMOUNT |
+-----------------------------------------------------------------------------------------------------+
Amos            James     A 71 State Rd        East Westminster     MA 01473  2-06-88 000006     $41,000.90
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $41,000.90 *
                                                                      ***************************************
Berger          Tom       H 700 McDonald Lane  Merrimack            NH 03060  2-06-88 000010     $12,341.60
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $12,341.60 *
                                                                      ***************************************
Carroll         Alice     L 192 Lewis Road     London               NH 03416  2-07-88 000012     $16,789.00
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $16,789.00 *
                                                                      ***************************************
Carter          Winston   R 123 Timpany Street Brookline            NH 03078  2-07-88 000011     $23,416.76
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $23,416.76 *
                                                                      ***************************************
Cooper          Frank     J 300 Mohican Avenue Mohawk               MA 01480  2-07-88 000014     $34,167.80
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $34,167.80 *
                                                                      ***************************************
Dickens         Arnold    C 100 Bleak Street   Gardner              MA 01440  2-07-88 000011     $12,341.67
CUSTOMER MASTER FILE REPORT                                                                 PAGE    5
+-----------------------------------------------------------------------------------------------------+
|        NAME                |                ADDRESS                         |        INVOICE        |
|-----------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|     STREET      |  CITY                |ST| ZIP  | DATE | NUMBER |   AMOUNT |
+-----------------------------------------------------------------------------------------------------+
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $12,341.67 *
                                                                      ***************************************
Dickinson       Rose      E 21 Depot Road      Amherst              MA 01423  2-07-88 000019     $66,688.90
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $66,688.90 *
                                                                      ***************************************
Frost           Alfred    R 123 Amherst Street Merrimack            NH 03060  2-07-88 000020     $11,114.90
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $11,114.90 *
                                                                      ***************************************
Hemingway       Joe       E 10 Cuba Street     Westminster          MA 01473  2-07-88 000013     $87,690.00
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $87,690.00 *
                                                                      ***************************************
Thoreaux        Ralph     H 800 Emerson Street Walden               MA 01416  2-07-88 000016          $6.00
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:           $6.00 *
                                                                      ***************************************
Williams        Samuel    T 310 England Road   Worcester            MA 01400  2-07-88 000017     $78,900.00
                                                                      ***************************************
                   TOTAL RECORDS:              1                      *  INVOICE SUB TOTAL:      $78,900.00 *
                                                                      ***************************************
CUSTOMER MASTER FILE REPORT                                                                 PAGE    6
+-----------------------------------------------------------------------------------------------------+
|        NAME                |                ADDRESS                         |        INVOICE        |
|-----------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|     STREET      |  CITY                |ST| ZIP  | DATE | NUMBER |   AMOUNT |
+-----------------------------------------------------------------------------------------------------+
                                                                   ******************************************
             GRAND TOTAL RECORDS:            32                    *  GRAND TOTAL INVOICES:     $732,927.86 *
                                                                   ******************************************
                                            ******************************
                                            *                            *
                                            *    Customer Master File    *
                                            *                            *
                                            *         25-08-89           *
                                            *                            *
                                            *      End of Report 2       *
                                            *                            *
                                            ******************************
```

ZK-1478A-1-GE

16.9.4 REPORT3—Detail Report Program

Example 16–7 (REPORT3) is a Report Writer program that uses the REPORT HEADING, PAGE HEADING, DETAIL, CONTROL FOOTING, and REPORT FOOTING report groups and produces a detail report—CUSTMAST3.LIS (shown in Figure 16–17).

Example 16–7: Sample Program 3

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT3.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOMER-FILE ASSIGN TO "SYS$DISK:".
        SELECT SORT-FILE     ASSIGN TO "SYS$DISK:".
        SELECT SORTED-FILE   ASSIGN TO "SYS$DISK:".
        SELECT PRINTER-FILE  ASSIGN TO "SYS$DISK:".

DATA DIVISION.
FILE SECTION.
SD      SORT-FILE.
01      SORTED-CUSTOMER-MASTER-FILE.
        02  SORT-NAME                   PIC X(26).
        02                              PIC X(71).

FD      CUSTOMER-FILE
        VALUE OF ID IS "CUSTMAST.DAT".
01      CUSTOMER-MASTER-FILE            PIC X(97).

FD      SORTED-FILE
        VALUE OF ID IS "SORTEDMAS.DAT".
01      SORTED-RECORD.
        02  SORTED-NAME                 PIC X(26).
        02  S-ADDRESS                   PIC X(20).
        02  S-CITY                      PIC X(20).
        02  S-STATE                     PIC XX.
        02  S-ZIP                       PIC 99999.
        02  S-SALESMAN-NUMBER           PIC 99999.
        02  S-INVOICE-DATA.
                03  S-INVOICE-NUMBER    PIC 999999.
                03  S-INVOICE-SALES     PIC S9(5)V99.
                03  S-INVOICE-DATE.
                        04  S-INV-DAY   PIC 99.
                        04  S-INV-MO    PIC 99.
                        04  S-INV-YR    PIC 99.

FD      PRINTER-FILE
        VALUE OF ID IS "CUSTMAST3.LIS"
        REPORT IS MASTER-LIST.

WORKING-STORAGE SECTION.

01      UNEDITED-DATE.
        02  UE-YEAR     PIC 99.
        02  UE-MONTH    PIC 99.
        02  UE-DAY      PIC 99.
        02  FILLER      PIC X(6).
01      ONE-COUNT                       PIC 9 VALUE 1.
01      EOF                             PIC X VALUE "N".
01      SAVE-INVOICE-SALES              PIC S9(9)V99 VALUE 0.
```

(continued on next page)

```
01      CUSTOMER-MASTER-RECORD.
        02  NAME.
            03  LAST-NAME         PIC X(15).
            03  FIRST-NAME        PIC X(10).
            03  MIDDLE-INIT       PIC X.
        02  ADDRESS               PIC X(20).
        02  CITY                  PIC X(20).
        02  STATE                 PIC XX.
        02  ZIP                   PIC 99999.
        02  SALESMAN-NUMBER       PIC 99999.
        02  INVOICE-DATA.
            03  INVOICE-NUMBER    PIC 999999.
            03  INVOICE-SALES     PIC S9(5)V99.
            03  INVOICE-DATE.
                04  INV-DAY       PIC 99.
                04  INV-MO        PIC 99.
                04  INV-YR        PIC 99.

REPORT SECTION.

RD      MASTER-LIST
        PAGE LIMIT IS 66
        HEADING       1
        FIRST DETAIL  13
        LAST DETAIL   55
        CONTROLS ARE FINAL.
01      REPORT-HEADER TYPE IS REPORT HEADING NEXT GROUP NEXT PAGE.
        02      LINE 24.
                03      COLUMN 45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN 45
                        PIC X(31) VALUE "*    Customer Master File     *".

        02      LINE 27.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN 57
                        PIC X   VALUE "-".
                03      COLUMN 58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN 60
                        PIC X   VALUE "-".
                03      COLUMN 61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN 75
                        PIC X VALUE "*".
```

(continued on next page)

Example 16–7 (Cont.): Sample Program 3

```
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30.
                03      COLUMN  45
                        PIC X(31) VALUE "*          Report 3           *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X(31) VALUE "*       Detail  Report        *".
        02      LINE 32.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 33.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 34.
                03      COLUMN  45
                        PIC X(31) VALUE ALL "*".

01        TYPE IS PAGE HEADING.
        02      LINE  5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 100
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 105
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
        02      LINE  7.
                03      COLUMN  1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
        02      LINE  8.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 10
                        PIC X(4) VALUE "NAME".
                03      COLUMN  29
                        PIC X VALUE "|".
                03      COLUMN 43
                        PIC X(7) VALUE "ADDRESS".
                03      COLUMN  81
                        PIC X VALUE "|".
                03      COLUMN  91
                        PIC X(7) VALUE "INVOICE".
                03      COLUMN 112
                        PIC X VALUE "|".
```

(continued on next page)

Example 16–7 (Cont.): Sample Program 3

```
        02      LINE  9.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  10.
                03      COLUMN  1
                        PIC X(6) VALUE "| LAST".
                03      COLUMN 16
                        PIC X(7) VALUE "| FIRST".
                03      COLUMN 26
                        PIC X(4) VALUE "|MI|".
                03      COLUMN  35
                        PIC X(6) VALUE "STREET".
                03      COLUMN 48
                        PIC X VALUE "|".
                03      COLUMN 52
                        PIC X(4) VALUE "CITY".
                03      COLUMN  71
                        PIC X VALUE "|".
                03      COLUMN 72
                        PIC X(2) VALUE "ST".
                03      COLUMN 74
                        PIC X VALUE "|".
                03      COLUMN 76
                        PIC X(3) VALUE "ZIP".
                03      COLUMN 81
                        PIC X VALUE "|".
                03      COLUMN 83
                        PIC X(4) VALUE "DATE".
                03      COLUMN 88
                        PIC X VALUE "|".
                03      COLUMN  90
                        PIC X(6) VALUE "NUMBER".
                03      COLUMN 98
                        PIC X VALUE "|".
                03      COLUMN 103
                        PIC X(6) VALUE "AMOUNT".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  11.
                03      COLUMN 1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
01      DETAIL-LINE
        TYPE DETAIL LINE IS PLUS 1.
        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT.
        02 COLUMN 30    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 51    PIC X(20) SOURCE CITY.
        02 COLUMN 72    PIC XX    SOURCE STATE.
        02 COLUMN 75    PIC 99999 SOURCE ZIP.
        02 COLUMN 81    PIC Z9    SOURCE INV-DAY.
```

(continued on next page)

```
        02 COLUMN 83    PIC X     VALUE "-".
        02 COLUMN 84    PIC 99    SOURCE INV-MO.
        02 COLUMN 86    PIC X     VALUE "-".
        02 COLUMN 87    PIC 99    SOURCE INV-YR.
        02 COLUMN 90    PIC 9(6)  SOURCE INVOICE-NUMBER.
        02 COLUMN 97    PIC $$$,$$$,$$$.99-
                          SOURCE SAVE-INVOICE-SALES.
01      FINAL-FOOTING TYPE IS CONTROL FOOTING FINAL
                      NEXT GROUP NEXT PAGE.
        02      LINE IS PLUS 2.
                03      COLUMN  70
                        PIC X(33) VALUE "*********************************".
                03      COLUMN  103
                        PIC X(13) VALUE "*************".
        02      LINE IS PLUS 1.
                03      COLUMN  70 PIC X(24) VALUE "*  GRAND TOTAL INVOICES:".
                03 FIA  COLUMN  95 PIC $,$$$,$$$,$$$.99- SUM INVOICE-SALES.
                03      COLUMN  114 PIC XXX VALUE " * ".
        02      LINE IS PLUS 1.
                03      COLUMN  70
                        PIC X(33) VALUE "*********************************".
                03      COLUMN  103
                        PIC X(13) VALUE "*************".

01      REPORT-FOOTER TYPE IS REPORT FOOTING.
        02      LINE 24  ON NEXT PAGE COLUMN  45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN  45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28 .
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN  55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN  57
                        PIC X   VALUE "-".
                03      COLUMN  58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN  60
                        PIC X   VALUE "-".
                03      COLUMN  61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN  75
                        PIC X VALUE "*".
```

(continued on next page)

Example 16–7 (Cont.): Sample Program 3

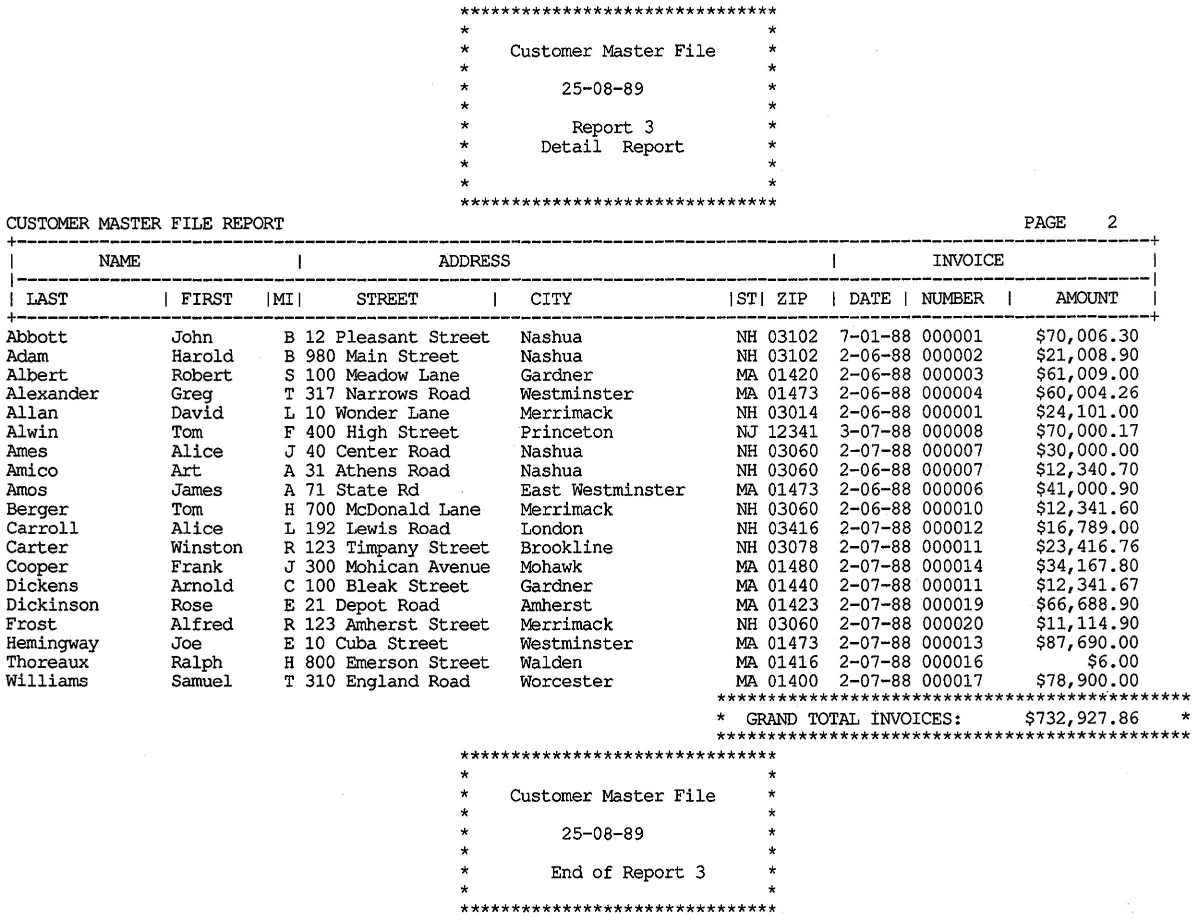
```
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30 COLUMN  45
                        PIC X(31) VALUE "*        End of Report 3     *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 32 COLUMN  45
                        PIC X(31) VALUE ALL "*".

PROCEDURE DIVISION.

DECLARATIVES.
BOR SECTION.
        USE BEFORE REPORTING REPORT-HEADER.
BOR-A.
        DISPLAY "BEGINNING-OF-REPORT".
EOR SECTION.
        USE BEFORE REPORTING REPORT-FOOTER.
EOR-A.
        DISPLAY "END-OF-REPORT".
DET SECTION.
        USE BEFORE REPORTING DETAIL-LINE.
DET-A.
        IF SORTED-NAME = NAME
                MOVE SORTED-RECORD TO CUSTOMER-MASTER-RECORD
                ADD INVOICE-SALES TO SAVE-INVOICE-SALES
                SUPPRESS PRINTING.
        IF NAME = SPACES SUPPRESS PRINTING.
END DECLARATIVES.

MAIN SECTION.
000-DO-SORT.
        SORT SORT-FILE ON ASCENDING KEY SORT-NAME
                WITH DUPLICATES IN ORDER
                USING CUSTOMER-FILE
                GIVING SORTED-FILE.

000-START.
        OPEN INPUT  SORTED-FILE.
        OPEN OUTPUT PRINTER-FILE.
        ACCEPT UNEDITED-DATE FROM DATE.
        MOVE SPACES TO NAME.
        INITIATE MASTER-LIST.
        PERFORM 200-READ-MASTER UNTIL EOF = "Y".

100-END-OF-FILE.
        TERMINATE MASTER-LIST.
        CLOSE SORTED-FILE, PRINTER-FILE.
        STOP RUN.

200-READ-MASTER.
        READ SORTED-FILE AT END MOVE "Y" TO EOF
                                MOVE HIGH-VALUES TO SORTED-NAME.
```

(continued on next page)

Example 16–7 (Cont.): Sample Program 3

```
         GENERATE DETAIL-LINE.
        IF SORTED-NAME NOT = NAME
                           MOVE S-INVOICE-SALES TO SAVE-INVOICE-SALES.

        IF EOF NOT = "Y" MOVE SORTED-RECORD TO CUSTOMER-MASTER-RECORD.
```

Figure 16–17: CUSTMAST3.LIS



```
                                            *******************************
                                            *                             *
                                            *    Customer Master File     *
                                            *                             *
                                            *         25-08-89            *
                                            *                             *
                                            *          Report 3           *
                                            *       Detail  Report        *
                                            *                             *
                                            *                             *
                                            *******************************
CUSTOMER MASTER FILE REPORT                                                                       PAGE     2
+--------------------------------------------------------------------------------------------------------------+
|        NAME                |             ADDRESS                                   |        INVOICE          |
|--------------------------------------------------------------------------------------------------------------|
| LAST          | FIRST    |MI|     STREET       |   CITY                  |ST| ZIP  | DATE | NUMBER  |  AMOUNT  |
+--------------------------------------------------------------------------------------------------------------+
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001      $70,006.30
Adam            Harold     B 980 Main Street      Nashua               NH 03102  2-06-88 000002      $21,008.90
Albert          Robert     S 100 Meadow Lane      Gardner              MA 01420  2-06-88 000003      $61,009.00
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004      $60,004.26
Allan           David      L 10 Wonder Lane       Merrimack            NH 03014  2-06-88 000001      $24,101.00
Alwin           Tom        F 400 High Street      Princeton            NJ 12341  3-07-88 000008      $70,000.17
Ames            Alice      J 40 Center Road       Nashua               NH 03060  2-07-88 000007      $30,000.00
Amico           Art        A 31 Athens Road       Nashua               NH 03060  2-06-88 000007      $12,340.70
Amos            James      A 71 State Rd          East Westminster     MA 01473  2-06-88 000006      $41,000.90
Berger          Tom        H 700 McDonald Lane    Merrimack            NH 03060  2-06-88 000010      $12,341.60
Carroll         Alice      L 192 Lewis Road       London               NH 03416  2-07-88 000012      $16,789.00
Carter          Winston    R 123 Timpany Street   Brookline            NH 03078  2-07-88 000011      $23,416.76
Cooper          Frank      J 300 Mohican Avenue   Mohawk               MA 01480  2-07-88 000014      $34,167.80
Dickens         Arnold     C 100 Bleak Street     Gardner              MA 01440  2-07-88 000011      $12,341.67
Dickinson       Rose       E 21 Depot Road        Amherst              MA 01423  2-07-88 000019      $66,688.90
Frost           Alfred     R 123 Amherst Street   Merrimack            NH 03060  2-07-88 000020      $11,114.90
Hemingway       Joe        E 10 Cuba Street       Westminster          MA 01473  2-07-88 000013      $87,690.00
Thoreaux        Ralph      H 800 Emerson Street   Walden               MA 01416  2-07-88 000016           $6.00
Williams        Samuel     T 310 England Road     Worcester            MA 01400  2-07-88 000017      $78,900.00
                                                                     **********************************************
                                                                     *  GRAND TOTAL INVOICES:      $732,927.86    *
                                                                     **********************************************
                                            *******************************
                                            *                             *
                                            *    Customer Master File     *
                                            *                             *
                                            *         25-08-89            *
                                            *                             *
                                            *       End of Report 3       *
                                            *                             *
                                            *******************************
```

ZK–1479A–GE

16.9.5 REPORT4—Detail Report Program

Example 16–8 (REPORT4) is a Report Writer program that uses the REPORT HEADING, PAGE HEADING, DETAIL, PAGE FOOTING, CONTROL FOOTING, and REPORT FOOTING report groups. The program also uses the TYPE DETAIL clause—GROUP INDICATE. The program produces a detail report—CUSTMAST4.LIS (shown in Figure 16–18).

Example 16–8: Sample Program 4

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT4.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOMER-FILE ASSIGN TO "SYS$DISK:".
        SELECT SORT-FILE     ASSIGN TO "SYS$DISK:".
        SELECT SORTED-FILE   ASSIGN TO "SYS$DISK:".
        SELECT PRINTER-FILE  ASSIGN TO "SYS$DISK:".

DATA DIVISION.
FILE SECTION.
SD      SORT-FILE.
01      SORTED-CUSTOMER-MASTER-FILE.
        02  SORT-NAME                   PIC X(26).
        02                              PIC X(71).

FD      CUSTOMER-FILE
        VALUE OF ID IS "CUSTMAST.DAT".
01      CUSTOMER-MASTER-FILE            PIC X(97).

FD      SORTED-FILE
        VALUE OF ID IS "SORTEDMAS.DAT".
01      CUSTOMER-MASTER-FILE.
        02  NAME.
                03  LAST-NAME           PIC X(15).
                03  FIRST-NAME          PIC X(10).
                03  MIDDLE-INIT         PIC X.
        02  ADDRESS                     PIC X(20).
        02  CITY                        PIC X(20).
        02  STATE                       PIC XX.
        02  ZIP                         PIC 99999.
        02  SALESMAN-NUMBER             PIC 99999.
        02  INVOICE-DATA.
                03  INVOICE-NUMBER      PIC 999999.
                03  INVOICE-SALES       PIC S9(5)V99.
                03  INVOICE-DATE.
                        04  INV-DAY     PIC 99.
                        04  INV-MO      PIC 99.
                        04  INV-YR      PIC 99.

FD      PRINTER-FILE
        VALUE OF ID IS "CUSTMAST4.LIS"
        REPORT IS MASTER-LIST.

WORKING-STORAGE SECTION.
01      UNEDITED-DATE.
        02  UE-YEAR      PIC 99.
        02  UE-MONTH     PIC 99.
        02  UE-DAY       PIC 99.
        02  FILLER       PIC X(6).

01      ONE-COUNT PIC 9 VALUE 1.

REPORT SECTION.
RD MASTER-LIST
        PAGE LIMIT IS 66
        HEADING       1
        FIRST DETAIL  13
        LAST DETAIL   55
        FOOTING       58
        CONTROLS ARE FINAL
                     NAME.
```

(continued on next page)

Example 16–8 (Cont.): Sample Program 4

```
01      REPORT-HEADER TYPE IS REPORT HEADING NEXT GROUP NEXT PAGE.
        02      LINE 24.
                03      COLUMN 45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN 45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN 57
                        PIC X   VALUE "-".
                03      COLUMN 58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN 60
                        PIC X   VALUE "-".
                03      COLUMN 61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30.
                03      COLUMN  45
                        PIC X(31) VALUE "*       GROUP INDICATE        *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X(31) VALUE "*       Detail Report 4       *".
        02      LINE 32.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 33.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 34.
                03      COLUMN  45
                        PIC X(31) VALUE ALL "*".
```

(continued on next page)

```
01      TYPE IS PAGE HEADING.
        02      LINE  5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 100
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 105
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
        02      LINE  7.
                03      COLUMN  1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
        02      LINE  8.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 10
                        PIC X(4) VALUE "NAME".
                03      COLUMN  29
                        PIC X VALUE "|".
                03      COLUMN 43
                        PIC X(7) VALUE "ADDRESS".
                03      COLUMN  81
                        PIC X VALUE "|".
                03      COLUMN  91
                        PIC X(7) VALUE "INVOICE".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  9.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  10.
                03      COLUMN  1
                        PIC X(6) VALUE "| LAST".
                03      COLUMN 16
                        PIC X(7) VALUE "| FIRST".
                03      COLUMN 26
                        PIC X(4) VALUE "|MI|".
                03      COLUMN  35
                        PIC X(6) VALUE "STREET".
                03      COLUMN 48
                        PIC X VALUE "|".
                03      COLUMN 52
                        PIC X(4) VALUE "CITY".
                03      COLUMN  71
                        PIC X VALUE "|".
                03      COLUMN 72
                        PIC X(2) VALUE "ST".
                03      COLUMN 74
                        PIC X VALUE "|".
                03      COLUMN 76
                        PIC X(3) VALUE "ZIP".
                03      COLUMN 81
```

(continued on next page)

Example 16–8 (Cont.): Sample Program 4

```
                        PIC X VALUE "|".
                03      COLUMN 83
                        PIC X(4) VALUE "DATE".
                03      COLUMN 88
                        PIC X VALUE "|".
                03      COLUMN  90
                        PIC X(6) VALUE "NUMBER".
                03      COLUMN 98
                        PIC X VALUE "|".
                03      COLUMN 103
                        PIC X(6) VALUE "AMOUNT".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  11.
                03      COLUMN 1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 1.
        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME              GROUP INDICATE.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME             GROUP INDICATE.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT            GROUP INDICATE.
        02 COLUMN 30    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 51    PIC X(20) SOURCE CITY.
        02 COLUMN 72    PIC XX    SOURCE STATE.
        02 COLUMN 75    PIC 99999 SOURCE ZIP.
        02 COLUMN 81    PIC Z9    SOURCE INV-DAY.
        02 COLUMN 83    PIC X     VALUE "-".
        02 COLUMN 84    PIC 99    SOURCE INV-MO.
        02 COLUMN 86    PIC X     VALUE "-".
        02 COLUMN 87    PIC 99    SOURCE INV-YR.
        02 COLUMN 90    PIC 9(6)  SOURCE INVOICE-NUMBER.
        02 COLUMN 97    PIC $$$,$$$,$$$.99-
                                  SOURCE INVOICE-SALES.
        02 DETAIL-COUNT PIC S9(10) SOURCE ONE-COUNT.
        02 INV-AMOUNT   PIC S9(9)V99 SOURCE INVOICE-SALES.

01      PAGE-FOOTING TYPE IS PAGE FOOTING.
        02      LINE 59.
                03      COLUMN 45
                        PIC X(16) VALUE "C O M P A N Y  ".
                03      COLUMN  62
                        PIC X(25) VALUE  "C O N F I D E N T I A L  ".
        02      LINE 60.
                03      COLUMN 45
                        PIC X(16) VALUE "C O M P A N Y  ".
                03      COLUMN  62
                        PIC X(25) VALUE  "C O N F I D E N T I A L  ".
```

(continued on next page)

Example 16–8 (Cont.): Sample Program 4

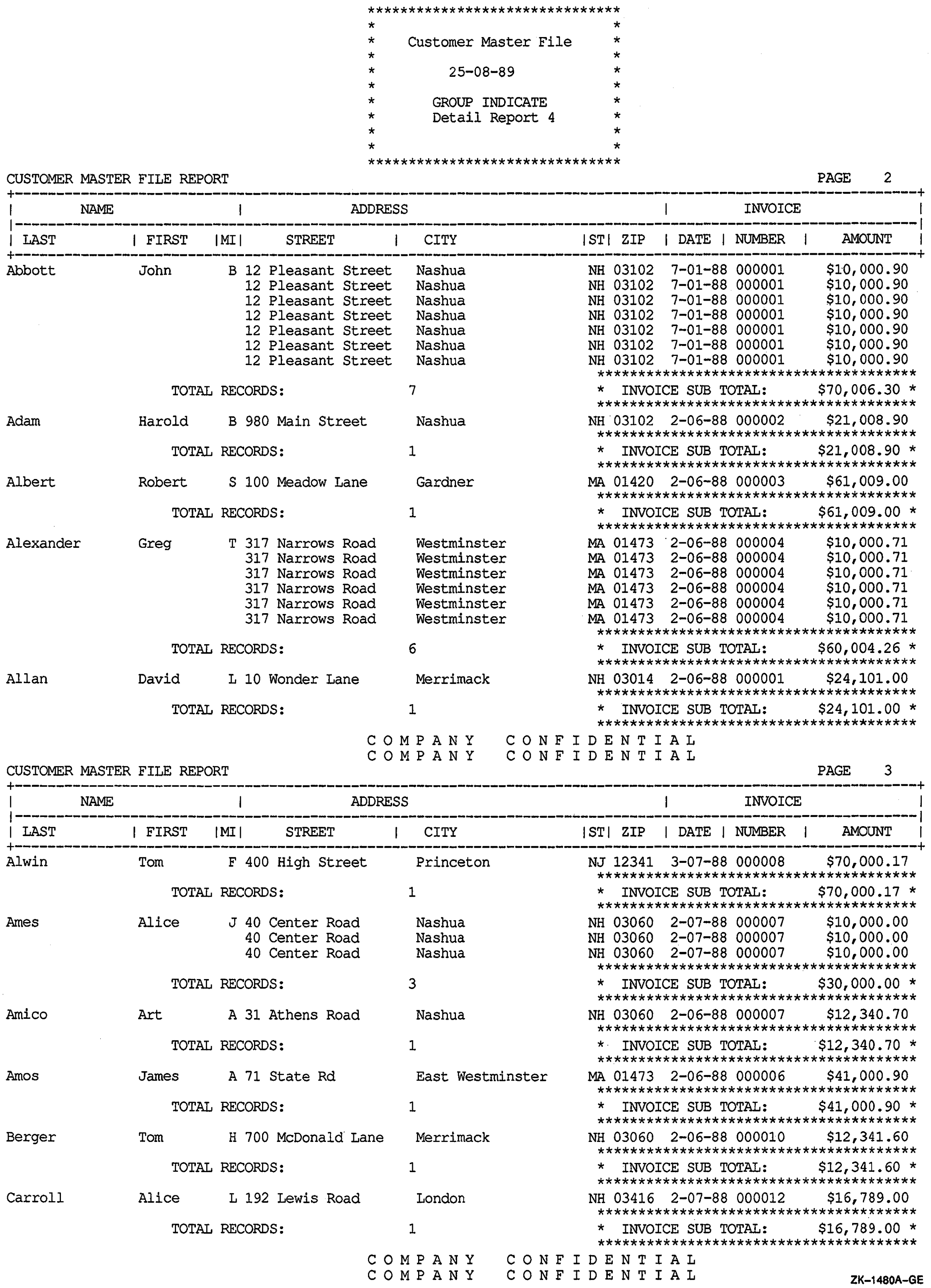
```
01      TYPE IS CONTROL FOOTING NAME
                NEXT GROUP IS PLUS 2.
        02      LINE IS PLUS 2.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  20  PIC X(17) VALUE " TOTAL RECORDS: ".
                03 IDC  COLUMN  40  PIC ZZZ,ZZZ,ZZ9 SUM ONE-COUNT.
                03      COLUMN  73  PIC X(22) VALUE "*  INVOICE SUB TOTAL: ".
                03 IIA  COLUMN  96  PIC $$$,$$$,$$$.99- SUM INVOICE-SALES.
                03      COLUMN  111 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".

01      FINAL-FOOTING TYPE IS CONTROL FOOTING FINAL
                      NEXT GROUP NEXT PAGE.
        02      LINE IS PLUS 2.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  14 PIC X(21) VALUE "GRAND TOTAL RECORDS: ".
                03 FDC  COLUMN  40 PIC ZZZ,ZZZ,ZZ9 SUM IDC.
                03      COLUMN  70 PIC X(24) VALUE "*  GRAND TOTAL INVOICES:".
                03 FIA  COLUMN  94 PIC $,$$$,$$$,$$$.99- SUM IIA.
                03      COLUMN  111 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".

01      REPORT-FOOTER TYPE IS REPORT FOOTING.
        02      LINE 24  ON NEXT PAGE COLUMN  45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN  45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN  55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN  57
```

(continued on next page)

Example 16–8 (Cont.): Sample Program 4

```
                        PIC X   VALUE "-".
                03      COLUMN  58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN  60
                        PIC X   VALUE "-".
                03      COLUMN  61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN  75
                        PIC X VALUE "*".
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30 COLUMN  45
                        PIC X(31) VALUE "*        End of Report 4      *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 32 COLUMN  45
                        PIC X(31) VALUE ALL "*".

PROCEDURE DIVISION.
DECLARATIVES.
BOR SECTION.
        USE BEFORE REPORTING REPORT-HEADER.
BOR-A.
        DISPLAY "BEGINNING-OF-REPORT".
EOR SECTION.
        USE BEFORE REPORTING REPORT-FOOTER.
EOR-A.
        DISPLAY "END-OF-REPORT".
END DECLARATIVES.

MAIN SECTION.
000-DO-SORT.
        SORT SORT-FILE ON ASCENDING KEY SORT-NAME
                WITH DUPLICATES IN ORDER
                USING CUSTOMER-FILE
                GIVING SORTED-FILE.
000-START.
        OPEN INPUT  SORTED-FILE.
        OPEN OUTPUT PRINTER-FILE.
        ACCEPT UNEDITED-DATE FROM DATE.
        INITIATE MASTER-LIST.
        PERFORM 200-READ-MASTER UNTIL NAME = HIGH-VALUES.
100-END-OF-FILE.
        TERMINATE MASTER-LIST.
        CLOSE SORTED-FILE, PRINTER-FILE.
        STOP RUN.
200-READ-MASTER.
        READ SORTED-FILE AT END MOVE HIGH-VALUES TO NAME.
        IF NAME NOT = HIGH-VALUES GENERATE DETAIL-LINE.
```

Figure 16–18: CUSTMAST4.LIS

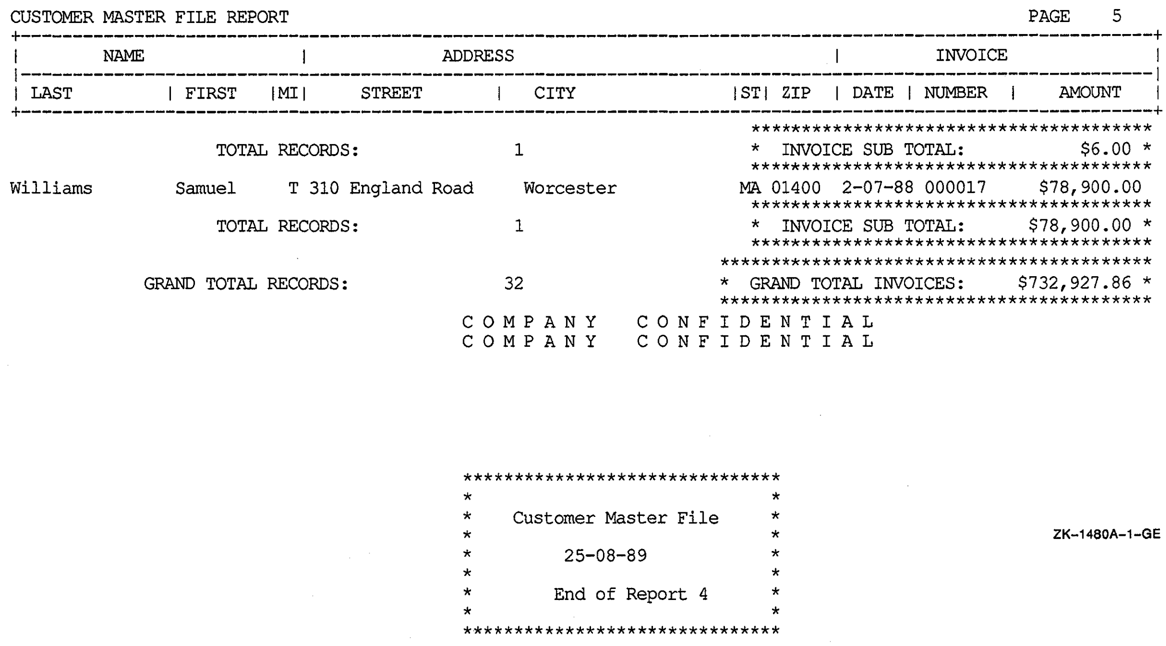


```
                                            *******************************
                                            *                             *
                                            *    Customer Master File     *
                                            *                             *
                                            *        25-08-89             *
                                            *                             *
                                            *      GROUP INDICATE         *
                                            *      Detail Report 4        *
                                            *                             *
                                            *                             *
                                            *******************************
CUSTOMER MASTER FILE REPORT                                                                       PAGE     2
+--------------------------------------------------------------------------------------------------------------+
|        NAME               |             ADDRESS                               |        INVOICE               |
|--------------------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|     STREET       |   CITY               |ST| ZIP  | DATE | NUMBER  |   AMOUNT    |
+--------------------------------------------------------------------------------------------------------------+
Abbott          John       B 12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                             12 Pleasant Street   Nashua               NH 03102  7-01-88 000001     $10,000.90
                                                                        ***************************************
                    TOTAL RECORDS:               7                      *  INVOICE SUB TOTAL:      $70,006.30 *
                                                                        ***************************************
Adam            Harold     B 980 Main Street      Nashua               NH 03102  2-06-88 000002     $21,008.90
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $21,008.90 *
                                                                        ***************************************
Albert          Robert     S 100 Meadow Lane      Gardner              MA 01420  2-06-88 000003     $61,009.00
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $61,009.00 *
                                                                        ***************************************
Alexander       Greg       T 317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                             317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                             317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                             317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                             317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                             317 Narrows Road     Westminster          MA 01473  2-06-88 000004     $10,000.71
                                                                        ***************************************
                    TOTAL RECORDS:               6                      *  INVOICE SUB TOTAL:      $60,004.26 *
                                                                        ***************************************
Allan           David      L 10 Wonder Lane       Merrimack            NH 03014  2-06-88 000001     $24,101.00
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $24,101.00 *
                                                                        ***************************************
                                            C O M P A N Y    C O N F I D E N T I A L
                                            C O M P A N Y    C O N F I D E N T I A L
CUSTOMER MASTER FILE REPORT                                                                       PAGE     3
+--------------------------------------------------------------------------------------------------------------+
|        NAME               |             ADDRESS                               |        INVOICE               |
|--------------------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|     STREET       |   CITY               |ST| ZIP  | DATE | NUMBER  |   AMOUNT    |
+--------------------------------------------------------------------------------------------------------------+
Alwin           Tom        F 400 High Street      Princeton            NJ 12341  3-07-88 000008     $70,000.17
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $70,000.17 *
                                                                        ***************************************
Ames            Alice      J 40 Center Road       Nashua               NH 03060  2-07-88 000007     $10,000.00
                             40 Center Road       Nashua               NH 03060  2-07-88 000007     $10,000.00
                             40 Center Road       Nashua               NH 03060  2-07-88 000007     $10,000.00
                                                                        ***************************************
                    TOTAL RECORDS:               3                      *  INVOICE SUB TOTAL:      $30,000.00 *
                                                                        ***************************************
Amico           Art        A 31 Athens Road       Nashua               NH 03060  2-06-88 000007     $12,340.70
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $12,340.70 *
                                                                        ***************************************
Amos            James      A 71 State Rd          East Westminster     MA 01473  2-06-88 000006     $41,000.90
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $41,000.90 *
                                                                        ***************************************
Berger          Tom        H 700 McDonald Lane    Merrimack            NH 03060  2-06-88 000010     $12,341.60
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $12,341.60 *
                                                                        ***************************************
Carroll         Alice      L 192 Lewis Road       London               NH 03416  2-07-88 000012     $16,789.00
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $16,789.00 *
                                                                        ***************************************
                                            C O M P A N Y    C O N F I D E N T I A L
                                            C O M P A N Y    C O N F I D E N T I A L
```

ZK-1480A-GE

(continued on next page)

Figure 16–18 (Cont.): CUSTMAST4.LIS



```
CUSTOMER MASTER FILE REPORT                                                                        PAGE    5
+--------------------------------------------------------------------------------------------------------------+
|        NAME               |             ADDRESS                               |         INVOICE              |
|--------------------------------------------------------------------------------------------------------------|
| LAST         | FIRST   |MI|     STREET       |   CITY               |ST| ZIP  | DATE | NUMBER  |    AMOUNT   |
+--------------------------------------------------------------------------------------------------------------+
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:           $6.00 *
                                                                        ***************************************
Williams        Samuel     T 310 England Road     Worcester            MA 01400  2-07-88 000017     $78,900.00
                                                                        ***************************************
                    TOTAL RECORDS:               1                      *  INVOICE SUB TOTAL:      $78,900.00 *
                                                                        ***************************************
                                                                     ******************************************
             GRAND TOTAL RECORDS:              32                   *  GRAND TOTAL INVOICES:     $732,927.86 *
                                                                     ******************************************
                                            C O M P A N Y    C O N F I D E N T I A L
                                            C O M P A N Y    C O N F I D E N T I A L




                                            *******************************
                                            *                             *
                                            *    Customer Master File     *
                                            *                             *
                                            *         25-08-89            *
                                            *                             *
                                            *        End of Report 4      *
                                            *                             *
                                            *******************************
```

ZK-1480A-1-GE

16.9.6 REPORT5—Summary Report Program

Example 16–9 (REPORT5) is a Report Writer program that uses the REPORT HEADING, PAGE HEADING, DETAIL, CONTROL FOOTING, PAGE FOOTING, and REPORT FOOTING report groups. The program produces a summary report—CUSTMAST5.LIS (shown in Figure 16–19)—because the GENERATE statement specifies a report name (MASTER-LIST) rather than a DETAIL report group.

Example 16–9: Sample Program 5

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT5.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOMER-FILE ASSIGN TO "SYS$DISK:".
        SELECT SORT-FILE     ASSIGN TO "SYS$DISK:".
        SELECT SORTED-FILE   ASSIGN TO "SYS$DISK:".
        SELECT PRINTER-FILE  ASSIGN TO "SYS$DISK:".
```

(continued on next page)

Example 16-9 (Cont.): Sample Program 5

```
DATA DIVISION.
FILE SECTION.

SD      SORT-FILE.
01      SORTED-CUSTOMER-MASTER-FILE.
        02  SORT-NAME                   PIC X(26).
        02                              PIC X(71).

FD      CUSTOMER-FILE
        VALUE OF ID IS "CUSTMAST.DAT".
01      CUSTOMER-MASTER-FILE            PIC X(97).

FD      SORTED-FILE
        VALUE OF ID IS "SORTEDMAS.DAT".
01      CUSTOMER-MASTER-FILE.
        02  NAME.
                03  LAST-NAME           PIC X(15).
                03  FIRST-NAME          PIC X(10).
                03  MIDDLE-INIT         PIC X.
        02  ADDRESS                     PIC X(20).
        02  CITY                        PIC X(20).
        02  STATE                       PIC XX.
        02  ZIP                         PIC 99999.
        02  SALESMAN-NUMBER             PIC 99999.
        02  INVOICE-DATA.
                03  INVOICE-NUMBER      PIC 999999.
                03  INVOICE-SALES       PIC S9(5)V99.
                03  INVOICE-DATE.
                        04  INV-DAY     PIC 99.
                        04  INV-MO      PIC 99.
                        04  INV-YR      PIC 99.

FD      PRINTER-FILE
        VALUE OF ID IS "CUSTMAST5.LIS"
        REPORT IS MASTER-LIST.

WORKING-STORAGE SECTION.
01      UNEDITED-DATE.
        02  UE-YEAR     PIC 99.
        02  UE-MONTH    PIC 99.
        02  UE-DAY      PIC 99.
        02  FILLER      PIC X(6).

01      ONE-COUNT       PIC 9 VALUE 1.

REPORT SECTION.
RD      MASTER-LIST
        PAGE LIMIT IS 66
        HEADING       1
        FIRST DETAIL  13
        LAST DETAIL   55
        FOOTING       58
        CONTROLS ARE FINAL
                    NAME.
01      REPORT-HEADER TYPE IS REPORT HEADING NEXT GROUP NEXT PAGE.
        02      LINE 24.
                03      COLUMN 45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN 45
```

(continued on next page)

Example 16–9 (Cont.): Sample Program 5

```
                        PIC X(31) VALUE "*     Customer Master File    *".
        02      LINE 27.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN 45
                        PIC X VALUE "*".
                03      COLUMN 55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN 57
                        PIC X   VALUE "-".
                03      COLUMN 58
                        PIC 99
                        SOURCE UE-MONTH.
                03      COLUMN 60
                        PIC X   VALUE "-".
                03      COLUMN 61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30.
                03      COLUMN  45
                        PIC X(31) VALUE "*         Report 5            *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X(31) VALUE "*       Summary Report        *".
        02      LINE 32.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 33.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 34.
                03      COLUMN  45
                        PIC X(31) VALUE ALL "*".
```

(continued on next page)

Example 16–9 (Cont.): Sample Program 5

```
01      TYPE IS PAGE HEADING.
        02      LINE  5.
                03      COLUMN 1
                        PIC X(27) VALUE "CUSTOMER MASTER FILE REPORT".
                03      COLUMN 100
                        PIC X(4)  VALUE "PAGE".
                03      COLUMN 105
                        PIC ZZZ9
                        SOURCE PAGE-COUNTER.
        02      LINE  7.
                03      COLUMN  1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".
        02      LINE  8.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 10
                        PIC X(4) VALUE "NAME".
                03      COLUMN  29
                        PIC X VALUE "|".
                03      COLUMN 43
                        PIC X(7) VALUE "ADDRESS".
                03      COLUMN  81
                        PIC X VALUE "|".
                03      COLUMN  91
                        PIC X(7) VALUE "INVOICE".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  9.
                03      COLUMN  1
                        PIC X VALUE "|".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  10.
                03      COLUMN  1
                        PIC X(6) VALUE "| LAST".
                03      COLUMN 16
                        PIC X(7) VALUE "| FIRST".
                03      COLUMN 26
                        PIC X(4) VALUE "|MI|".
                03      COLUMN  35
                        PIC X(6) VALUE "STREET".
```

(continued on next page)

Example 16–9 (Cont.): Sample Program 5

```
                03      COLUMN 48
                        PIC X VALUE "|".
                03      COLUMN 52
                        PIC X(4) VALUE "CITY".
                03      COLUMN  71
                        PIC X VALUE "|".
                03      COLUMN 72
                        PIC X(2) VALUE "ST".
                03      COLUMN 74
                        PIC X VALUE "|".
                03      COLUMN 76
                        PIC X(3) VALUE "ZIP".
                03      COLUMN 81
                        PIC X VALUE "|".
                03      COLUMN 83
                        PIC X(4) VALUE "DATE".
                03      COLUMN 88
                        PIC X VALUE "|".
                03      COLUMN  90
                        PIC X(6) VALUE "NUMBER".
                03      COLUMN 98
                        PIC X VALUE "|".
                03      COLUMN 103
                        PIC X(6) VALUE "AMOUNT".
                03      COLUMN 112
                        PIC X VALUE "|".
        02      LINE  11.
                03      COLUMN 1
                        PIC X VALUE "+".
                03      COLUMN 2
                        PIC X(110) VALUE ALL "-".
                03      COLUMN 112
                        PIC X VALUE "+".

01      DETAIL-LINE
        TYPE DETAIL
        LINE PLUS 1.
        02 COLUMN 1     PIC X(15) SOURCE LAST-NAME        GROUP INDICATE.
        02 COLUMN 17    PIC X(10) SOURCE FIRST-NAME       GROUP INDICATE.
        02 COLUMN 28    PIC XX    SOURCE MIDDLE-INIT      GROUP INDICATE.
        02 COLUMN 30    PIC X(20) SOURCE ADDRESS.
        02 COLUMN 51    PIC X(20) SOURCE CITY.
        02 COLUMN 72    PIC XX    SOURCE STATE.
        02 COLUMN 75    PIC 99999 SOURCE ZIP.
        02 COLUMN 81    PIC Z9    SOURCE INV-DAY.
        02 COLUMN 83    PIC X     VALUE "-".
        02 COLUMN 84    PIC 99    SOURCE INV-MO.
        02 COLUMN 86    PIC X     VALUE "-".
        02 COLUMN 87    PIC 99    SOURCE INV-YR.
        02 COLUMN 90    PIC 9(6)  SOURCE INVOICE-NUMBER.
        02 COLUMN 97    PIC $$$,$$$,$$$.99-
                                  SOURCE INVOICE-SALES.
        02 DETAIL-COUNT PIC S9(10) SOURCE ONE-COUNT.
        02 INV-AMOUNT   PIC S9(9)V99 SOURCE INVOICE-SALES.
```

(continued on next page)

Example 16–9 (Cont.): Sample Program 5

```
01      TYPE IS CONTROL FOOTING NAME
                NEXT GROUP IS PLUS 2.
        02      LINE IS PLUS 2.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  20  PIC X(17) VALUE " TOTAL RECORDS: ".
                03 IDC  COLUMN  40  PIC ZZZ,ZZZ,ZZ9 SUM ONE-COUNT.
                03      COLUMN  73  PIC X(22) VALUE "*  INVOICE SUB TOTAL: ".
                03 IIA  COLUMN  96  PIC $$$,$$$,$$$.99- SUM INVOICE-SALES.
                03      COLUMN  111 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  73
                        PIC X(39) VALUE ALL "*".
01      FINAL-FOOTING TYPE IS CONTROL FOOTING FINAL
                      NEXT GROUP NEXT PAGE.
        02      LINE IS PLUS 2.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".
        02      LINE IS PLUS 1.
                03      COLUMN  14 PIC X(21) VALUE "GRAND TOTAL RECORDS: ".
                03 FDC  COLUMN  40 PIC ZZZ,ZZZ,ZZ9 SUM IDC.
                03      COLUMN  70 PIC X(24) VALUE "*  GRAND TOTAL INVOICES:".
                03 FIA  COLUMN  94 PIC $,$$$,$$$,$$$.99- SUM IIA.
                03      COLUMN  111 PIC X VALUE "*".
        02      LINE IS PLUS 1.
                03      COLUMN  70
                        PIC X(42) VALUE ALL "*".
01      REPORT-FOOTER TYPE IS REPORT FOOTING.
        02      LINE 24  ON NEXT PAGE COLUMN  45
                        PIC X(31) VALUE ALL "*".
        02      LINE 25.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 26.
                03      COLUMN  45
                        PIC X(31) VALUE "*    Customer Master File     *".
        02      LINE 27.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 28.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN  55
                        PIC Z9
                        SOURCE UE-DAY.
                03      COLUMN  57
                        PIC X   VALUE "-".
                03      COLUMN  58
                        PIC 99
                        SOURCE UE-MONTH.
```

(continued on next page)

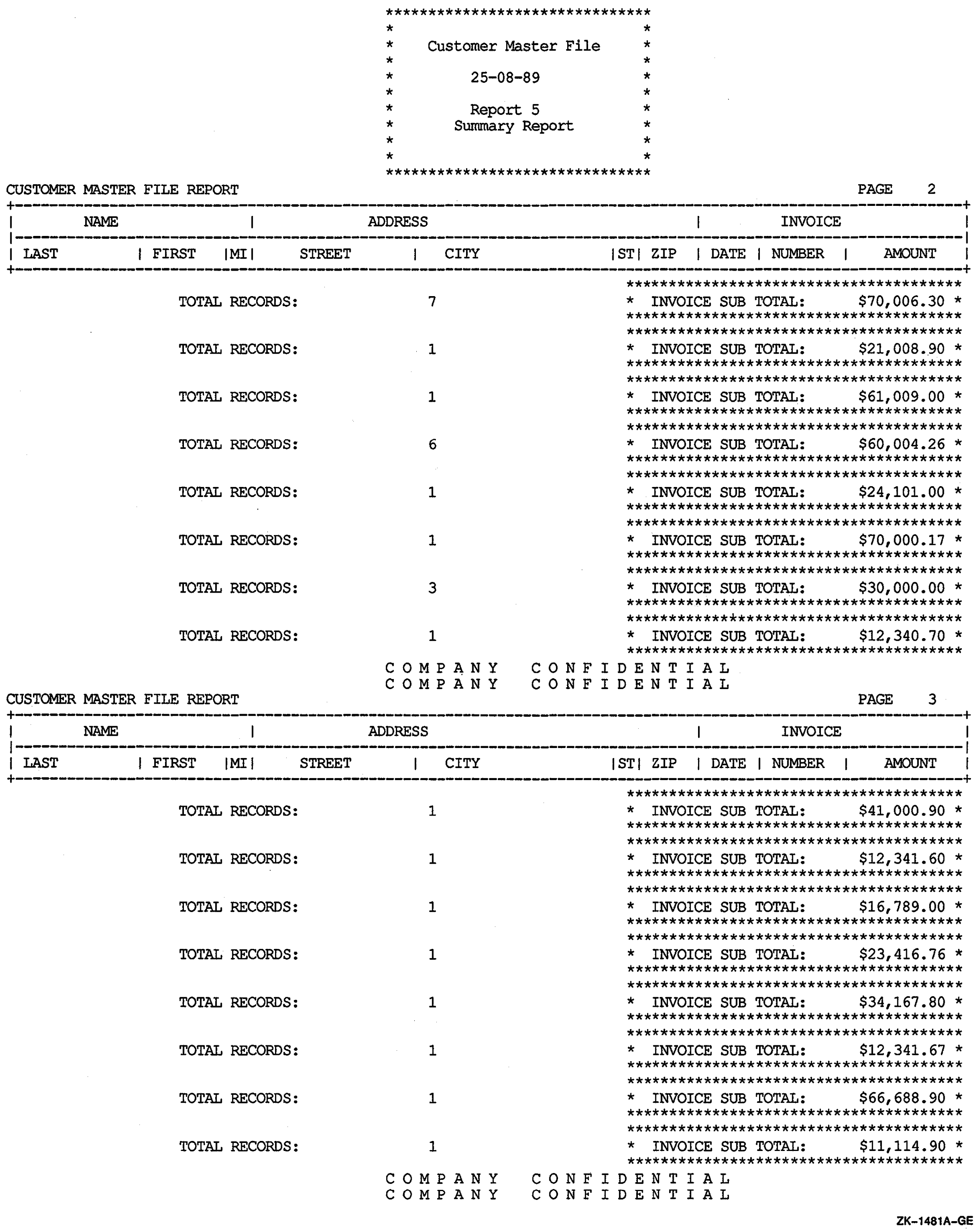
```
                03      COLUMN  60
                        PIC X   VALUE "-".
                03      COLUMN  61
                        PIC 99
                        SOURCE UE-YEAR.
                03      COLUMN  75
                        PIC X VALUE "*".
        02      LINE 29.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 30 COLUMN  45
                        PIC X(31) VALUE "*        End of Report 5      *".
        02      LINE 31.
                03      COLUMN  45
                        PIC X VALUE "*".
                03      COLUMN 75
                        PIC X VALUE "*".
        02      LINE 32 COLUMN  45
                        PIC X(31) VALUE ALL "*".

01      PAGE-FOOTING TYPE IS PAGE FOOTING.
        02      LINE 59.
                03      COLUMN 45
                        PIC X(16) VALUE "C O M P A N Y  ".
                03      COLUMN  62
                        PIC X(25) VALUE  "C O N F I D E N T I A L  ".
        02      LINE 60.
                03      COLUMN 45
                        PIC X(16) VALUE "C O M P A N Y  ".
                03      COLUMN  62
                        PIC X(25) VALUE  "C O N F I D E N T I A L  ".

PROCEDURE DIVISION.
DECLARATIVES.
BOR SECTION.
        USE BEFORE REPORTING REPORT-HEADER.
BOR-A.
        DISPLAY "BEGINNING-OF-REPORT".
EOR SECTION.
        USE BEFORE REPORTING REPORT-FOOTER.
EOR-A.
        DISPLAY "END-OF-REPORT".
END DECLARATIVES.

MAIN SECTION.
000-DO-SORT.
       SORT SORT-FILE ON ASCENDING KEY SORT-NAME
               WITH DUPLICATES IN ORDER
               USING CUSTOMER-FILE
               GIVING SORTED-FILE.
```

(continued on next page)

Example 16–9 (Cont.): Sample Program 5

```
000-START.
        OPEN INPUT  SORTED-FILE.
        OPEN OUTPUT PRINTER-FILE.
        ACCEPT UNEDITED-DATE FROM DATE.
        INITIATE MASTER-LIST.
        PERFORM 200-READ-MASTER UNTIL NAME = HIGH-VALUES.
100-END-OF-FILE.
        TERMINATE MASTER-LIST.
        CLOSE SORTED-FILE, PRINTER-FILE.
        STOP RUN.
200-READ-MASTER.
        READ SORTED-FILE AT END MOVE HIGH-VALUES TO NAME.
        IF NAME NOT = HIGH-VALUES GENERATE MASTER-LIST.
```

Figure 16–19: CUSTMAST5.LIS

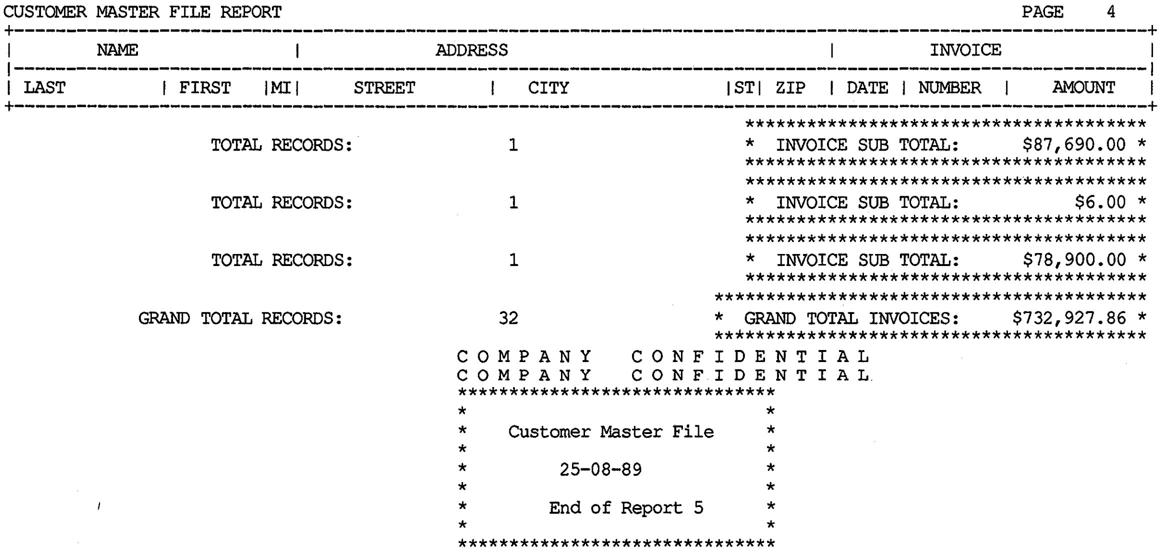


```
                                   *******************************
                                   *                             *
                                   *     Customer Master File    *
                                   *                             *
                                   *          25-08-89           *
                                   *                             *
                                   *          Report 5           *
                                   *       Summary Report        *
                                   *                             *
                                   *                             *
                                   *******************************
CUSTOMER MASTER FILE REPORT                                                                      PAGE    2
+---------------------------------------------------------------------------------------------------------+
|         NAME             |                 ADDRESS                         |          INVOICE           |
|---------------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|      STREET      |   CITY             |ST| ZIP  | DATE | NUMBER  |   AMOUNT  |
+---------------------------------------------------------------------------------------------------------+
                                                              ****************************************
                    TOTAL RECORDS:            7               *  INVOICE SUB TOTAL:      $70,006.30 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $21,008.90 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $61,009.00 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            6               *  INVOICE SUB TOTAL:      $60,004.26 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $24,101.00 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $70,000.17 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            3               *  INVOICE SUB TOTAL:      $30,000.00 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $12,340.70 *
                                                              ****************************************
                                   C O M P A N Y   C O N F I D E N T I A L
                                   C O M P A N Y   C O N F I D E N T I A L
CUSTOMER MASTER FILE REPORT                                                                      PAGE    3
+---------------------------------------------------------------------------------------------------------+
|         NAME             |                 ADDRESS                         |          INVOICE           |
|---------------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|      STREET      |   CITY             |ST| ZIP  | DATE | NUMBER  |   AMOUNT  |
+---------------------------------------------------------------------------------------------------------+
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $41,000.90 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $12,341.60 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $16,789.00 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $23,416.76 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $34,167.80 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $12,341.67 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $66,688.90 *
                                                              ****************************************
                                                              ****************************************
                    TOTAL RECORDS:            1               *  INVOICE SUB TOTAL:      $11,114.90 *
                                                              ****************************************
                                   C O M P A N Y   C O N F I D E N T I A L
                                   C O M P A N Y   C O N F I D E N T I A L
```

ZK-1481A-GE

(continued on next page)

Figure 16–19 (Cont.): CUSTMAST5.LIS



```
CUSTOMER MASTER FILE REPORT                                                                   PAGE    4
+-----------------------------------------------------------------------------------------------------+
|       NAME              |                  ADDRESS                          |         INVOICE        |
|-----------------------------------------------------------------------------------------------------|
| LAST          | FIRST   |MI|      STREET      |  CITY            |ST| ZIP  | DATE | NUMBER |   AMOUNT  |
+-----------------------------------------------------------------------------------------------------+
                                                                   ***************************************
               TOTAL RECORDS:           1                          *  INVOICE SUB TOTAL:      $87,690.00 *
                                                                   ***************************************
                                                                   ***************************************
               TOTAL RECORDS:           1                          *  INVOICE SUB TOTAL:           $6.00 *
                                                                   ***************************************
                                                                   ***************************************
               TOTAL RECORDS:           1                          *  INVOICE SUB TOTAL:      $78,900.00 *
                                                                   ***************************************
                                                                 ******************************************
          GRAND TOTAL RECORDS:          32                       *  GRAND TOTAL INVOICES:    $732,927.86 *
                                                                 ******************************************
                                    C O M P A N Y    C O N F I D E N T I A L
                                    C O M P A N Y    C O N F I D E N T I A L
                                    ******************************
                                    *                            *
                                    *   Customer Master File     *
                                    *                            *
                                    *        25-08-89            *
                                    *                            *
                                    *       End of Report 5      *
                                    *                            *
                                    ******************************
```

ZK-1481A-1-GE

16.10 Solving Report Problems

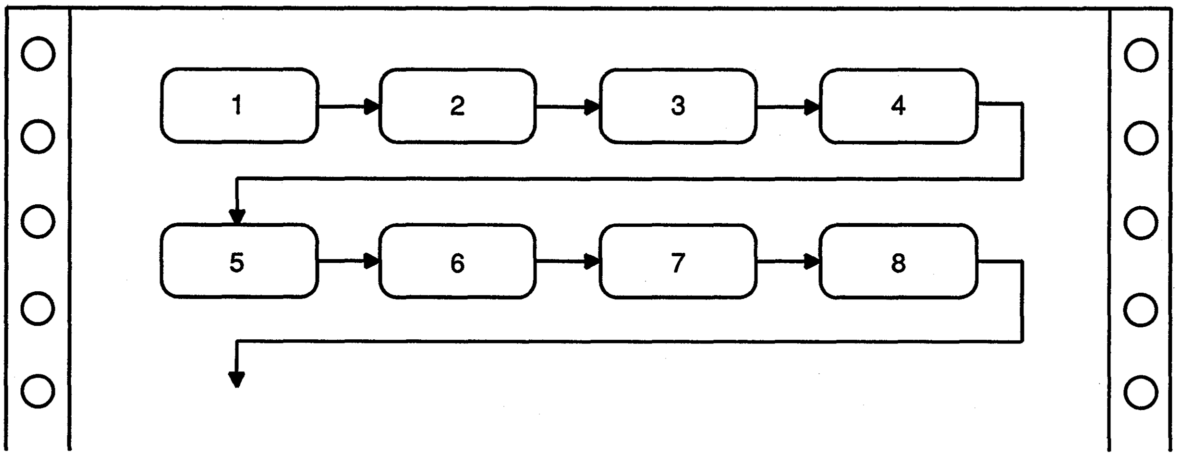
Several variations to the basic report format are discussed in the next sections.

16.10.1 Printing More Than One Logical Line on a Single Physical Line

When your report has only a few columns, you can print several logical lines on one physical line. If you were to print names and addresses on four-up self-sticking multilabel forms, you would print the form left to right and top to bottom, as shown in Figure 16–20 and Example 16–10. To print four-up self-sticking labels, you must format each logical line with four input records.

However, if the columns must be sorted by column, the task becomes more difficult. The last line at the end of the first column is continued at the top of the second column of the same page, indented to the right, and so forth, as shown in Figure 16–21 and Example 16–11. Example 16–11 defines a table containing all data to appear on the page. It reads the input records, stores the data in the table as it is to appear on the page, prints the contents of the table and then fills spaces. When it reaches the end of file, the remaining entries in the table are automatically blank. You can extend this technique to print any number of logical lines on a single physical line.

Figure 16–20: Printing Labels Four-Up



ZK-6088-GE

Example 16–10: Printing Labels Four-Up

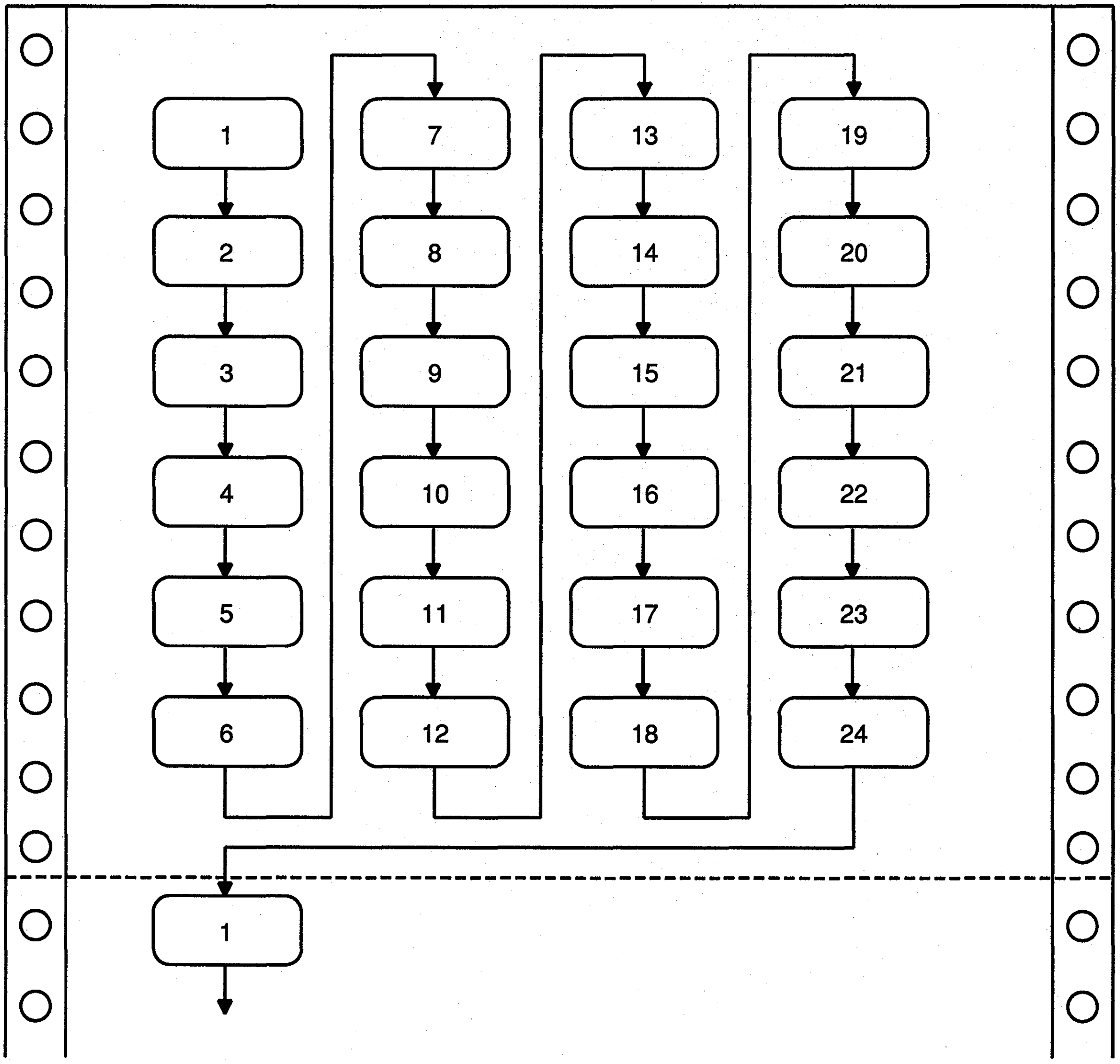
```
IDENTIFICATION DIVISION.
PROGRAM-ID. REP02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "LABELS.DAT".
    SELECT REPORT-FILE ASSIGN TO "LABELS.REP".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-NAME      PIC X(20).
    02  INPUT-ADDRESS   PIC X(15).
    02  INPUT-CITY      PIC X(10).
    02  INPUT-STATE     PIC XX.
    02  INPUT-ZIP       PIC 99999.
FD  REPORT-FILE.
01  REPORT-RECORD       PIC X(132).
WORKING-STORAGE SECTION.
01  LABELS-TABLE.
        03  NAME-LINE.
            05  LINE-1 OCCURS 4 TIMES INDEXED BY INDEX-1.
                07  LABEL-NAME       PIC X(20).
                07  FILLER           PIC X(10).
        03  ADDRESS-LINE.
            05  LINE-2 OCCURS 4 TIMES INDEXED BY INDEX-2.
                07  LABEL-ADDRESS    PIC X(15).
                07  FILLER           PIC X(15).
        03  CSZ-LINE.
            05  LINE-3 OCCURS 4 TIMES INDEXED BY INDEX-3.
```

(continued on next page)

Example 16–10 (Cont.): Printing Labels Four-Up

```
                07  LABEL-CITY          PIC X(10).
                07  FILLER              PIC XXXX.
                07  LABEL-STATE         PIC XX.
                07  FILLER              PIC XXXX.
                07  LABEL-ZIP           PIC 99999.
                07  FILLER              PIC XXXXX.
01  END-OF-FILE                         PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    MOVE SPACES TO LABELS-TABLE.
    SET INDEX-1, INDEX-2, INDEX-3 TO 1.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A050-WRAP-UP.
    IF LABEL-NAME(1) IS NOT EQUAL TO SPACES
       PERFORM A300-PRINT-FOUR-LABELS.
A050-END-OF-JOB.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END OF JOB".
    STOP RUN.
*
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE = "Y" NEXT SENTENCE
       ELSE PERFORM A200-GENERATE-TABLE.
*
A200-GENERATE-TABLE.
    MOVE INPUT-NAME     TO LABEL-NAME(INDEX-1)
    MOVE INPUT-ADDRESS  TO LABEL-ADDRESS(INDEX-2)
    MOVE INPUT-CITY     TO LABEL-CITY(INDEX-3)
    MOVE INPUT-STATE    TO LABEL-STATE(INDEX-3)
    MOVE INPUT-ZIP      TO LABEL-ZIP(INDEX-3)
    IF INDEX-1 = 4 PERFORM A300-PRINT-FOUR-LABELS
       ELSE        SET INDEX-1, INDEX-2, INDEX-3 UP BY 1.
*
A300-PRINT-FOUR-LABELS.
    WRITE REPORT-RECORD FROM NAME-LINE AFTER ADVANCING 3.
    WRITE REPORT-RECORD FROM ADDRESS-LINE AFTER ADVANCING 1.
    WRITE REPORT-RECORD FROM CSZ-LINE AFTER ADVANCING 1.
    MOVE SPACES TO LABELS-TABLE.
    SET INDEX-1, INDEX-2, INDEX-3 TO 1.
```

Figure 16–21: Printing Labels Four-Up in Sort Order



ZK–1556–GE

Example 16–11: Printing Labels Four-Up in Sort Order

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REP03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "LABELS.DAT".
    SELECT REPORT-FILE ASSIGN TO "LABELS.REP".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-NAME      PIC X(20).
    02  INPUT-ADDRESS   PIC X(15).
    02  INPUT-CITY      PIC X(10).
    02  INPUT-STATE     PIC XX.
    02  INPUT-ZIP       PIC 99999.
FD  REPORT-FILE.
01  REPORT-RECORD       PIC X(132).
WORKING-STORAGE SECTION.
01  LABELS-TABLE.
    03  FOUR-UP OCCURS 6 TIMES INDEXED BY ROW-INDEX.
        04  NAME-LINE.
            05  LINE-1 OCCURS 4 TIMES INDEXED BY NAME-INDEX.
                07  LABEL-NAME       PIC X(20).
                07  FILLER           PIC X(10).
        04  ADDRESS-LINE.
            05  LINE-2 OCCURS 4 TIMES INDEXED BY ADDRESS-INDEX.
                07  LABEL-ADDRESS    PIC X(15).
                07  FILLER           PIC X(15).
        04  CSZ-LINE.
            05  LINE-3 OCCURS 4 TIMES INDEXED BY CSZ-INDEX.
                07  LABEL-CITY       PIC X(10).
                07  FILLER           PIC XXXX.
                07  LABEL-STATE      PIC XX.
                07  FILLER           PIC XXXX.
                07  LABEL-ZIP        PIC 99999.
                07  FILLER           PIC XXXXX.
01  END-OF-FILE                      PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    MOVE SPACES TO LABELS-TABLE.
    SET ROW-INDEX, NAME-INDEX, ADDRESS-INDEX, CSZ-INDEX TO 1.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A050-WRAP-UP.
    IF LABEL-NAME(1, 1) IS NOT EQUAL TO SPACES
       PERFORM A300-PRINT-PAGE-OF-LABELS VARYING ROW-INDEX
               FROM 1 BY 1 UNTIL ROW-INDEX IS GREATER THAN 6.
A050-END-OF-JOB.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END OF JOB".
    STOP RUN.
```

(continued on next page)

Example 16–11 (Cont.): Printing Labels Four-Up in Sort Order

```
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE = "Y" NEXT SENTENCE
       ELSE PERFORM A200-GENERATE-LABELS.
A200-GENERATE-LABELS.
    MOVE INPUT-NAME     TO LABEL-NAME(ROW-INDEX, NAME-INDEX)
    MOVE INPUT-ADDRESS  TO LABEL-ADDRESS(ROW-INDEX, ADDRESS-INDEX)
    MOVE INPUT-CITY     TO LABEL-CITY(ROW-INDEX, CSZ-INDEX)
    MOVE INPUT-STATE    TO LABEL-STATE(ROW-INDEX, CSZ-INDEX)
    MOVE INPUT-ZIP      TO LABEL-ZIP(ROW-INDEX, CSZ-INDEX)
    IF ROW-INDEX = 6 AND NAME-INDEX = 4
       PERFORM A300-PRINT-PAGE-OF-LABELS VARYING ROW-INDEX
               FROM 1 BY 1 UNTIL ROW-INDEX IS GREATER THAN 6
       MOVE SPACES TO LABELS-TABLE
       SET ROW-INDEX, NAME-INDEX, ADDRESS-INDEX, CSZ-INDEX TO 1
     ELSE
       PERFORM A210-UPDATE-INDEXES.
A210-UPDATE-INDEXES.
    IF ROW-INDEX =  6 SET ROW-INDEX      TO 1
                      SET NAME-INDEX
                          ADDRESS-INDEX
                          CSZ-INDEX   UP BY 1
       ELSE
               SET ROW-INDEX UP BY 1.
A300-PRINT-PAGE-OF-LABELS.
    WRITE REPORT-RECORD FROM NAME-LINE(ROW-INDEX)
          AFTER ADVANCING 3.
    WRITE REPORT-RECORD FROM ADDRESS-LINE(ROW-INDEX)
          AFTER ADVANCING 1.
    WRITE REPORT-RECORD FROM CSZ-LINE(ROW-INDEX)
          AFTER ADVANCING 1.
```

16.10.2 Group Indicating

The group indicating process greatly improves a report's readability where long sequences of entries have some element in common. You print the element once, then leave it blank for subsequent lines, as long as there is no change in that element. For example, if your sample file's sort sequence is State (major key) and City (minor key), you get sequences like those in Table 16–1.

Table 16–1: Results of Group Indicating

Without Group Indicating			With Group Indicating		
STATE	CITY	STORE NUMBER	STATE	CITY	STORE NUMBER
Arizona	Grand Canyon	111111	Arizona	Grand Canyon	111111
Arizona	Grand Canyon	123456			123456
Arizona	Grand Canyon	222222			222222
Arizona	Tucson	333333	Arizona	Tucson	333333
Arizona	Tucson	444444			444444

(continued on next page)

Table 16–1 (Cont.): Results of Group Indicating

Without Group Indicating			With Group Indicating		
STATE	**CITY**	**STORE NUMBER**	**STATE**	**CITY**	**STORE NUMBER**
Arizona	Tucson	555555			555555
Massachusetts	Maynard	111111	Massachusetts	Maynard	111111
Massachusetts	Maynard	222222			222222
Massachusetts	Maynard	333333			333333
Massachusetts	Maynard	444444			444444
Massachusetts	Tewksbury	111111	Massachusetts	Tewksbury	111111
Massachusetts	Tewksbury	222222			222222
New Hampshire	Manchester	111111	New Hampshire	Manchester	111111
New Hampshire	Manchester	222222			222222
New Hampshire	Merrimack	333333	New Hampshire	Merrimack	333333
New Hampshire	Merrimack	444444			444444
New Hampshire	Merrimack	555555			555555
New Hampshire	Nashua	666666	New Hampshire	Nashua	666666

16.10.3 Fitting Reports on the Page

If you need more columns than physically can fit on a page, you can do the following:

- Eliminate as many unused spaces as possible between columns. Columns should not be run together; however, you can use one blank space instead of several.
- Eliminate nonessential information.
- Print two or more lines with staggered headers and columns.
- Print two reports.

16.10.4 Printing Totals Before Detail Lines

A report that must include totals at the top of the page before the detail lines has three solutions as follows:

- Store the logical print lines in a table, total the table, and then print from the table.
- Pass through the file twice. The first time, compute the totals. The second time, print the report. This method is slow and complicated if there are many subtotals.
- Write the lines into a file with a sort key containing the report, page, and line number. When your program writes the last line and computes the total, have it assign a page and line number to the total line's sort key. Use an appropriate page and line number to cause the total line to sort in front of its detail lines. After the program completes, sort the file, read it, drop the sort key, and produce the report.

16.10.5 Underlining Items in Your Reports

Sometimes you must underline a column of numbers to denote a total and also underline the total to highlight it:

```
1234
1122
----
2356
====
```

To print a single underline, use the underscore character and suppress line spacing. For example:

```
WRITE PRINT-LINE FROM SINGLE-UNDERLINE-TOTAL
                 BEFORE ADVANCING 0 LINES.
```

This overprints the underscore (_) on the previous line, underlining the item: 1122. Use the equal sign (=) to simulate double underlines. Note that you must write the equal signs on the next line. For example:

```
WRITE PRINT-LINE FROM DOUBLE-UNDERLINE-TOTAL
                 AFTER ADVANCING 1 LINE.
```

16.10.6 Bolding Items in Your Reports

To bold an entire line in a report:

1. Write the line as many times as you want, specifying the BEFORE ADVANCING 0 LINES phrase (three times is sufficient). This darkens the line but does not advance to the next line.
2. Write the line one last time without the BEFORE ADVANCING phrase. This overprints the line again and advances to the next print line.

For example:

```
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE.
```

This example produces a darker image in the report. You can use similar statements for characters and words, as well as complete lines. To bold only a word or only a character within a line, you must:

1. Write the print line and specify the BEFORE ADVANCING 0 LINES phrase.
2. Use reference modification to create a skeleton line containing only the items in the print line you want bolded.
3. Write the skeleton line as many times as you want and specify the BEFORE ADVANCING 0 LINES phrase. This darkens the items in the skeleton line but does not advance to the next line.
4. Write the skeleton line one last time without the BEFORE ADVANCING phrase. This overprints the line again and advances to the next print line.

For example:

```
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
*
* Move spaces over the items in the source print line (TOTAL-LINE)
* that are not to be bolded
*
    MOVE SPACES TO ...
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
    WRITE PRINT-LINE FROM TOTAL-LINE.
```

Chapter 17

Forms for Video Terminals

This chapter explains how you can design an online video form similar to a printed form, using VAX COBOL ACCEPT and DISPLAY statements. These statements provide options for developing video forms on VT52, VT100, VT200, or VT300 [1] terminals, and let you write your application without regard to the type of terminal the application will eventually run on. You can run your forms application on any of these terminals. However, not all options are available for the VT52 terminal; those unavailable options have no effect on the form.

For simple screen applications, or applications requiring specialized screen displays, such as scrolling regions or double-width/double-height displays, refer to the VAX documentation on Run-Time Library (RTL) routines. This manual provides information on terminal-independent screen manipulation RTL procedures.

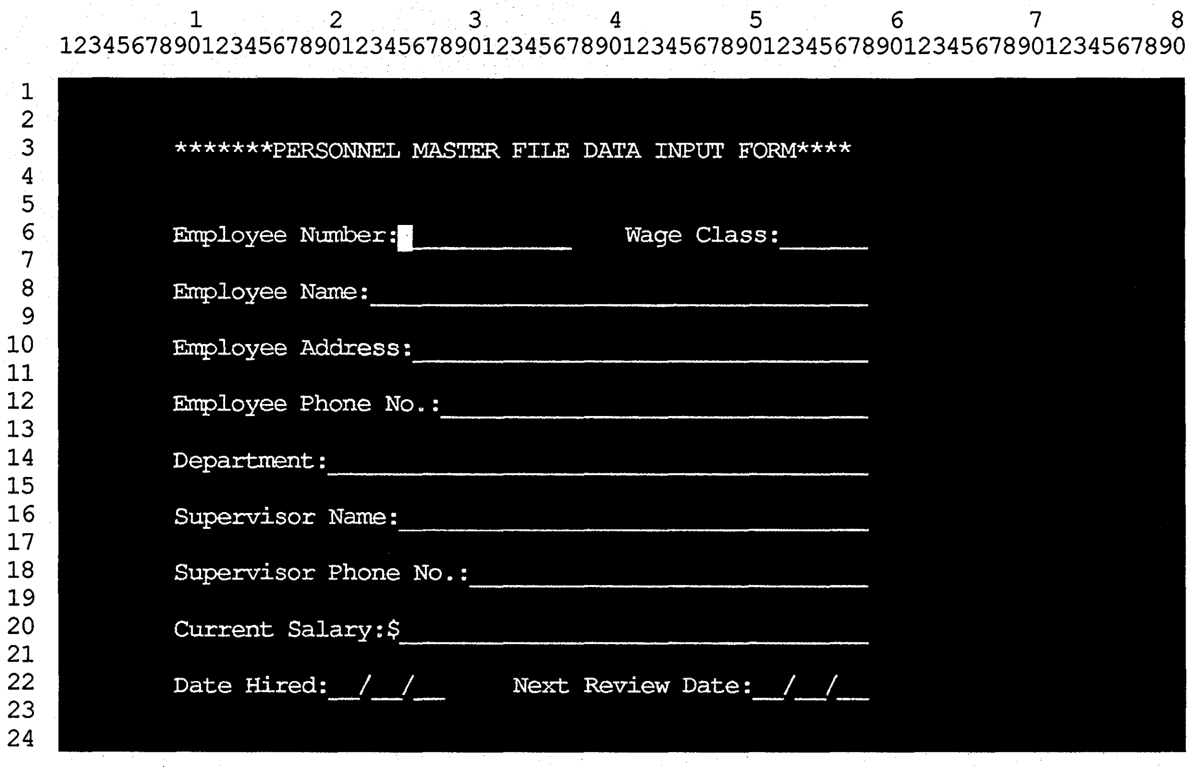
A video form allows you to do the following:

- Improve the appearance of an application's terminal dialogue
- Make data entry applications, menu selections, and special control keys easier to use
- Clarify the type of input expected from an operator

For example, Figure 17–1 is a sample form created by a VAX COBOL program that lets you enter employee information into a master file. This program prompts you for input data to the form. Once all data is entered, the program writes it to the master file and displays a new form.

[1] VAX COBOL does not provide mouse or split screen support

Figure 17–1: Adding Information to a Master File with a Video Form



```
        *******PERSONNEL MASTER FILE DATA INPUT FORM****


        Employee Number:___________    Wage Class:______

        Employee Name:_____________________________________

        Employee Address:__________________________________

        Employee Phone No.:________________________________

        Department:________________________________________

        Supervisor Name:___________________________________

        Supervisor Phone No.:______________________________

        Current Salary:$___________________________________

        Date Hired:__/__/__     Next Review Date:__/__/__
```

ZK–6089–GE

Designing Your Form with ACCEPT/DISPLAY Options

To help you design a video form, the ACCEPT/DISPLAY options allow you to do the following:

- Erase specific parts or the entire screen
- Use relative and absolute cursor positioning
- Specify video attributes of data to be displayed and accepted
- Convert data to appropriate usage when accepting or displaying data
- Handle error conditions when accepting and displaying data
- Provide screen protection by limiting the number of characters typed on the terminal when accepting data
- Accept data without echoing
- Specify default values for ACCEPT statements
- Define and handle special control keys for ACCEPT statements
- Allow field editing.

The remainder of this chapter discusses these topics.

17.1 Clearing a Screen Area

To clear part or all of your screen before you accept or display data, you can use one of the following ERASE options of the ACCEPT and DISPLAY statements:

- ERASE SCREEN—Erases the entire screen before accepting or displaying data at the specified or implied cursor position.
- ERASE LINE—Erases the entire specified line before accepting or displaying data at the specified or implied cursor position.
- ERASE TO END OF SCREEN—Erases from the specified or implied cursor position to the end of the screen before accepting or displaying data at the specified cursor position.
- ERASE TO END OF LINE—Erases from the specified or implied cursor position to the end of the line before accepting or displaying data at the specified cursor position.

Table 17–1 lists the ERASE options and indicates whether each option requires relative or absolute cursor positioning for your terminal type. (See Section 17.2.)

Table 17–1: Cursor Positioning Requirements for ERASE Options

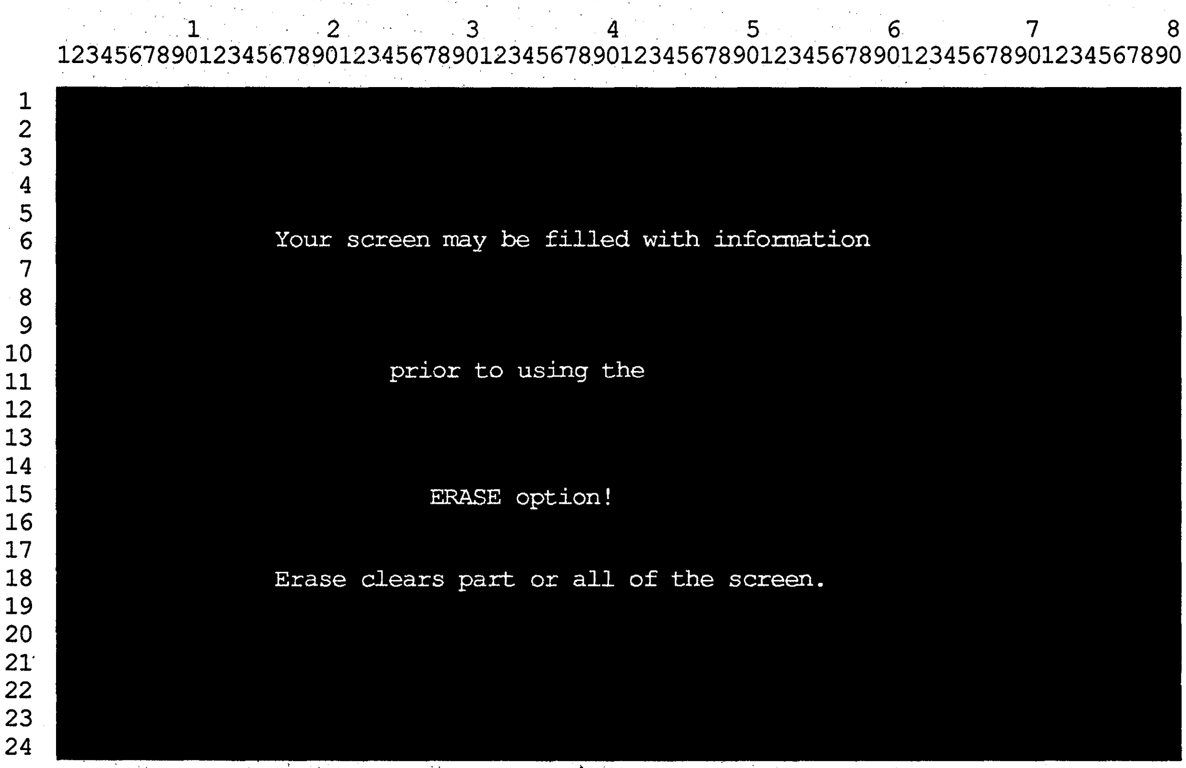
	Cursor Positioning for Your Terminal Type	
ERASE Option	**VT52**	**VT100/VT200/VT300**
ERASE SCREEN	Absolute only	Absolute or Relative
ERASE LINE	Absolute only	Absolute or Relative
ERASE TO END OF SCREEN	Absolute or Relative	Absolute or Relative
ERASE TO END OF LINE	Absolute or Relative	Absolute or Relative

In Example 17–1, the entire screen is erased before Employee number: is displayed. Figure 17–2 shows how the screen looks before the ERASE statement executes. Figure 17–3 shows how the screen looks after the ERASE statement executes.

Example 17–1: Erasing a Screen

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ERASEIT.
DATA DIVISION.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Employee number:" LINE 4 COLUMN 5 ERASE SCREEN.
    DISPLAY " " LINE 23 COLUMN 1.
    STOP RUN.
```

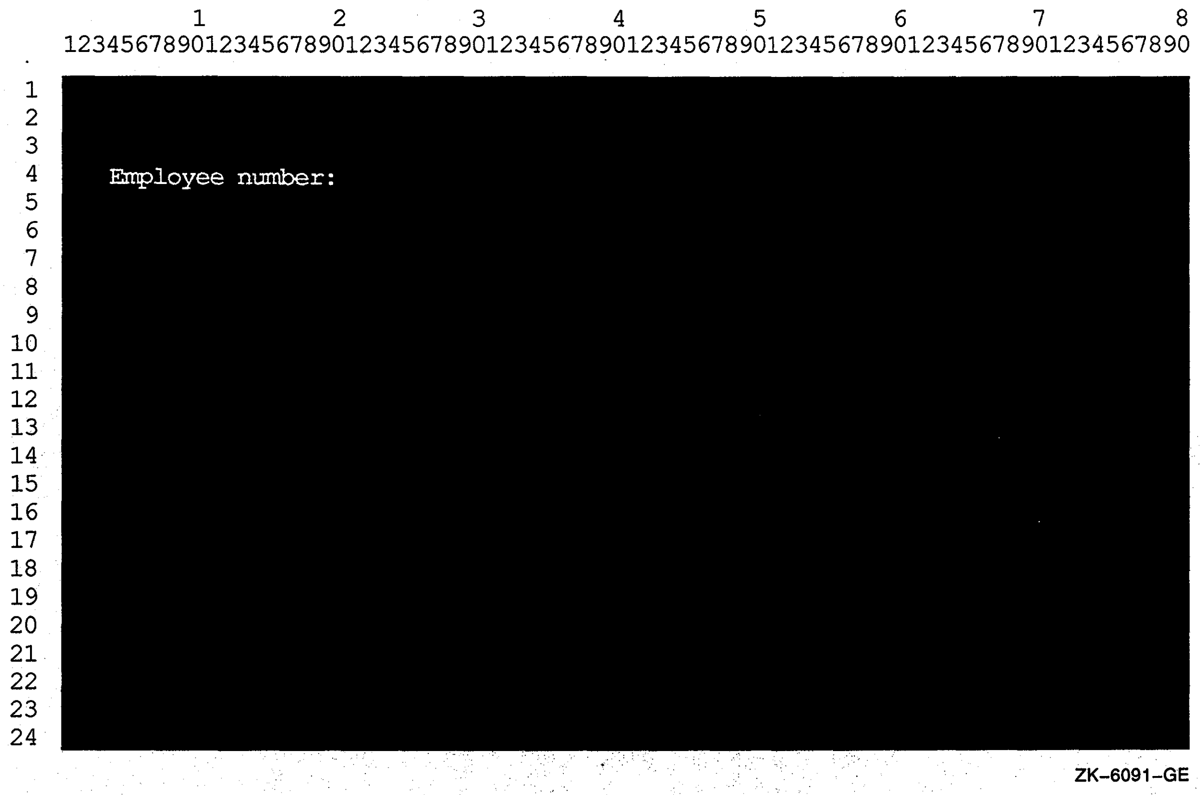
Figure 17–2: Screen Before the ERASE Statement Executes



```
Your screen may be filled with information


prior to using the


ERASE option!

Erase clears part or all of the screen.
```

ZK-6090-GE

Figure 17–3: Screen After the ERASE Statement Executes



```



    Employee number:
```

17.2 Horizontal and Vertical Positioning of the Cursor

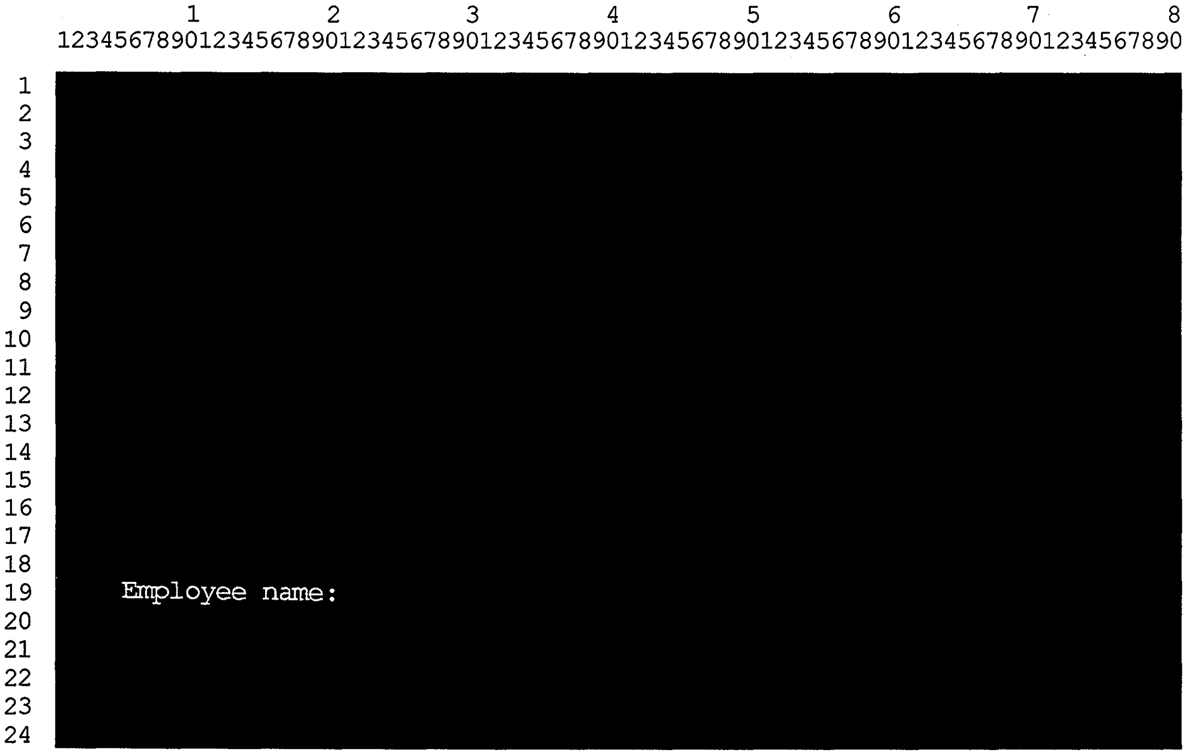
To position data items at a specified line and column, you use the LINE NUMBER and COLUMN NUMBER phrases. You can use these phrases with both the ACCEPT and DISPLAY statements. You can also use literals or numeric data items to specify line and column numbers.

In Example 17–2 and in Figure 17–4, Employee name: is displayed on line 19 in column 5.

Example 17–2: Cursor Positioning

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  LOCATE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COL-NUM              PIC 99   VALUE 5.
PROCEDURE DIVISION.
A00-OUT-PARA.
    DISPLAY "Employee name:"     LINE 19
                                 COLUMN COL-NUM
                                 ERASE SCREEN.
    DISPLAY " " LINE 24
                COLUMN 1.
    STOP RUN.
```

Figure 17–4: Positioning the Data on Line 19, Column 5



```
Employee name:
```

ZK–6092–GE

If you use LINE, but not COLUMN, data is accepted or displayed at column 1 of the specified line position.

If you use COLUMN, but not LINE, data is accepted or displayed at the current line and specified column position.

If you do not use either phrase, data is accepted or displayed at the position specified by the rules for Format 1 ACCEPT and DISPLAY in the *VAX COBOL Reference Manual.*

NOTE

> The presence of either or both the LINE and COLUMN clauses implies NO ADVANCING.

You can use the PLUS option with the LINE or COLUMN clauses for relative cursor positioning. The PLUS option eliminates the need for counting lines or columns. Once you specify an initial LINE or COLUMN number, you can position items by using the LINE PLUS or COLUMN PLUS phrases. If you use PLUS option without an integer, PLUS 1 is implied. Note that cursor positioning is relative to where the cursor is after the previous ACCEPT or DISPLAY.

To get predictable results from your relative cursor positioning statements, do not:

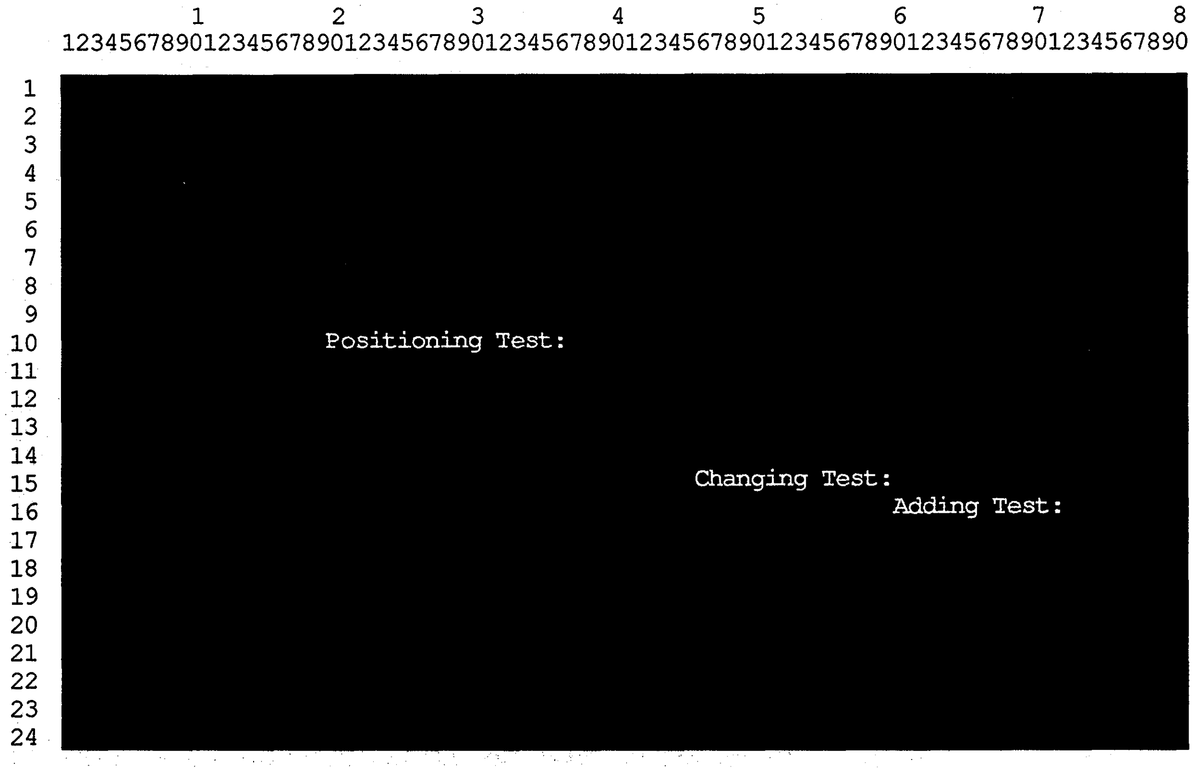
- Cause a display line to wrap around to the next line.
- Accept data into unprotected fields.
- Go beyond the top or bottom of the screen.
- Mix displays of double-high characters and relative cursor positioning.

In Example 17–3 the PLUS phrase is used twice to show relative positioning, once with an integer, and once without. Figure 17–5 shows the results.

Example 17–3: Using PLUS for Cursor Positioning

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LINEPLUS.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Positioning Test" LINE 10      COLUMN 20 ERASE SCREEN
            "Changing Test"    LINE PLUS 5  COLUMN PLUS 10
            "Adding Test"      LINE PLUS    COLUMN PLUS.
    DISPLAY " " LINE 23 COLUMN 1.
    STOP RUN.
```

Figure 17–5: Cursor Positioning Using the PLUS Option



```
Positioning Test:

                    Changing Test:
                                  Adding Test:
```

Positioning Test is displayed on line 10, column 20 of the form; Changing Test is displayed on line 15, column 46 of the form; and Adding Test is displayed on line 16, column 60 of the form.

NOTE

> If you use the LINE PLUS phrase so that relative positioning goes beyond the bottom of the screen, your form scrolls with each such display.

17.3 Assigning Character Attributes to Your Format Entries

Depending on your terminal type, you can use one or more of the character attributes in Table 17–2 to highlight your screen data. Example 17–4 shows the use of these attributes in a program segment. Figure 17–6 shows the results of the program segment in Example 17–4.

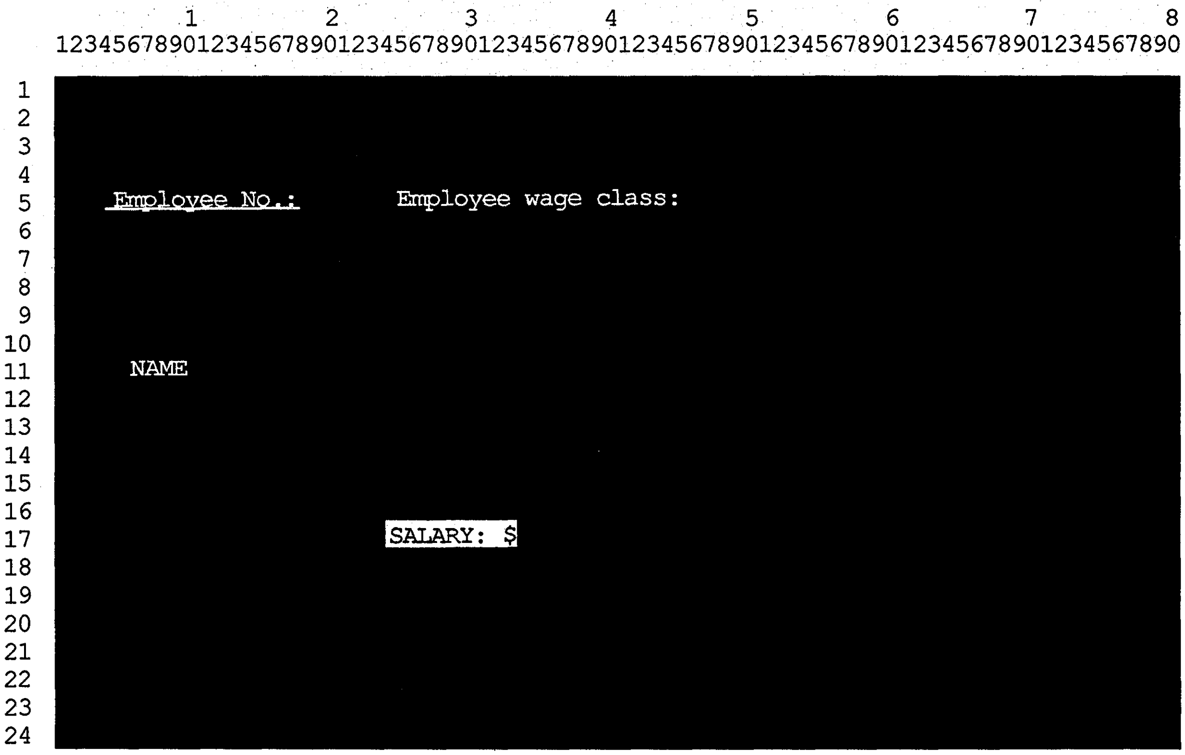
Table 17–2: Available Character Attributes by Terminal Type

Character Attribute	VT100, VT200, and VT300 Series Terminals with Advanced Video Option	VT52 and VT100 Without the Advanced Video Option
BELL sounds your terminal bell	Available	Available
UNDERLINED underlines your text	Available	Not Available
BOLD intensifies your text	Available	Not Available
BLINKING blinks your text	Available	Not Available
REVERSED changes your screen's background	Available	Not Available

Example 17–4: Using Character Attributes

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHARATTR.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Employee No:" UNDERLINED LINE 5 COLUMN 5 ERASE SCREEN.
    DISPLAY "Employee wage class:" BOLD LINE 5 COLUMN 24.
    DISPLAY "NAME" BLINKING LINE PLUS 6  COLUMN 6.
    DISPLAY "SALARY: $" REVERSED LINE PLUS 6 COLUMN 24.
    DISPLAY " " LINE 23 COLUMN 1.
```

Figure 17–6: Screen Display with Character Attributes



```
    Employee No.:        Employee wage class:





     NAME





                        SALARY: $
```

ZK–6093–GE

17.4 Using the CONVERSION Clause to Display Data

Use the CONVERSION clause to display the contents of numeric fields. When you use the CONVERSION clause with a DISPLAY statement, the numeric item appears on the screen:

- In DISPLAY usage
- With a decimal point (if needed) or comma (if DECIMAL-POINT IS COMMA)
- Edited (if needed)
- With a sign (if needed)

This display lets you see the values of non-DISPLAY data items in a form that the user can read. The size of the displayed field is determined by the PICTURE clause of the displayed item. Example 17–5 and Figure 17–7 show how to display different types of data with the CONVERSION clause.

Example 17–5: Using the CONVERSION Clause

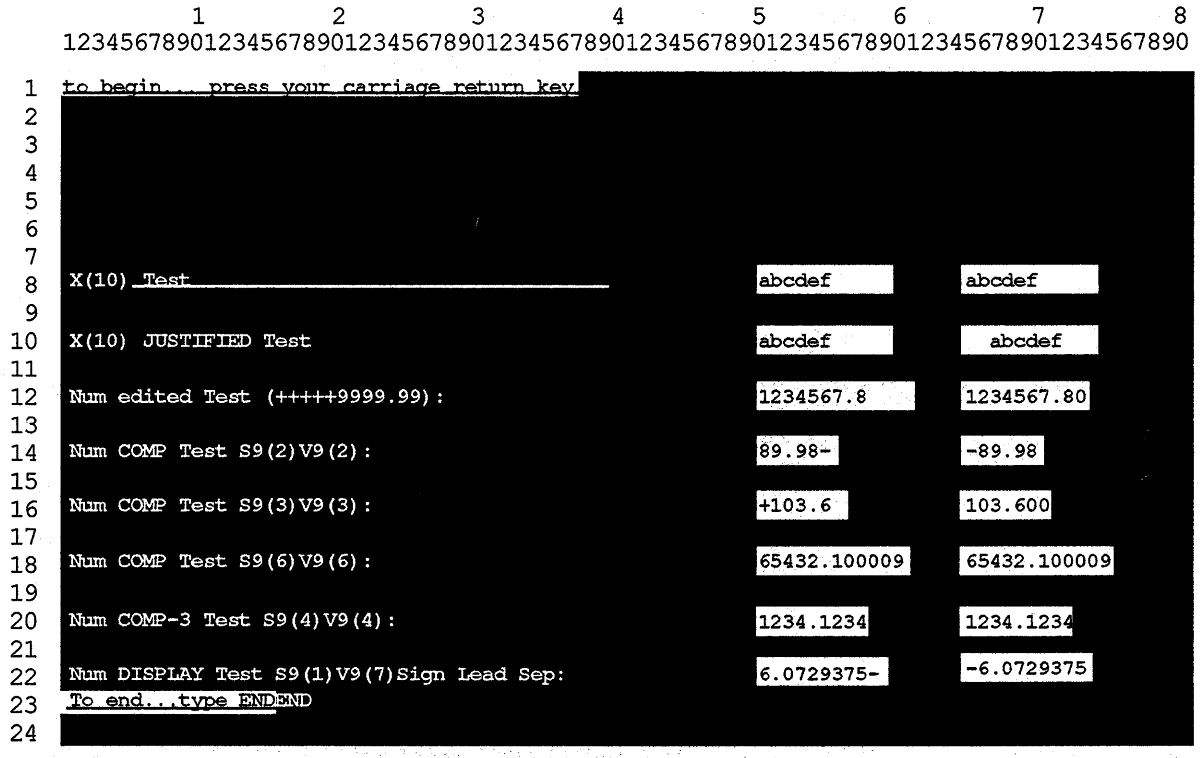
```
IDENTIFICATION DIVISION.
PROGRAM-ID.  CONVERT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA1A     PIC X(10).
01  DATA1B     PIC X(10) JUST.
01  DATA2      PIC +++++9999.99.
01  DATA3      PIC S9(2)V9(2) COMP.
01  DATA4      PIC S9(3)V9(3) COMP.
01  DATA5      PIC S9(6)V9(6) COMP.
01  DATA6      PIC S9(4)V9(4) COMP-3.
01  DATA7      PIC S9(1)V9(7) SIGN LEADING SEPARATE.
PROCEDURE DIVISION.
CONVERT-CHECK SECTION.
P1.
    DISPLAY "To begin... press your carriage return key"
            LINE 1 COLUMN 1 ERASE SCREEN
            BELL UNDERLINED REVERSED.
    ACCEPT DATA1A.
    DISPLAY "X(10) Test" LINE 8 ERASE LINE.
    ACCEPT DATA1A WITH CONVERSION PROTECTED REVERSED
           LINE 8 COLUMN 50.
    DISPLAY DATA1A REVERSED WITH CONVERSION
            LINE 8 COLUMN 65.
    DISPLAY "X(10) JUSTIFIED Test" LINE 10 ERASE LINE.
    ACCEPT DATA1B WITH CONVERSION PROTECTED REVERSED
           LINE 10 COLUMN 50.
    DISPLAY DATA1B REVERSED WITH CONVERSION
            LINE 10 COLUMN 65.
P2.
    DISPLAY "Num edited Test (+++++9999.99):" LINE 12 ERASE LINE.
    ACCEPT DATA2 PROTECTED REVERSED WITH CONVERSION
           LINE 12 COLUMN 50.
    DISPLAY DATA2 REVERSED WITH CONVERSION
            LINE 12 COLUMN 65.
P3.
    DISPLAY "Num COMP Test S9(2)V9(2):" LINE 14 ERASE LINE.
    ACCEPT DATA3 PROTECTED REVERSED WITH CONVERSION
           LINE 14 COLUMN 50.
    DISPLAY DATA3 REVERSED WITH CONVERSION LINE 14 COLUMN 65.
P4.
    DISPLAY "Num COMP Test S9(3)V9(3):" LINE 16 ERASE LINE.
    ACCEPT DATA4 PROTECTED REVERSED WITH CONVERSION
           LINE 16 COLUMN 50.
    DISPLAY DATA4 REVERSED WITH CONVERSION
            LINE 16 COLUMN 65.
```

(continued on next page)

Example 17–5 (Cont.): Using the CONVERSION Clause

```
P5.
    DISPLAY "Num COMP Test S9(6)V9(6):" LINE 18 ERASE LINE.
    ACCEPT DATA5 PROTECTED REVERSED WITH CONVERSION
           LINE 18 COLUMN 50.
    DISPLAY DATA5 REVERSED WITH CONVERSION
            LINE 18  COLUMN 65.
P6.
    DISPLAY "Num COMP-3 Test S9(4)V9(4):" LINE 20 ERASE LINE.
    ACCEPT DATA6 PROTECTED REVERSED WITH CONVERSION
           LINE 20 COLUMN 50.
    DISPLAY DATA6 REVERSED WITH CONVERSION
            LINE 20 COLUMN 65.
P7.
    DISPLAY "Num DISPLAY Test S9(1)V9(7)Sign Lead Sep:"
           LINE 22 ERASE LINE.
    ACCEPT DATA7 PROTECTED REVERSED WITH CONVERSION
           LINE 22 COLUMN 50.
    DISPLAY DATA7 REVERSED WITH CONVERSION
            LINE 22 COLUMN 65.
P8.
    DISPLAY "To end...type END"
            LINE PLUS COLUMN 1 ERASE LINE
            BELL UNDERLINED REVERSED.
    ACCEPT DATA1A.
    IF DATA1A = "END" STOP RUN.
    GO TO P1.
```

Figure 17–7: Sample Run of Program CONVERT



```
to begin... press your carriage return key






X(10) Test                                       abcdef          abcdef

X(10) JUSTIFIED Test                             abcdef            abcdef

Num edited Test (+++++9999.99):                  1234567.8       1234567.80

Num COMP Test S9(2)V9(2):                        89.98-          -89.98

Num COMP Test S9(3)V9(3):                        +103.6          103.600

Num COMP Test S9(6)V9(6):                        65432.100009    65432.100009

Num COMP-3 Test S9(4)V9(4):                      1234.1234       1234.1234

Num DISPLAY Test S9(1)V9(7)Sign Lead Sep:        6.0729375-      -6.0729375
To end...type ENDEND
```

ZK–6094–GE

Note that you can also display COMP-1, COMP-2, RMS-STS, RMS-STV, LINAGE register, RWCS LINE, and RWCS PAGE items; the DBMS registers DB-CONDITION, DB-CURRENT-RECORD-NAME, DB-KEY, DB-UWA and DB-CURRENT-RECORD-ID; and VALUE EXTERNAL and POINTER VALUE REFERENCE items.

17.5 Handling Data with ACCEPT Options

Several ACCEPT phrases help you handle data, including the CONVERSION, AT END, ON EXCEPTION, PROTECTED, SIZE, NO ECHO, and DEFAULT clauses.

17.5.1 Using CONVERSION with ACCEPT Data

When you use the CONVERSION clause with an ACCEPT numeric operand (other than floating point), VAX COBOL converts the data entered on the form to a trailing-signed decimal field. Editing is performed when specified by destination. The data is then moved from the screen to your program using standard MOVE statement rules.

When you use the CONVERSION clause with an ACCEPT numeric floating-point operand, VAX COBOL converts input data to floating-point (COMP-1 or COMP-2 as appropriate). The converted result is then moved to the destination as if moving a numeric literal equivalent to the input data with the MOVE statement.

When an ACCEPT operand is not numeric, the CONVERSION clause moves the input characters as an alphanumeric string, using standard MOVE statement rules. This lets you accept data into an alphanumeric-edited or JUSTIFIED field.

If you use the CONVERSION clause while accepting numeric data, you can also use the ON EXCEPTION clause. This clause lets you control data entry errors that can occur when entering numeric data.

If you do not use the CONVERSION clause, data is transferred to the destination item according to Format 1 ACCEPT statement rules.

17.5.2 Using ON EXCEPTION When Accepting Data with CONVERSION

If you enter illegal numeric data or exceed the PICTURE description (if not protected) of the ACCEPT data (with an overflow to either the left or right of the decimal point), the imperative statement associated with ON EXCEPTION executes, and the destination field does not change.

Example 17–6 (and Figure 17–8) show how the ON EXCEPTION clause executes if you enter an alphanumeric or a numeric item out of the valid range. The statements following ON EXCEPTION prompt you to try again.

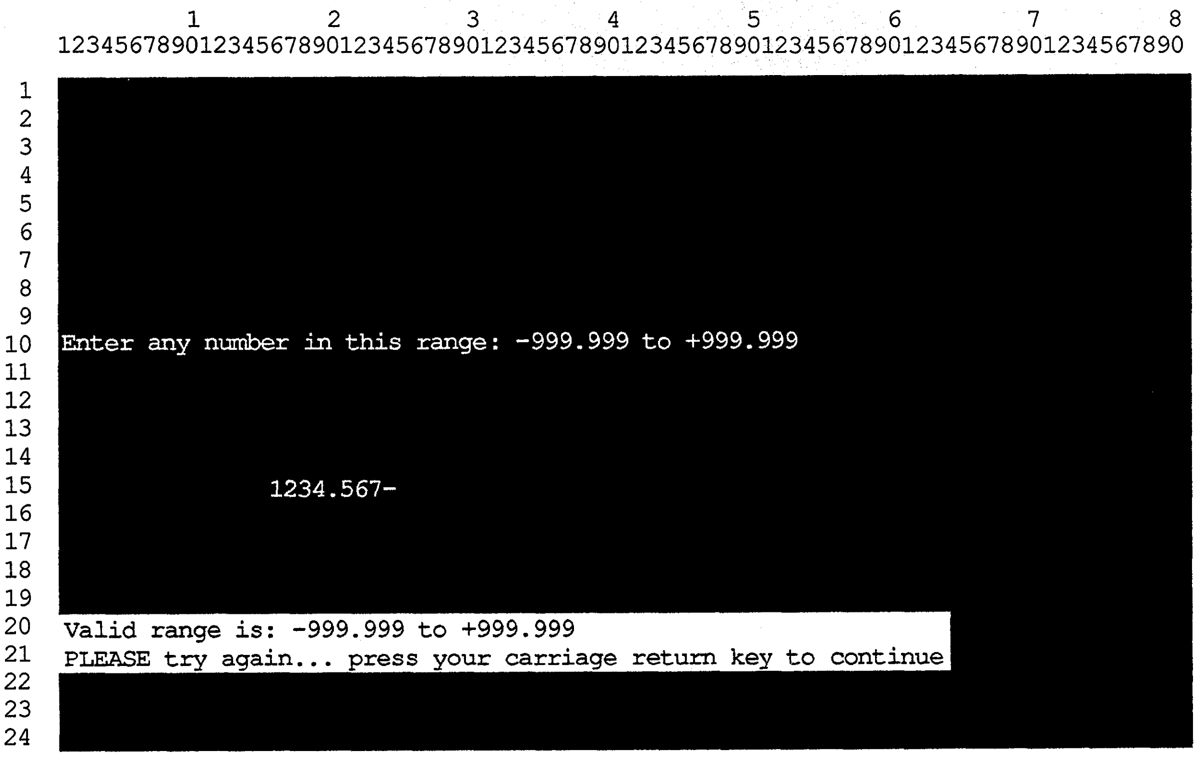
If you do not use ON EXCEPTION and a conversion error occurs:

- The field on the screen is filled with spaces.
- The terminal bell rings and the terminal automatically reprompts you for the data results.
- The contents of the destination field are not changed.

Example 17–6: Using the ON EXCEPTION Clause

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ONEXC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NUM-DATA   PIC S9(3)V9(3) COMP-3.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Enter any number in this range: -999.999 to +999.999"
            LINE 10 COLUMN 1
            ERASE SCREEN.
    ACCEPT NUM-DATA WITH CONVERSION LINE 15 COLUMN 15
        ON EXCEPTION
           DISPLAY "Valid range is: -999.999 to +999.999"
              LINE 20 REVERSED WITH BELL ERASE TO END OF SCREEN
           DISPLAY "PLEASE try again... press your carriage return key"
              " to continue" LINE PLUS REVERSED
           ACCEPT NUM-DATA.
    GO TO A00-BEGIN.
```

Figure 17–8: Accepting Data with the ON EXCEPTION Option



```
Enter any number in this range: -999.999 to +999.999




               1234.567-





Valid range is: -999.999 to +999.999
PLEASE try again... press your carriage return key to continue
```

ZK-6095-GE

17.5.3 Protecting Your Screen

You can use the PROTECTED phrase in an ACCEPT statement to limit the number of characters that can be entered. This clause prevents overwriting or deleting parts of the screen.

If you use this phrase, and you try to type past the rightmost position of the input field or delete past the left edge of the input field, the terminal bell sounds and the screen cursor does not move. You can accept the data on the screen by pressing a legal terminator key, or you can delete the data by pressing the DELETE key. If you specify PROTECTED WITH AUTOTERMINATE, the ACCEPT operation terminates when the maximum number of characters has been entered unless a terminator has been entered prior to this point. For more information on legal terminator keys, refer to the CONTROL KEY phrase of the ACCEPT statement in the *VAX COBOL Reference Manual.*

You can also use the REVERSED, BOLD, BLINKING, or UNDERLINED attributes with the PROTECTED phrase. Using these attributes lets you see the size of the input field on the screen before you enter data. The characters you enter also echo the specified attribute.

You can specify the NO BLANK and FILLER phrases with the PROTECTED phrase. The NO BLANK phrase specifies that the protected input field is not to be filled with spaces until after the first character is entered. The FILLER phrase initializes each character position of the input field with the filler character specified.

When you use the FILLER phrase with the NO BLANK phrase, the input field is filled with the filler character only after you have entered the first character.

The PROTECTED SIZE phrase sets the size of the input field on the screen and allows you to change the size of the input field from the size indicated by the PICTURE phrase of the destination item. Example 17–7 and Figure 17–9 show how to use the SIZE phrase with the PROTECTED phrase. When the example specifies SIZE 3, any attempt to enter more than three characters makes the terminal bell ring. When the example specifies SIZE 10, the ACCEPT statement includes the ON EXCEPTION clause to warn you whenever you enter a number that would result in truncation at either end of the assumed decimal point. Figure 17–9 shows such an example in which the operator entered a 10-digit number, exceeding the storage capacity of the data item NUM-DATA on the left side of the assumed decimal point.

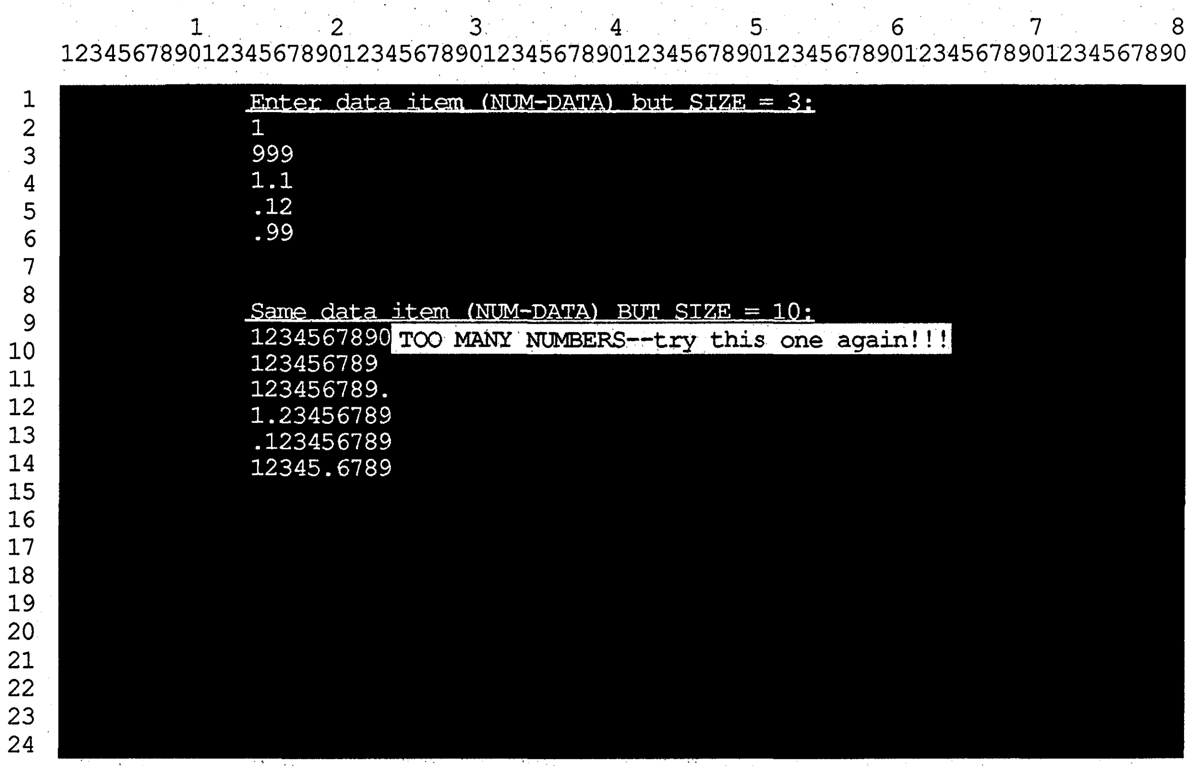
NOTE

> The SIZE phrase controls only the number of characters you can enter; it does not alter any other PICTURE clause requirements. Truncation, space or zero filling, and decimal point alignment occur according to MOVE statement rules only if CONVERSION is specified.

Example 17–7: Using the SIZE Phrase

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROTECT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NUM-DATA  PIC S9(9)V9(9) COMP-3.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Enter data item (NUM-DATA) max SIZE = 3:"
             LINE 1 COLUMN 15
             UNDERLINED
             ERASE SCREEN.
    PERFORM ACCEPT-THREE 5 TIMES.
    DISPLAY "Same data item (NUM-DATA) max SIZE = 10:" LINE PLUS 3
             COLUMN 15
             UNDERLINED.
    PERFORM ACCEPT-TEN 5 TIMES.
    STOP RUN.
ACCEPT-THREE.
    ACCEPT NUM-DATA WITH CONVERSION PROTECTED SIZE 3
           LINE PLUS COLUMN 15.
ACCEPT-TEN.
    ACCEPT NUM-DATA WITH CONVERSION PROTECTED SIZE 10
           LINE PLUS COLUMN 15
           ON EXCEPTION
               DISPLAY "TOO MANY NUMBERS--try this one again!!!"
                        COLUMN PLUS
                        REVERSED
                        GO TO ACCEPT-TEN.
```

Figure 17–9: Screen Display of NUM-DATA Using the PROTECTED Option



```
             Enter data item (NUM-DATA) but SIZE = 3:
             1
             999
             1.1
             .12
             .99


             Same data item (NUM-DATA) BUT SIZE = 10:
             1234567890 TOO MANY NUMBERS--try this one again!!!
             123456789
             123456789.
             1.23456789
             .123456789
             12345.6789
```

ZK-6109-GE

When you do not use the PROTECTED phrase, the amount of data transferred is determined according to the ACCEPT statement rules (see the *VAX COBOL Reference Manual*).

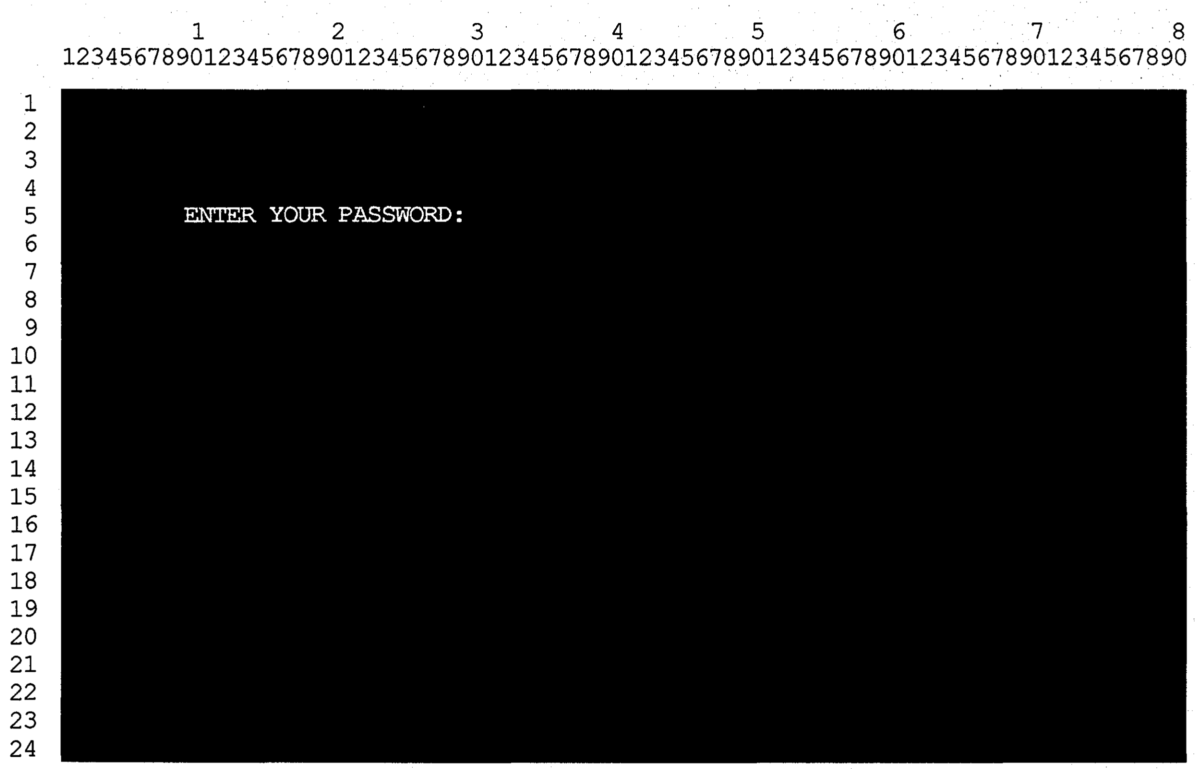
17.5.4 Using NO ECHO with ACCEPT Data

By default, the characters you type at the terminal are displayed on the screen. Example 17–8 and Figure 17–10 show how the NO ECHO phrase prevents the input field from being displayed; thus, the NO ECHO phrase allows you to keep passwords and other information confidential.

Example 17–8: Using NO ECHO

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  NOSHOW.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PASSWORD PIC X(25).
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "ENTER YOUR PASSWORD: " LINE 5 COLUMN 10
            ERASE SCREEN.
    ACCEPT PASSWORD WITH NO ECHO.
    STOP RUN.
```

Figure 17–10: Accepting Data with the NO ECHO Option



```
ENTER YOUR PASSWORD:
```

17.5.5 Assigning Default Values to Data Fields

Use the DEFAULT phrase to assign a value to an ACCEPT data item whenever:

- The program requires a value, and the operator does not have a value for the data item.

- There is a high probability that the default value is identical in most of the records—as where a constant (such as a state's abbreviation) is used in a mailing list.

When you use the DEFAULT phrase, the program executes as if the default value had been typed in when you press RETURN. However, the value is not automatically displayed on the screen.

You can also use the CURRENT VALUE phrase with the DEFAULT phrase to specify that the default input value is the initial value of the ACCEPT destination item.

Example 17–9 and Figure 17–11 show you how to use the DEFAULT phrase to specify default input values (the value must be an alphanumeric data name, a nonnumeric literal, or figurative constant). The example uses the TO-BE-SUPPLIED abbreviations [TBS], ***[TBS]****, and +00.00 as the default values for three data items in the program.

Example 17–9: Using the DEFAULT Phrase

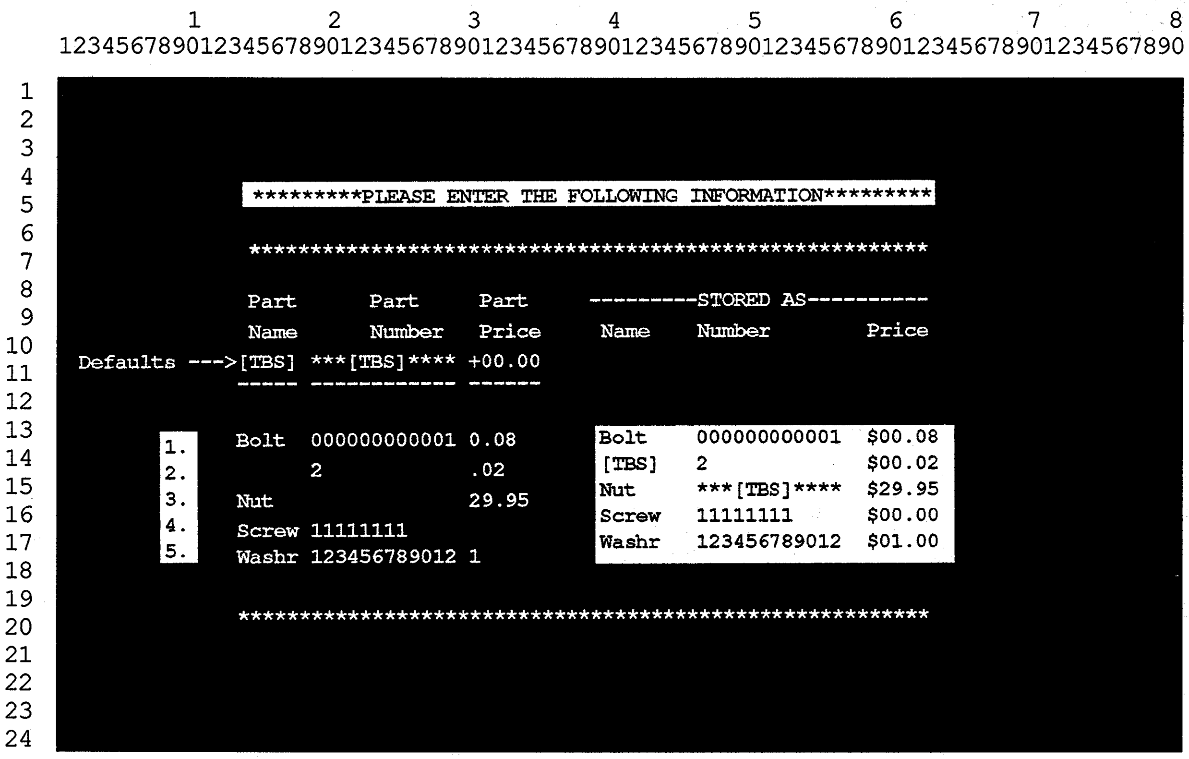
```
IDENTIFICATION DIVISION.
PROGRAM-ID. TRYDEF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA1A       PIC 9(12).
01  NAME1A       PIC XXXXX.
01  PRICEA       PIC S99V99.
01  DATA123.
    02  NAME1B   PIC XXXXX.
    02           PIC XX VALUE SPACES.
    02  DATA1B   PIC XXXXXXXXXXXX.
    02           PIC XXX VALUE SPACES.
    02  PRICEB   PIC $99.99-.
01  NAME-DEFAULT PIC XXXXX VALUE "[TBS]".
01  COL-NUM      PIC 99    VALUE 10.
PROCEDURE DIVISION.
A00-DEFAULT-TEST.
    DISPLAY "*********PLEASE ENTER THE FOLLOWING INFORMATION*********"
             LINE 5 COLUMN 15
             REVERSED BLINKING
             ERASE SCREEN.
    DISPLAY "********************************************************"
             LINE 7 COLUMN 15.
    DISPLAY " Part     Part     Part"
            "   ---------STORED AS-----------"
             LINE 9 COLUMN 15.
    DISPLAY " Name    Number    Price"
            "    Name    Number         Price "
             LINE 10 COLUMN 15.
    DISPLAY "Defaults --->[TBS] ***[TBS]**** +00.00"
             LINE 11 COLUMN 2.
    DISPLAY "----- ------------ ------"
             LINE 12 COLUMN 15.
    DISPLAY "********************************************************"
             LINE 20 COLUMN 15.
    DISPLAY "5. " REVERSED BLINKING LINE 18 COLUMN COL-NUM.
    DISPLAY "4. " REVERSED BLINKING LINE 17 COLUMN COL-NUM.
    DISPLAY "3. " REVERSED BLINKING LINE 16 COLUMN COL-NUM.
    DISPLAY "2. " REVERSED BLINKING LINE 15 COLUMN COL-NUM.
```

(continued on next page)

Example 17–9 (Cont.): Using the DEFAULT Phrase

```
    DISPLAY "1. " REVERSED BLINKING LINE 14 COLUMN COL-NUM.
    DISPLAY " " LINE 13 COLUMN 15.
    PERFORM A05-GET-DATA 5 TIMES.
    DISPLAY " " LINE 22 COLUMN 1.
    STOP RUN.

A05-GET-DATA.
    ACCEPT NAME1A
           PROTECTED
           DEFAULT NAME-DEFAULT
           LINE PLUS COLUMN 15 ERASE TO END OF LINE.
    ACCEPT DATA1A
           PROTECTED
           DEFAULT "***[TBS]****"
           COLUMN 21.
    ACCEPT PRICEA
           PROTECTED
           WITH CONVERSION
           DEFAULT ZERO
           COLUMN 34.
    MOVE NAME1A TO NAME1B.
    MOVE DATA1A TO DATA1B.
    MOVE PRICEA TO PRICEB.
    DISPLAY DATA123 REVERSED COLUMN 44.
```

Figure 17–11: Accepting Data with the DEFAULT Phrase



```
                   *********PLEASE ENTER THE FOLLOWING INFORMATION*********

                   ********************************************************

                   Part       Part       Part     ---------STORED AS----------
                   Name       Number     Price     Name     Number        Price
  Defaults --->[TBS] ***[TBS]**** +00.00
               ----- ------------ ------

          1.   Bolt  000000000001 0.08     Bolt     000000000001  $00.08
          2.         2            .02      [TBS]    2             $00.02
          3.   Nut                29.95    Nut      ***[TBS]****  $29.95
          4.   Screw 11111111              Screw    11111111      $00.00
          5.   Washr 123456789012 1        Washr    123456789012  $01.00

               ********************************************************
```

17.6 Using Keys on Your Terminal to Define Special Program Functions

Use the CONTROL KEY IN phrase of the ACCEPT statement to tailor your screen handling programs to give special meanings to any or all of these keys on your terminal:

- Cursor positioning keys (up arrow, down arrow, left arrow, and right arrow keys)
- Program function keys (PF1, PF2, PF3, and PF4)
- Function keys (F6 to F20)
- Auxiliary keypad keys (if in keypad mode) 0 to 9, minus (–), comma (,), period (.), ENTER, FIND, INSERT HERE, REMOVE, SELECT, PREV SCREEN, NEXT SCREEN

You can use the CONTROL KEY IN phrase to accept data and to terminate it with a control key or to allow a user to press only a control key (for menu applications).

Table 17–3 lists the characters returned to the data name specified in the CONTROL KEY IN phrase.

Table 17–3 is for VT52, VT100, VT200, and VT300 terminals. Depending on your terminal type, certain keys listed in this table are not applicable to your terminal keyboard.

Table 17–3: VAX COBOL Characters Returned for Cursor Positioning, Program Function, Function, and Auxiliary Keypad Keys

		Characters Returned in the Data Name Specified by CONTROL KEY IN	
Key Name	**Keypad Name**	**First**	**Remaining**
Cursor up	up arrow	CSI[1]	A
Cursor down	down arrow	CSI[1]	B
Cursor right	right arrow	CSI[1]	C
Cursor left	left arrow	CSI[1]	D
Program function	PF1	SS3[1]	P
Program function	PF2	SS3[1]	Q
Program function	PF3	SS3[1]	R
Program function	PF4	SS3[1]	S
Auxiliary keypad	left blank	SS3[1]	P
Auxiliary keypad	center blank	SS3[1]	Q
Auxiliary keypad	right blank	SS3[1]	R
Auxiliary keypad	0	SS3[1]	p
Auxiliary keypad	1	SS3[1]	q
Auxiliary keypad	2	SS3[1]	r
Auxiliary keypad	3	SS3[1]	s
Auxiliary keypad	4	SS3[1]	t
Auxiliary keypad	5	SS3[1]	u
Auxiliary keypad	6	SS3[1]	v
Auxiliary keypad	7	SS3[1]	w
Auxiliary keypad	8	SS3[1]	x
Auxiliary keypad	9	SS3[1]	y
Auxiliary keypad	-	SS3[1]	m
Auxiliary keypad	,	SS3[1]	l
Auxiliary keypad	.	SS3[1]	n
Auxiliary keypad	ENTER	SS3[1]	M
Auxiliary keypad	FIND	CSI[1]	1~
Auxiliary keypad	INSERT HERE	CSI[1]	2~
Auxiliary keypad	REMOVE	CSI[1]	3~
Auxiliary keypad	SELECT	CSI[1]	4~
Auxiliary keypad	PREV SCREEN	CSI[1]	5~
Auxiliary keypad	NEXT SCREEN	CSI[1]	6~
TAB	TAB	9	
RETURN	RETURN	13	

[1] The CSI and SS3 characters are shown for your information only. You need not check for their presence because the remaining characters are unique and need no qualification.

(continued on next page)

Table 17–3 (Cont.): VAX COBOL Characters Returned for Cursor Positioning, Program Function, Function, and Auxiliary Keypad Keys

		Characters Returned in the Data Name Specified by CONTROL KEY IN	
Key Name	Keypad Name	First	Remaining
Function key	HOLD SCREEN	Not Available	
Function key	PRINT SCREEN	Not Available	
Function key	SET-UP	Not Available	
Function key	DATA/TALK	Not Available	
Function key	BREAK	Not Available	
Function key	F6	Not Available	
Function key	F7	CSI[1]	18~
Function key	F8	CSI[1]	19~
Function key	F9	CSI[1]	20~
Function key	F10	CSI[1]	21~
Function key	F11 (ESC)	CSI[1]	23~
Function key	F12 (BS)	CSI[1]	24~
Function key	F13 (LF)	CSI[1]	25~
Function key	F14	CSI[1]	26~
Function key	F15 (HELP)	CSI[1]	28~
Function key	F16 (DO)	CSI[1]	29~
Function key	F17	CSI[1]	31~
Function key	F18	CSI[1]	32~
Function key	F19	CSI[1]	33~
Function key	F20	CSI[1]	34~
CTRL/A		1	
CTRL/B		2	
CTRL/C		Not Available	
CTRL/D		4	
CTRL/E		5	
CTRL/F		6	
CTRL/G		7	
CTRL/H		8	
CTRL/I (TAB)		9	
CTRL/J		10	
CTRL/K		11	
CTRL/L		12	
CTRL/M (RETURN)		13	
CTRL/N		14	
CTRL/O		Not Available	

[1] The CSI and SS3 characters are shown for your information only. You need not check for their presence because the remaining characters are unique and need no qualification.

(continued on next page)

Table 17–3 (Cont.): VAX COBOL Characters Returned for Cursor Positioning, Program Function, Function, and Auxiliary Keypad Keys

		Characters Returned in the Data Name Specified by CONTROL KEY IN	
Key Name	**Keypad Name**	**First**	**Remaining**
CTRL/P		16	
CTRL/Q		Not Available	
CTRL/R		18	
CTRL/S		Not Available	
CTRL/T		Depends on SET CONTROL Setting	
CTRL/U		21	
CTRL/V		22	
CTRL/W		23	
CTRL/X		24	
CTRL/Y		Not Available	
CTRL/Z		Results depend on presence or absence of the AT END phrase in the ACCEPT statement	

The definition and value of the CSI and SS3 characters used in Table 17–3 follow:

```
01  SS3X                 PIC 9999 COMP VALUE 143.
01  SS3 REDEFINES SS3X   PIC X.
01  CSIX                 PIC 9999 COMP VALUE 155.
01  CSI REDEFINES CSIX   PIC X.
```

Figure 17–12, Figure 17–13, and Figure 17–14 show the standard keypads for the VT52, VT100, VT200, and VT300 terminals, respectively. The shaded keys correspond to the keypad names in Table 17–3, which lists the characters returned to the application program.

Figure 17–12: VAX COBOL Control Keys on the Standard VT52 Keypad

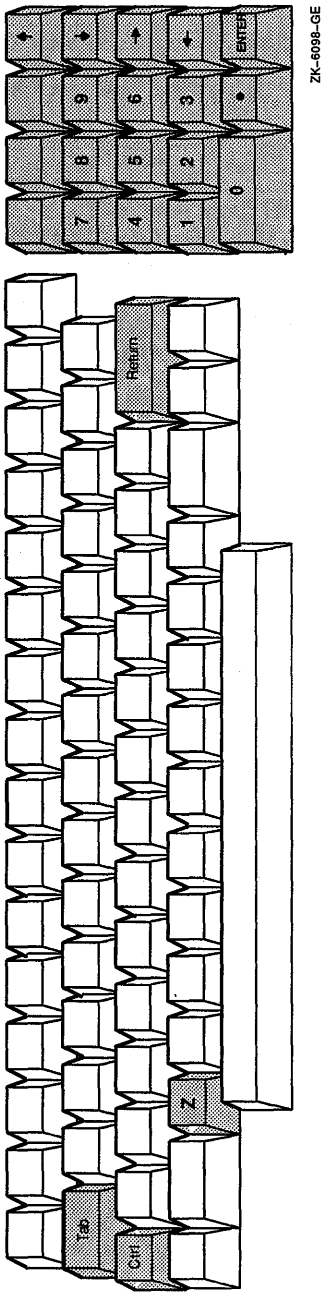


Figure 17–13: VAX COBOL Control Keys on the Standard VT100 Keypad

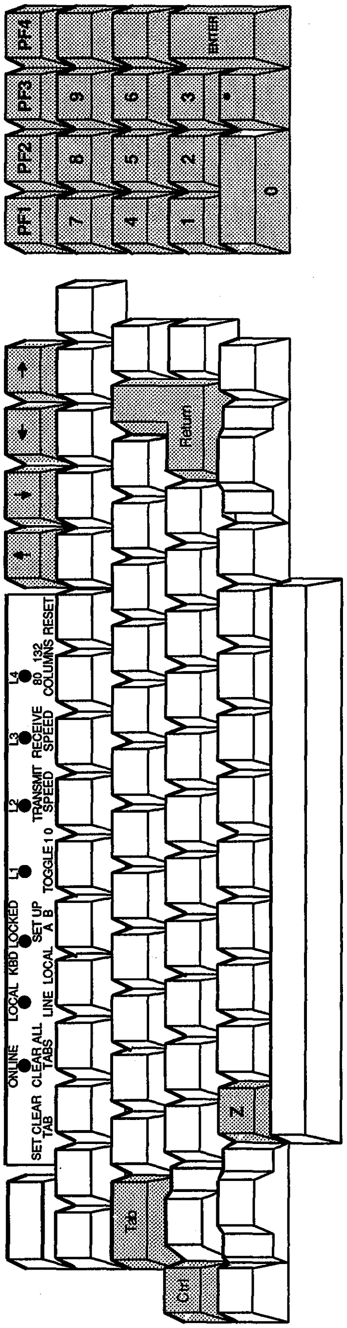
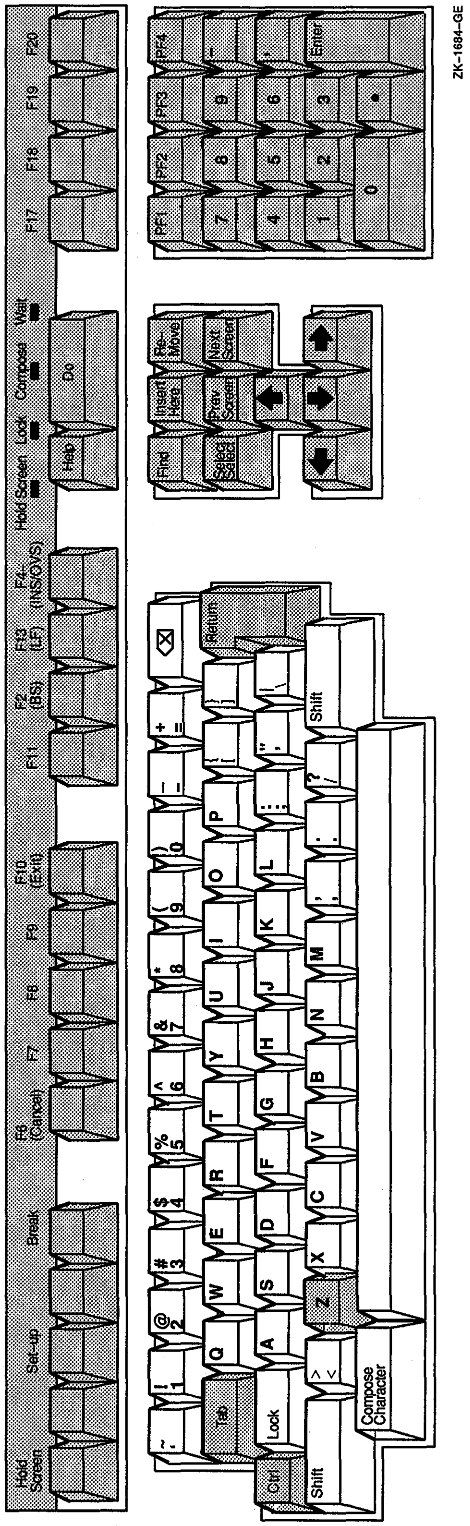


Figure 17–14: VAX COBOL Control Keys on the Standard VT200 and VT300 Keypad



Example 17–10 shows you how to use the CONTROL KEY phrase to handle arrow keys, program function keys, auxiliary keypad keys, CTRL/Z, TAB, and RETURN, using a VT100 terminal on a VMS operating system.

When you use this phrase, you allow program function keys and arrow keys, as well as RETURN and TAB keys, to terminate input. This phrase also permits you to use those keys to move the cursor and to make menu selections without typing any data on the screen.

NOTE

> To activate the auxiliary keypad, your program must execute DISPLAY ESC "=". You must also define ESC as the escape character. Refer to Example 17–10.

In Example 17–10, the terminator key codes are displayed on the screen.

Example 17–10: Using the CONTROL KEY IN Phrase

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL NAMES.
    SYMBOLIC CHARACTERS
        CR-VAL CSI-VAL    CTRL-Z-VAL  SS3-VAL  TAB-VAL  ESC
    ARE 14      156      27          144      10      28.
PROGRAM-ID.  SPECIAL.
DATA DIVISION.
WORKING-STORAGE SECTION.
*
*      The code returned will be the same on VT52,
*      VT100, VT200, and VT300 terminals.
*
01 CONTROL-KEY.
   02  FIRST-CHAR-CONTROL-KEY  PIC X.
        88 CR              VALUE CR-VAL.
        88 CSI             VALUE CSI-VAL.
        88 CTRL-Z          VALUE CTRL-Z-VAL.
        88 SS3             VALUE SS3-VAL.
        88 TAB             VALUE TAB-VAL.
   02  REMAINING-CHAR-CONTROL-KEY PIC XXXX.
        88 UP-ARROW        VALUE "A".
        88 DOWN-ARROW      VALUE "B".
        88 RIGHT-ARROW     VALUE "C".
        88 LEFT-ARROW      VALUE "D".
        88 PF1             VALUE "P".
        88 PF2             VALUE "Q".
        88 PF3             VALUE "R".
        88 PF4             VALUE "S".
        88 AUX0            VALUE "p".
        88 AUX1            VALUE "q".
        88 AUX2            VALUE "r".
        88 AUX3            VALUE "s".
        88 AUX4            VALUE "t".
        88 AUX5            VALUE "u".
        88 AUX6            VALUE "v".
        88 AUX7            VALUE "w".
        88 AUX8            VALUE "x".
```

(continued on next page)

Example 17–10 (Cont.): Using the CONTROL KEY IN Phrase

```
        88 AUX9           VALUE "y".
        88 AUXMINUS       VALUE "m".
        88 AUXCOMMA       VALUE "l".
        88 AUXPERIOD      VALUE "n".
        88 AUXENTER       VALUE "M".

PROCEDURE DIVISION.
P0.
*
* DISPLAY ESC "=" puts you in alternate keypad mode
*
    DISPLAY ESC "=".
    DISPLAY " "  ERASE SCREEN.
P1.

    DISPLAY "Press a directional arrow, PF, RETURN, TAB,  "
             LINE 3 COLUMN 4.
    DISPLAY "or auxiliary keypad key (CTRL/Z stops loop)"
             LINE 4 COLUMN 4.

    ACCEPT CONTROL KEY IN CONTROL-KEY AT END GO TO P2.
    IF CR = DISPLAY "RETURN" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF TAB = DISPLAY "\TAB" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF PF1 DISPLAY "PF1" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF PF2 DISPLAY "PF2" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF PF3 DISPLAY "PF3" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF PF4 DISPLAY "PF4" LINE 10 COLUMN 5 ERASE LINE GO TO P1.
    IF UP-ARROW DISPLAY "UP-ARROW" LINE 10 COLUMN 5 ERASE LINE
       GO TO P1.
    IF DOWN-ARROW DISPLAY "DOWN-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF LEFT-ARROW DISPLAY "LEFT-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF RIGHT-ARROW DISPLAY "RIGHT-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX0 DISPLAY "AUXILIARY KEYPAD 0" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX1 DISPLAY "AUXILIARY KEYPAD 1" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX2 DISPLAY "AUXILIARY KEYPAD 2" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX3 DISPLAY "AUXILIARY KEYPAD 3" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX4 DISPLAY "AUXILIARY KEYPAD 4" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX5 DISPLAY "AUXILIARY KEYPAD 5" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX6 DISPLAY "AUXILIARY KEYPAD 6" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX7 DISPLAY "AUXILIARY KEYPAD 7" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX8 DISPLAY "AUXILIARY KEYPAD 8" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUX9 DISPLAY "AUXILIARY KEYPAD 9" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUXMINUS DISPLAY "AUXILIARY KEYPAD -" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUXCOMMA DISPLAY "AUXILIARY KEYPAD ," LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUXPERIOD DISPLAY "AUXILIARY KEYPAD ." LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
    IF AUXENTER DISPLAY "AUXILIARY KEYPAD ENTER" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
```

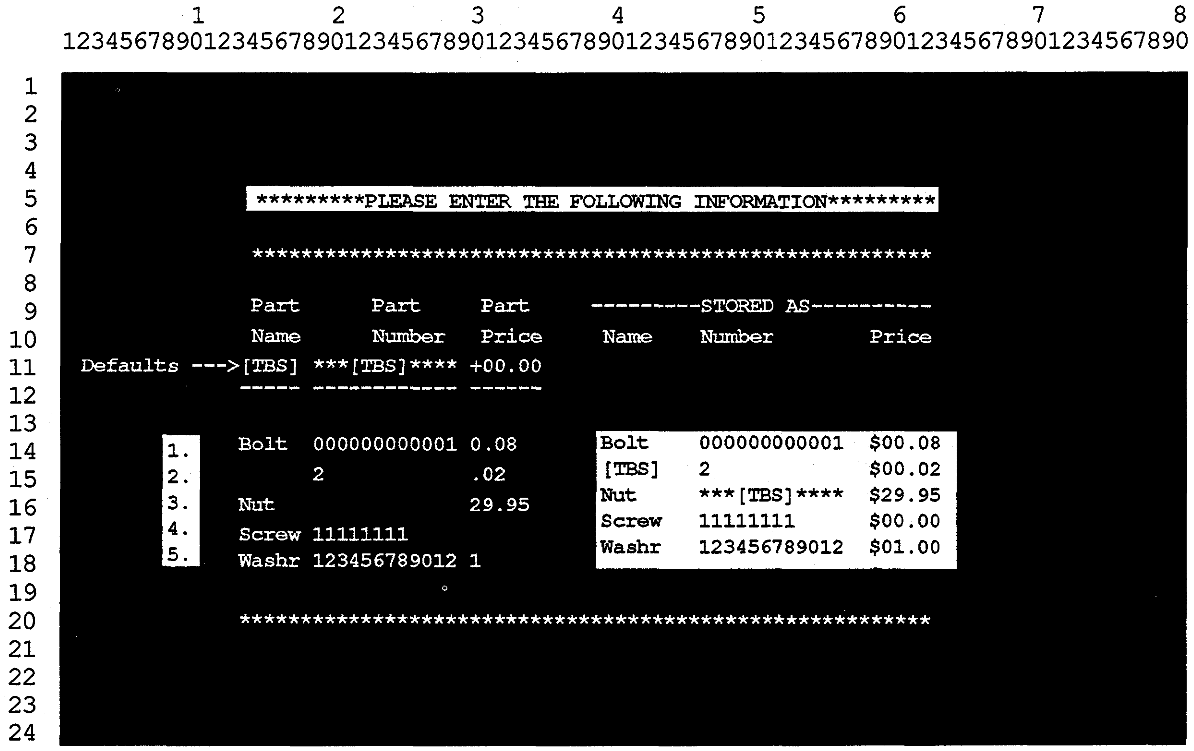
(continued on next page)

Example 17–10 (Cont.): Using the CONTROL KEY IN Phrase

```
    DISPLAY "Not an allowable control key -"
             "press the RETURN key to continue"

            LINE 10 COLUMN 5
            WITH BELL ERASE LINE.
    ACCEPT CONTROL-KEY.
    GO TO P1.
P2.
    DISPLAY "Press the RETURN key to end this job"
            LINE 11 COLUMN 5 ERASE LINE.
    ACCEPT CONTROL KEY IN CONTROL-KEY LINE 12 COLUMN 5 ERASE LINE.
    IF CR NOT = FIRST-CHAR-CONTROL-KEY GO TO P0
       ELSE
          DISPLAY "END OF JOB" LINE 13 COLUMN 35
                   BOLD BLINKING REVERSED BELL
                   ERASE SCREEN
P3.
* DISPLAY ESC ">" WITH NO puts you out of alternate keypad mode
*
     DISPLAY ESC ">" WITH NO.
     STOP RUN.
```

Figure 17–15 shows a sample run of the program in Example 17–10 using the right arrow terminal key.

Figure 17–15: Screen Display of Program SPECIAL



```




              *********PLEASE ENTER THE FOLLOWING INFORMATION*********

              ********************************************************

              Part      Part       Part       ---------STORED AS----------
              Name      Number     Price      Name     Number         Price
  Defaults --->[TBS] ***[TBS]**** +00.00
              ----- ------------ ------


        1.   Bolt  000000000001 0.08        Bolt     000000000001  $00.08
        2.         2            .02         [TBS]    2             $00.02
        3.   Nut                29.95       Nut      ***[TBS]****  $29.95
        4.   Screw 11111111                 Screw    11111111      $00.00
        5.   Washr 123456789012 1           Washr    123456789012  $01.00

             *********************************************************
```

ZK–6097–GE

To expand upon Example 17–10, you can, for example, accept data in addition to specifying the CONTROL KEY phrase. This enables you to accept data and determine what to do next based on the data. You can use the CONTROL KEY phrase to move the cursor around on the screen or take a specific course of action.

17.7 Using the EDITING Phrase

Specifying the EDITING phrase of the ACCEPT statement enables field editing. Table 17–4 briefly describes the keys that the EDITING phrase enables.

Table 17–4: Key Functions for the EDITING Phrase

Key	Function	Description
Left arrow, CTRL/D	Move-left	Moves the cursor one space to the left. If the cursor is at the first character position of the field, the terminal bell rings.
Right arrow, CTRL/F	Move-right	Moves the cursor one space to the right. If the cursor is one space beyond the rightmost character position of the field, the terminal bell rings.

(continued on next page)

Table 17–4 (Cont.): Key Functions for the EDITING Phrase

Key	Function	Description
CTRL/H, F12 (BS)	Beginning-of-field	Positions the cursor to the first character position of the field.
CTRL/E	End-of-field	Moves the cursor one position beyond the rightmost character position in the field.
CTRL/U	Erase-field	Erases the entire field and moves the cursor to the first character position of the field.
CTRL/A, F14	Switch-mode	Switches the editing mode between insert and overstrike.

Example 17–11 shows the sample code that produces the form in Figure 17–16. (The Current Value field is provided for example purposes only.)

Example 17–11: EDITING Phrase Sample Code

```
    .
    .
    .
PROCEDURE DIVISION.
A1000-BEGIN.
    OPEN I-O EMP-FILE.
    .
    .
    .
B1000-MODIFY.
    DISPLAY "MODIFY EMPLOYEE INFORMATION FORM"  ERASE SCREEN
                                            AT LINE 2       COLUMN 10.
    DISPLAY "Enter Employee Number : "  AT LINE PLUS 2 COLUMN 10.

    ACCEPT EMP-KEY
        FROM LINE 4 COLUMN 34
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.
    .
    .
    .
B2000-DISPLAY.

    MOVE EMP-REC TO OUT-REC.

    DISPLAY "Date of Hire : "           AT LINE PLUS 2 COLUMN 10.
    DISPLAY MON-IN                      AT              COLUMN 25.
    DISPLAY "-"                         AT              COLUMN 27.
    DISPLAY DAY-IN                      AT              COLUMN 28.
    DISPLAY "-"                         AT              COLUMN 30.
    DISPLAY YR-IN                       AT              COLUMN 31.
    DISPLAY "Current Value :"           AT COLUMN 40.
    DISPLAY MON-NUM                     AT              COLUMN 56.
    DISPLAY "-"                         AT              COLUMN 58.
    DISPLAY DAY-NUM                     AT              COLUMN 59.
    DISPLAY "-"                         AT              COLUMN 61.
    DISPLAY YR-NUM                      AT              COLUMN 62.
```

(continued on next page)

Example 17–11 (Cont.): EDITING Phrase Sample Code

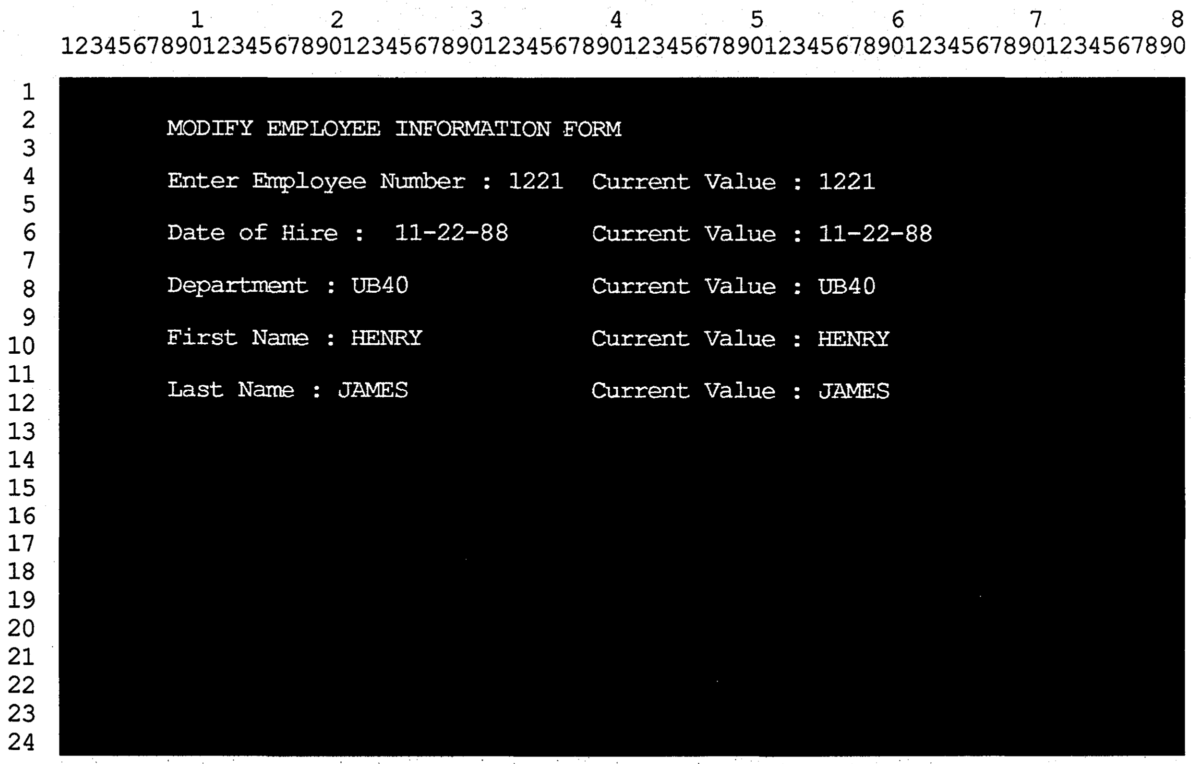
```
    DISPLAY "Department :"              AT LINE PLUS 2  COLUMN 10.
    DISPLAY DEPT-IN                     AT              COLUMN 23.
    DISPLAY "Current Value :"  AT  COLUMN 40.
    DISPLAY DEPT-NUM                    AT              COLUMN PLUS.

    DISPLAY "First Name :"              AT LINE PLUS 2  COLUMN 10.
    DISPLAY F-NAME-IN                   AT              COLUMN 23.
    DISPLAY "Current Value :"  AT              COLUMN 40.
    DISPLAY F-NAME                      AT              COLUMN PLUS.

    DISPLAY "Last Name :"               AT LINE PLUS 2  COLUMN 10.
    DISPLAY L-NAME-IN                   AT              COLUMN 22.
    DISPLAY "Current Value :"  AT              COLUMN 40.
    DISPLAY L-NAME                      AT              COLUMN PLUS.

    ACCEPT MON-NUM
        FROM LINE 6 COLUMN 25
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.

    DISPLAY MON-NUM                    AT LINE 6       COLUMN 56.

    ACCEPT DAY-NUM
        FROM LINE 6 COLUMN 28
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.

    DISPLAY DAY-NUM                    AT LINE 6       COLUMN 59.

    ACCEPT YR-NUM
        FROM LINE 6 COLUMN 31
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.

    DISPLAY YR-NUM                     AT LINE 6       COLUMN 62.

    ACCEPT DEPT-NUM
        FROM LINE 8 COLUMN 23
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.

    DISPLAY DEPT-NUM                   AT LINE 8       COLUMN 56.

    ACCEPT F-NAME
        FROM LINE 10 COLUMN 23
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END

          STOP RUN.

    DISPLAY F-NAME                     AT LINE 10      COLUMN 56.
```

(continued on next page)

Example 17–11 (Cont.): EDITING Phrase Sample Code

```
    ACCEPT L-NAME
        FROM LINE 12 COLUMN 22
        PROTECTED WITH EDITING REVERSED
        DEFAULT IS CURRENT
        AT END
          STOP RUN.

    DISPLAY L-NAME                    AT LINE 12      COLUMN 56.
  .
  .
  .
```

Figure 17–16: Form with ACCEPT WITH EDITING Phrase



```
        MODIFY EMPLOYEE INFORMATION FORM

        Enter Employee Number : 1221  Current Value : 1221

        Date of Hire :  11-22-88      Current Value : 11-22-88

        Department : UB40             Current Value : UB40

        First Name : HENRY            Current Value : HENRY

        Last Name : JAMES             Current Value : JAMES
```

ZK–1516A–GE

Since the ACCEPT statements in Example 17–11 contain EDITING phrases, a person using the form in Figure 17–16 can use any of the keys listed in Table 17–4 for field editing purposes to make corrections or modifications.

Chapter 18

Interprogram Communication

Interprogram communication occurs when VAX COBOL programs communicate with each other or with non-COBOL programs through the CALL statement and external data.

This chapter introduces you to multiple program (COBOL and non-COBOL) run units. The chapter explains and presents examples of how to transfer execution control and data from one program to another in the run unit. Also, the chapter presents information on contained COBOL programs, the Run-Time Library, and system services.

18.1 Multiple COBOL Program Run-Unit Concepts

This section defines a multiple COBOL program run unit, explains the calling procedures, and gives examples of run units.

18.1.1 Definition of a Multiple COBOL Program Run Unit

A multiple COBOL program run unit consists of either of the following:

- One main (driver) program and one or more separately compiled subprograms; each program can have none, one, or more contained (nested) programs.
- One main program with one or more contained (nested) subprograms.

18.1.2 Examples of COBOL Run Units

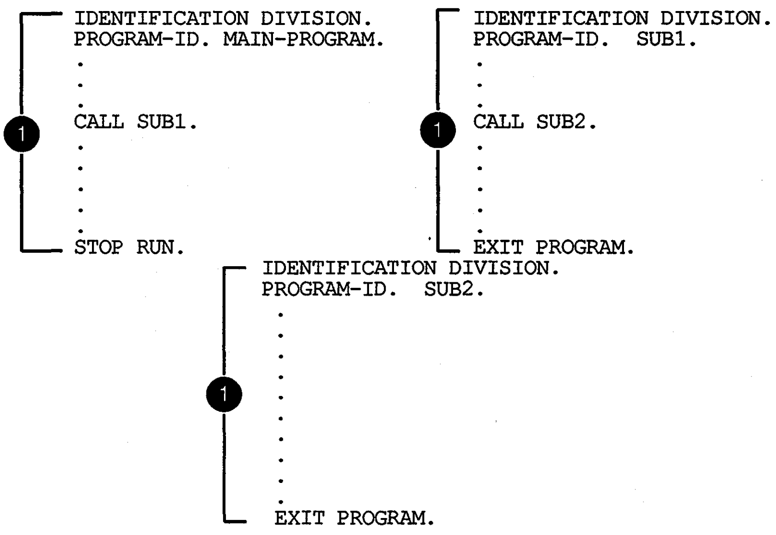
This section provides examples of COBOL run units.

Figure 18–1 shows a run unit with three separately compiled programs, none of which have contained programs.

NOTE

> A separately compiled program has a nesting level number of 1. If this program contains other source programs, it is the outermost containing program. A contained program has a nesting level number greater than 1.

Figure 18–1: Run Unit with Three Separately Compiled Programs



```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROGRAM.
.
.
.
CALL SUB1.
.
.
.
.
.
STOP RUN.
```

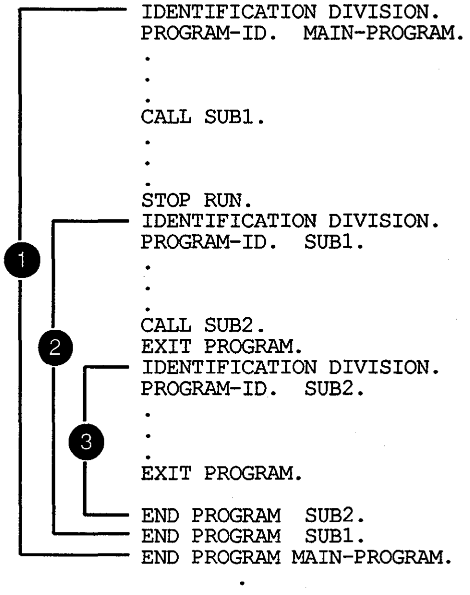
```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB1.
.
.
.
CALL SUB2.
.
.
.
.
.
EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB2.
 .
 .
 .
 .
 .
 .
 .
 .
 .
 .
 EXIT PROGRAM.
```

ZK-1432A-GE

Figure 18–2 shows a run unit with one main program and two contained programs (SUB1 is a directly contained program of MAIN-PROGRAM; SUB2 is an indirectly contained program of MAIN-PROGRAM).

Figure 18–2: Run Unit with a Main Program and Two Contained Programs

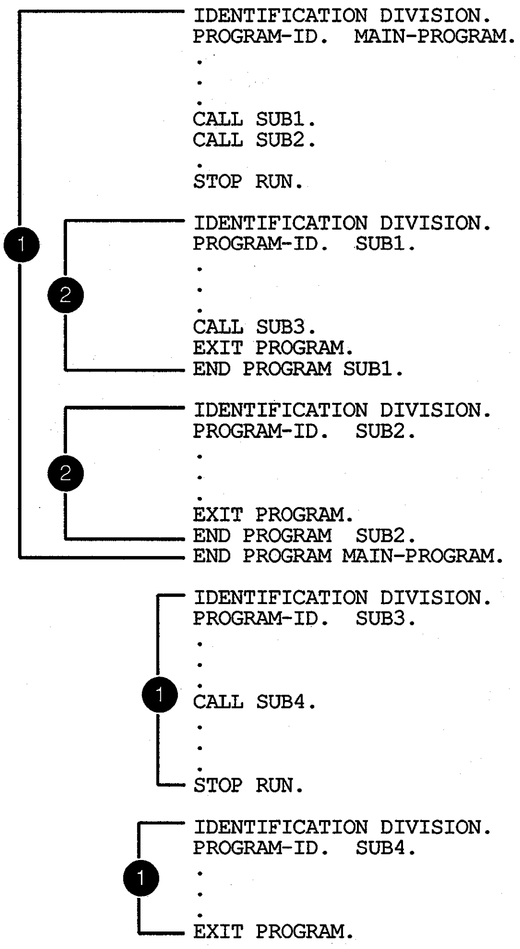


```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAIN-PROGRAM.
.
.
.
CALL SUB1.
.
.
.
STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB1.
.
.
.
CALL SUB2.
EXIT PROGRAM.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB2.
.
.
.
EXIT PROGRAM.

END PROGRAM  SUB2.
END PROGRAM  SUB1.
END PROGRAM MAIN-PROGRAM.
        .
```

ZK-1433A-GE

Figure 18–3 shows a run unit with three separately compiled programs, one of which, MAIN-PROGRAM, has two directly contained programs (SUB1, SUB2).

Figure 18–3: Run Unit with Three Separately Compiled Programs and Two Contained Programs



```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAIN-PROGRAM.
.
.
.
CALL SUB1.
CALL SUB2.
.
STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB1.
.
.
.
CALL SUB3.
EXIT PROGRAM.
END PROGRAM SUB1.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB2.
.
.
.
EXIT PROGRAM.
END PROGRAM  SUB2.
END PROGRAM MAIN-PROGRAM.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB3.
.
.
.
CALL SUB4.
.
.
.
STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB4.
.
.
.
EXIT PROGRAM.
```

ZK-1431A-GE

18.1.3 Calling Procedures

A COBOL main (driver) program calls subprograms (contained or separately compiled). Image execution begins and ends in the main program's Procedure Division. The program contains one or more CALL statements and is a calling program.

A COBOL subprogram is called by a main program or another subprogram. The subprogram contains none, one, or several CALL statements. If a subprogram contains a CALL statement, it is both a calling and a called program. If the subprogram does not contain a CALL statement, it is a called program only.

18.2 COBOL Program Attributes

Any VAX COBOL program can possess the INITIAL clause in the PROGRAM-ID paragraph. Data and files in a COBOL program can have the EXTERNAL clause.

18.2.1 The INITIAL Clause

A COBOL program with an INITIAL clause is initialized, whenever the program is called, to the same state as when that program was first called in the run unit.

During this initialization process, all internal program data whose description contains a VALUE clause is initialized to that defined value. Any item whose description does not include a VALUE clause is initialized to an undefined value.

When an INITIAL clause is present and when the program is called, an implicit CLOSE statement executes for all files in the open mode associated with internal file connectors.

When an INITIAL clause is not present, the status of the files and internal program data are the same as when the called program was exited.

The initial attribute is attained by specifying the INITIAL clause in the program's PROGRAM-ID paragraph. For example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  TEST-PROG  INITIAL.
```

18.2.2 The EXTERNAL Clause

Storage of data can be external or internal to the program in which the data is declared. A file connector can also be external or internal to the program in which it is defined.

External data or files can be referenced by every program in a run unit that describes that data or those files as external.

The EXTERNAL clause indicates that data or a file is external. This clause is specified only in File Description entries in the FILE SECTION or in Record Description entries in the WORKING-STORAGE Section. The EXTERNAL clause is one method of sharing data between programs. For example, in the following Working-Storage Section entry, the data items in RECORD-1 are available to any program in the image that also describes RECORD-1 and its data items as EXTERNAL:

```
01  RECORD-1 EXTERNAL.
    03  ITEMA  PIC X.
    03  ITEMB  PIC X(20).
    03  ITEMC  PIC 99.
```

NOTE

EXTERNAL files and data must be described identically in all programs in which they are defined.

18.3 Transferring Execution Control

This section explains how transfer of control occurs and how access of program data takes place between VAX COBOL programs in a run unit. Contained COBOL programs have additional communication mechanisms that are explained in Section 18.5.

You control a multiple program run-unit sequence by executing the following:

- A controlling CALL statement in the calling program (main or subprogram)
- An EXIT PROGRAM statement in the called subprogram

18.3.1 The CALL Statement

A CALL statement transfers the run unit's execution control from the calling program to the beginning of the called subprogram's Procedure Division. See the *VAX COBOL Reference Manual* for the CALL format.

The first time the called subprogram gains execution control, its state is that of a fresh copy of the program. Thereafter, each time it is called its state is the same as the last exit from that program, except when: (1) the called program has the INITIAL clause, or (2) the calling program cancels the called program.

NOTE

A program cannot cancel itself nor can any program cancel the program that called it.

18.3.2 The EXIT PROGRAM Statement

To return execution control to the calling program, the called subprogram executes an EXIT PROGRAM statement.

You can include more than one EXIT PROGRAM statement in a subprogram. However, if it appears in a consecutive sequence of imperative statements, the EXIT PROGRAM statement must appear as the last statement of the sequence. For example:

```
IF A = B DISPLAY "A equals B", EXIT PROGRAM.

READ INPUT-FILE      AT END DISPLAY "End of input file"
                     PERFORM END-OF-FILE-ROUTINE
                     EXIT PROGRAM.
```

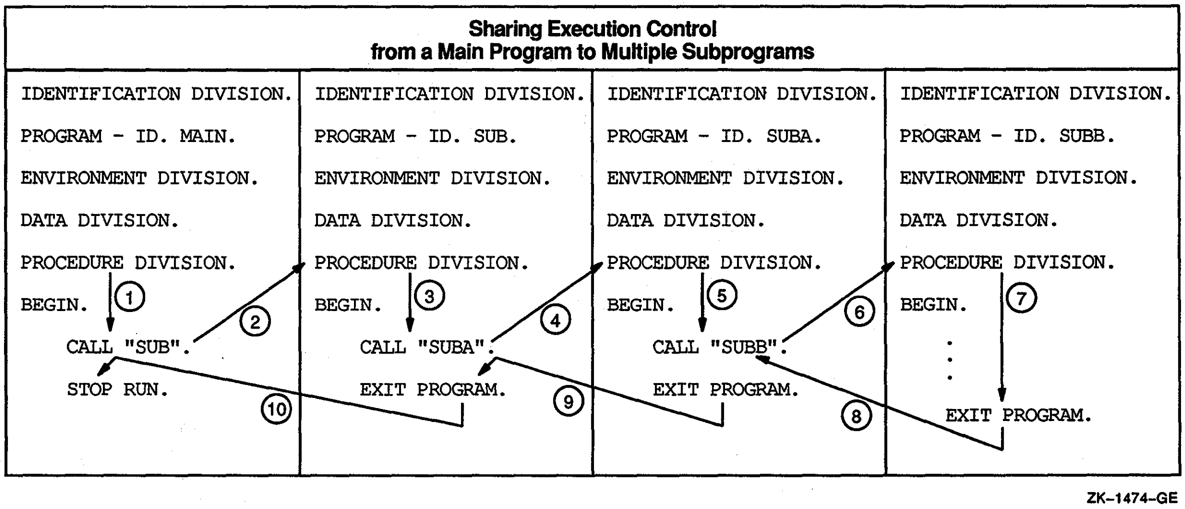
If you do not include an EXIT PROGRAM statement in a subprogram, the compiler generates one at the end of the program.

On executing an EXIT PROGRAM statement in a called subprogram, control returns to the statement following the calling program's CALL statement, or the first imperative statement in a NOT ON EXCEPTION clause specified for that CALL statement.

On executing an EXIT PROGRAM statement in a main program, the EXIT PROGRAM is ignored and control continues with the next statement.

Figure 18–4 shows how control is passed between programs.

Figure 18–4: Sharing Execution Control from a Main Program to Multiple Subprograms



Sharing Execution Control from a Main Program to Multiple Subprograms

```
IDENTIFICATION DIVISION.
PROGRAM - ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
   CALL "SUB".
   STOP RUN.
```

```
IDENTIFICATION DIVISION.
PROGRAM - ID. SUB.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
   CALL "SUBA".
   EXIT PROGRAM.
```

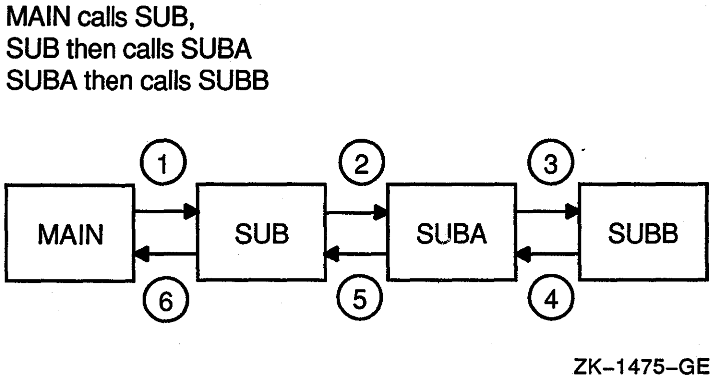
```
IDENTIFICATION DIVISION.
PROGRAM - ID. SUBA.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
   CALL "SUBB".
   EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM - ID. SUBB.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
   EXIT PROGRAM.
```

18.3.3 Nesting CALL Statements

A called subprogram can itself transfer execution control after receiving control from a main program or another subprogram. This technique is known as CALL statement nesting. For example, Figure 18–5 shows a nested image that executes a series of three CALL statements from three separate programs.

Figure 18–5: CALL Statement Nesting



MAIN calls SUB,
SUB then calls SUBA
SUBA then calls SUBB

The MAIN, SUB1, and SUB2 programs in Example 18–1 illustrate their execution sequence by displaying a series of 12 messages on the default output device. Image execution begins in MAIN with message number 1. It ends in MAIN with message number 12. The image's message sequence is shown following the program listings.

Example 18–1: Execution Sequence

```
IDENTIFICATION DIVISION.
*
* MAIN is a calling program only
*
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
    DISPLAY " 1. MAIN has the first execution control.          ".
    DISPLAY " 2. MAIN transfers execution control to SUB1       ".
    DISPLAY "         upon executing the following CALL.        ".
    CALL "SUB1"
    DISPLAY "11. MAIN has the last  execution control.          ".
    DISPLAY "12. MAIN terminates the entire image upon          ".
    DISPLAY "         execution of the STOP RUN statement.      ".
    STOP RUN.
IDENTIFICATION DIVISION.
*
* SUB1 is both a called and calling subprogram
*
*      It is called by MAIN
*
*      It then calls SUB2
PROGRAM-ID. SUB1.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
    DISPLAY " 3.      This is the entry point to SUB1.          ".
    DISPLAY " 4. SUB1 now has execution control.                ".
    DISPLAY " 5. SUB1 transfers execution control to SUB2.      ".
    CALL "SUB2"
    DISPLAY " 9. SUB1 regains execution control                 ".
    DISPLAY "10.      after executing the following             ".
    DISPLAY "         EXIT PROGRAM statement.                   ".
    EXIT PROGRAM.
IDENTIFICATION DIVISION.
*
* SUB2 is called subprogram only
*
*      It is called by SUB1
*
PROGRAM-ID. SUB2.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
    DISPLAY " 6.      This is the entry point to SUB2.          ".
    DISPLAY " 7. SUB2 now has execution control.                ".
    DISPLAY " 8. SUB2 returns execution control to SUB1         ".
    DISPLAY "         after executing the following             ".
    DISPLAY "         EXIT PROGRAM statement.                   ".
    EXIT PROGRAM.
```

Example 18–2 shows the messages printed to the default output device when the programs in Example 18–1 are run.

Example 18–2: Sequence of Messages Displayed

```
 1. MAIN has the first execution control.
 2. MAIN transfers execution control to SUB1
         upon executing the following CALL.
 3.      This is the entry point to SUB1.
 4. SUB1 now has execution control.
 5. SUB1 transfers execution control to SUB2.
 6.      This is the entry point to SUB2.
 7. SUB2 now has execution control.
 8. SUB2 returns execution control to SUB1
         after executing the following
         EXIT PROGRAM statement.
 9. SUB1 regains execution control
10.      after executing the following
         EXIT PROGRAM statement.
11. MAIN has the last  execution control.
12. MAIN terminates the entire image upon
         execution of the STOP RUN statement.
```

18.4 Accessing Another Program's Data Division

In a multiple COBOL program run unit, a called subprogram can access its calling program's Data Division. However, the calling program controls how much of it will be accessible to the called subprogram through:

- The USING phrase in both the CALL statement and the Procedure Division header
- The Linkage Section
- The EXTERNAL clause (see Section 18.2.2)
- The GLOBAL clause (see Section 18.5.2)

18.4.1 The USING Phrase

To access a calling program's Data Division, use a CALL statement in the calling program and a Procedure Division USING phrase in the called program. The USING phrases of both the CALL statement and the Procedure Division header must contain an equal number of data names. (See Figure 18–6.)

The CALL statement can make data available to the called program by five argument-passing mechanisms:

- REFERENCE—The address of (pointer to) the argument (arg) is passed to the calling program. This is the default mechanism.
- CONTENT—The address of a copy of the contents of arg is passed to the called program. Note that since a copy of the data is passed, the called program cannot change the original calling program data.
- VALUE—The value of arg is passed to the called program. If arg is a data name, its description in the Data Division can be as follows: (a) COMP usage with no scaling positions (the PICTURE clause can specify no more than nine digits) and (b) COMP-1 usage.

- DESCRIPTOR—The address of (pointer to) the data item's descriptor is passed to the called program.
- OMITTED—A value equivalent to BY VALUE 0 is passed to the called program. Note that OMITTED does not change the default mechanism.

NOTE

> A called COBOL subprogram must have arguments passed to it using BY REFERENCE, which is the default, or BY CONTENT. BY VALUE, OMITTED, and BY DESCRIPTOR are DIGITAL extensions and will not work as expected if passed to a COBOL program. These argument-passing mechanisms are necessary when calling Run-Time Library Routines and system service routines as described in Section 18.7.

The mechanism for each argument in the CALL statement USING phrase must be the same as the mechanism for each argument in the called program's parameter list.

If the BY REFERENCE phrase is either specified or implied for a parameter, the called program references the same storage area for the data item as the calling program. This mechanism ensures that the contents of the parameter in the calling program are always identical to the contents of the parameter in the called program.

If the BY CONTENT phrase is either specified or implied for a parameter, only the initial value of the parameter is made available to the called program. The called program references a separate storage area for the data item. This mechanism ensures that the called program cannot change the contents of the parameter in the calling program's USING phrase. However, the called program can change the value of the data item referenced by the corresponding data name in the called program's Procedure Division header.

Once a mechanism is established in a CALL statement, successive arguments default to the established mechanism until a new mechanism is used. For example:

```
CALL "TESTPRO" USING ITEM-A
  BY DESCRIPTOR ITEM-B
```

Note that ITEM-A is passed using the BY REFERENCE phrase and that ITEM-B is passed using the BY DESCRIPTOR phrase.

If the OMITTED phrase is specified for a parameter, the established call mechanism does not change.

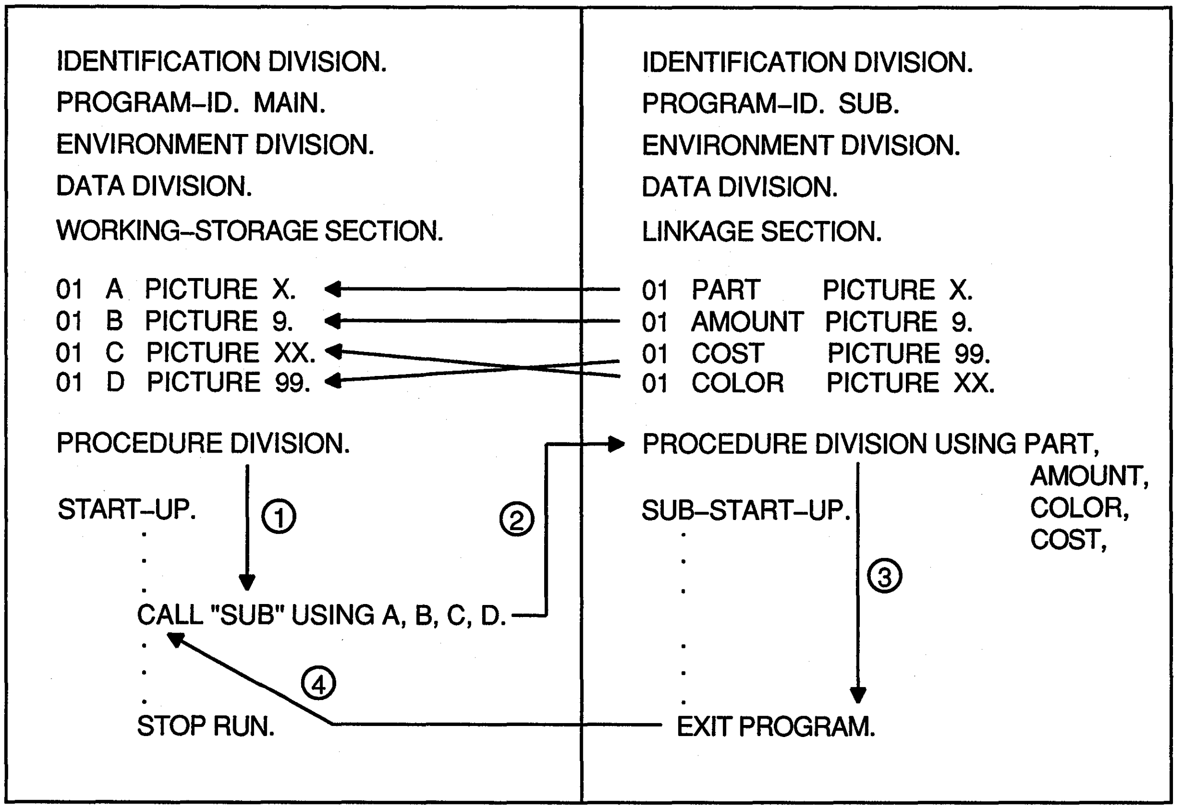
One other mechanism of the CALL verb is the ability to use a GIVING phrase in the CALL statement. This allows the subprogram to return a value through the data item in the GIVING phrase. For example:

```
CALL "TESTPRO" USING ITEMA ITEMB
    GIVING ITEMC.
```

Note that the GIVING result (ITEMC) must be an elementary integer numeric data item with COMP, COMP-1, or COMP-2 usage and no scaling positions.

The order in which USING identifiers appear in both calling and called programs determines the correspondence of single sets of data available to the called subprogram. The correspondence is by position, not by name.

Figure 18–6: Accessing Another Program's Data Division



```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

01  A  PICTURE X.
01  B  PICTURE 9.
01  C  PICTURE XX.
01  D  PICTURE 99.

PROCEDURE DIVISION.

START-UP.
    .
    .
    .
    CALL "SUB" USING A, B, C, D.
    .
    .
    .
    STOP RUN.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.

01  PART      PICTURE X.
01  AMOUNT    PICTURE 9.
01  COST      PICTURE 99.
01  COLOR     PICTURE XX.

PROCEDURE DIVISION USING PART,
                         AMOUNT,
SUB-START-UP.            COLOR,
    .                    COST,
    .
    .

    .
    .
    .
    EXIT PROGRAM.
```

ZK-1731-84

In Figure 18–6, when execution control transfers to SUB, it can access the four data items in the calling program by referring to the data names in its Procedure Division USING phrase. For example, the data names correspond as follows:

MAIN Calling Program data-name	SUB Called Subprogram data-name
A	PART
B	AMOUNT
C	COLOR
D	COST

18.4.1.1 The Linkage Section

You must define each data name from the Procedure Division header's USING data name list in the called subprogram's Linkage Section. For example:

```
LINKAGE SECTION.

01 PART     PICTURE...
01 AMOUNT   PICTURE...
01 INVOICE  PICTURE...
01 COLOR    PICTURE...
01 COST     PICTURE...

PROCEDURE DIVISION USING PART, AMOUNT, COLOR, COST.
```

Of those items you define in the Linkage Section, only those in the calling program's Procedure Division header's USING phrase are accessible to the called program. In the previous example, INVOICE is not accessible from the called program.

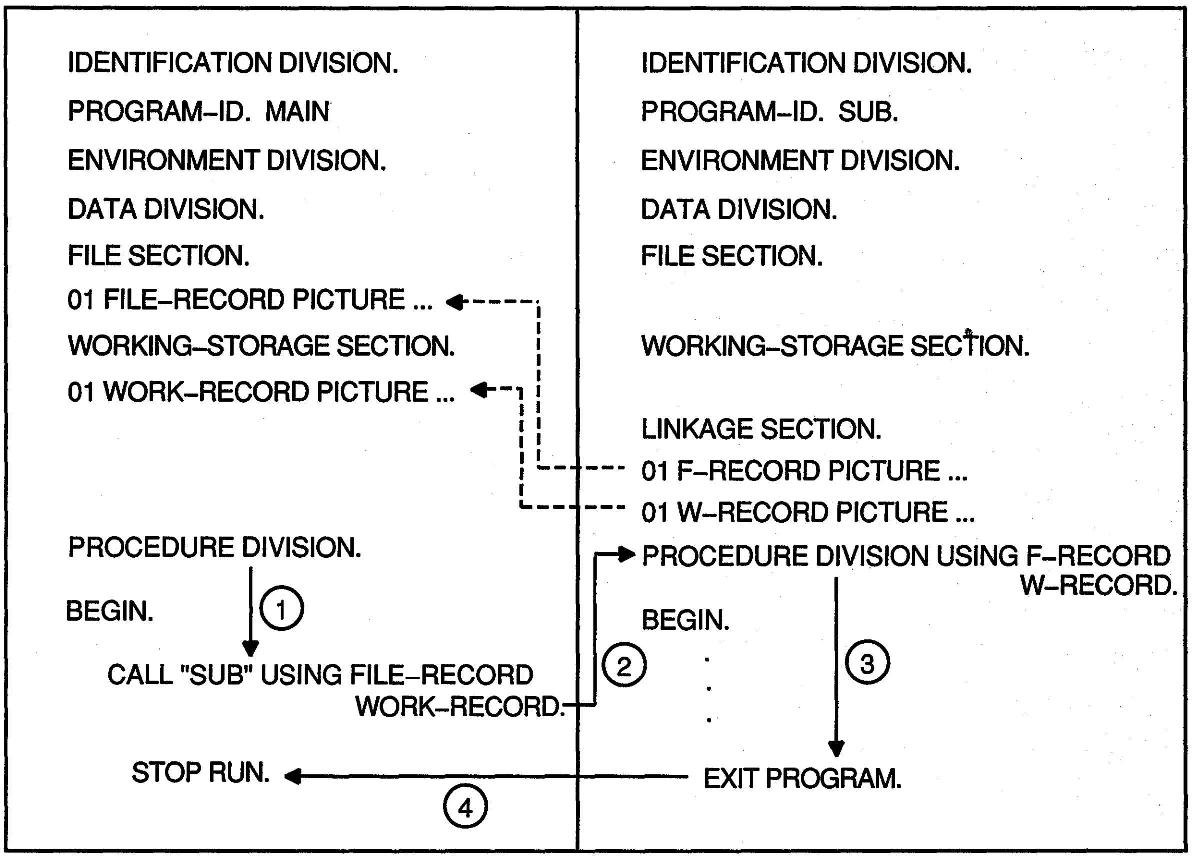
When a subprogram references a data name from the Procedure Division header's USING data name list, the subprogram processes it according to the definition in its Linkage Section.

A called program's Procedure Division can reference data items in its Linkage Section only if it references one of the following:

- Any data item in the Procedure Division USING data-item-list
- A data item that is subordinate to a Linkage Section data item in the Procedure Division USING data-item-list
- Any other association with a data item in the Procedure Division USING data-item-list; for example, index-name, redefinition, and so on.

In Figure 18–7, SUB is called by MAIN. Because MAIN includes FILE-RECORD and WORK-RECORD in its CALL "SUB" USING statement, SUB can reference these data items just as if they were in its own Data Division. However, SUB accesses these two data items with its own data names, F-RECORD and W-RECORD.

Figure 18–7: Defining Data Names in Linkage Section



```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.
01 FILE-RECORD PICTURE ...
WORKING-STORAGE SECTION.
01 WORK-RECORD PICTURE ...

PROCEDURE DIVISION.
BEGIN.
   CALL "SUB" USING FILE-RECORD
                    WORK-RECORD.
  STOP RUN.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

LINKAGE SECTION.
01 F-RECORD PICTURE ...
01 W-RECORD PICTURE ...
PROCEDURE DIVISION USING F-RECORD
                         W-RECORD.
BEGIN.
  .
  .
  .
     EXIT PROGRAM.
```

ZK-1477-GE

18.5 Communicating with Contained COBOL Programs

A contained COBOL program is a subprogram nested in another COBOL program (the containing program). The complete source of the contained program is found within the containing program. A contained program can also be a containing program.

A VAX COBOL containing/contained program provides you with program and data attributes that noncontained COBOL programs do not have. These attributes, described in the next several sections, often allow you to more easily share and more conveniently access COBOL data items and other program resources.

This VAX COBOL programming and data structuring capability encourages modular programming. In modular programming, you divide the solution of a large data processing problem into individual parts (the contained programs) that can be developed relatively independently.

Consequently, the use of this VAX COBOL containing/contained block structure as a modular programming design can increase program efficiency and assist in program modification and maintainability.

The contained program uses all calling procedures described in Section 18.3 and Section 18.4. However, when a contained program includes the COMMON clause (a program attribute) and the GLOBAL clause (a data and file trait), additional rules apply.

18.5.1 The COMMON Clause

The COMMON clause is a program attribute that can be applied to a directly contained program. The COMMON clause is a means of overriding normal scoping rules for program names, which state that a program that does not possess the common attribute and that is directly contained within another program can be referenced only by statements included in that containing program. (Refer to the *VAX COBOL Reference Manual* for Scope of Names rules.)

A program that does possess the common attribute can be referenced by statements included in that containing program and by any programs directly or indirectly contained in that containing program, except the program possessing the common attribute and any programs contained within it.

The common attribute is attained by specifying the COMMON clause in a program's Identification Division.

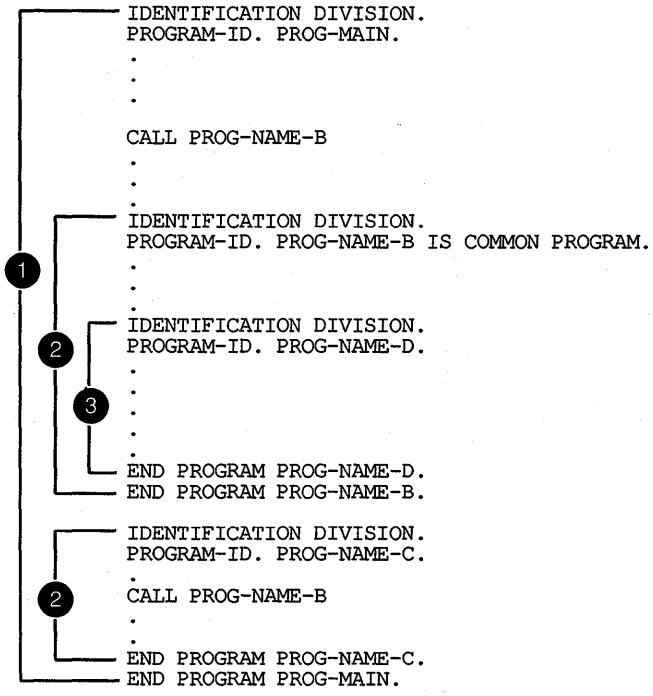
Figure 18–8 shows a run unit that has a COBOL program (PROG-MAIN) with three contained programs; one of which has the COMMON clause. The example indicates which programs can call the common program.

PROG-NAME-B and PROG-NAME-C are directly contained in PROG-MAIN; PROG-D is indirectly contained in PROG-MAIN.

PROG-MAIN can call PROG-NAME-B because PROG-MAIN directly contains PROG-NAME-B. PROG-NAME-B can call PROG-NAME-D because PROG-NAME-B directly contains PROG-NAME-D.

PROG-NAME-C can call PROG-NAME-B because: (1) PROG-NAME-C is not contained in PROG-NAME-B, (2) PROG-NAME-B has the common attribute, and (3) PROG-NAME-C is contained by PROG-MAIN. However, PROG-NAME-D cannot call PROG-NAME-B because PROG-NAME-D is contained within PROG-NAME-B.

Figure 18–8: Using the COMMON Clause



```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-MAIN.
.
.
.
CALL PROG-NAME-B
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-B IS COMMON PROGRAM.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-D.
.
.
.
.
.
END PROGRAM PROG-NAME-D.
END PROGRAM PROG-NAME-B.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-NAME-C.
.
CALL PROG-NAME-B
.
.
END PROGRAM PROG-NAME-C.
END PROGRAM PROG-MAIN.
```

ZK-1430A-GE

18.5.2 Defining and Using the GLOBAL Clause

Data and files can be described as either global or local. A local name can be referenced only by the program that declares it. A global name is declared in only one program but can be referenced by both that program and any program contained in the program that declares the global name.

Some names are always global, other names are always local, and some names are either local or global depending on specifications in the program that declares the names. (See Scope of Names rules in the *VAX COBOL Reference Manual.*)

18.5.2.1 Sharing Data

A data name is global if the GLOBAL clause is specified in the Data Description entry by which the data name is declared or in another entry to which that Data Description entry is subordinate. If a program is contained within another program, both programs may reference data possessing the global attribute. The following example shows the Working-Storage Section of a containing program MAIN. Any contained program in MAIN, as well as program MAIN, can reference that data (unless the contained program declares other data with the same name).

```
WORKING-STORAGE SECTION.
01     CUSTOMER-FILE-STATUS    PIC XX       GLOBAL.
01     REPLY                   PIC X(10)    GLOBAL.
01     ACC-NUM                 PIC 9(18)    GLOBAL.
```

18.5.2.2 Sharing Files

A file connector is global if the GLOBAL clause is specified in the File Description entry for that file connector. If a program is contained within another program, both programs may reference a file possessing the global attribute. The following example shows a file (CUSTOMER-FILE) with the GLOBAL clause in a containing program MAIN. Any contained program in MAIN, as well as program MAIN, can reference that file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   MAIN.
.
.
.
DATA DIVISION.
FILE SECTION.
FD    CUSTOMER-FILE
      GLOBAL
.
.
.
```

Any special registers associated with a GLOBAL file are also global.

18.5.2.3 Sharing Database Resources

The following user-defined words are always implicitly defined as global in the Subschema Section:

- Data name
- Keeplist name
- Key name
- Realm name
- Record name
- Set name

In addition, database special registers, DB-CONDITION, DB-CURRENT-RECORD-NAME, DB-UWA, DB-KEY, and DB-CURRENT-RECORD-ID are always global.

The program defining the Subschema Section and any program it contains can reference these user-defined words and special registers.

18.5.2.4 Sharing Other Resources

Condition names, record names and report names can also have the global attribute. Any program directly or indirectly contained within the program declaring the global name can reference the global name.

A condition name declared in a Data Description entry is global if the condition-variable it is associated with is a global name.

A record name is global if the GLOBAL clause is specified in the Record Description entry by which the record name is declared, or in the case of Record Description entries in the File Section, if the GLOBAL clause is specified in the File Description entry for the file name associated with the Record Description entry.

A report name is global if the GLOBAL clause is specified in the Report Description entry by which the report name is declared. In addition, if the Report Description entry contains the GLOBAL clause, the special registers LINE-COUNTER and PAGE-COUNTER are global names.

Because you cannot specify a Configuration Section for a program contained within another program, the following types of user-defined words are always global; that is, they are always accessible from within a contained program:

- Alphabet name
- Class name
- Condition name
- Mnemonic name
- Symbolic character name

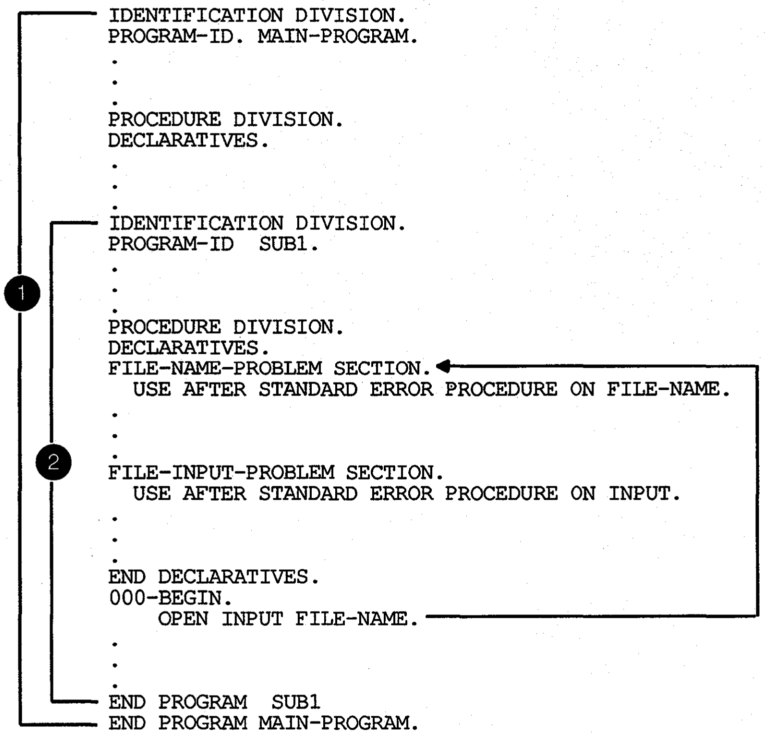
These user-defined words can be referenced by statements and entries either in the program that contains the Configuration Section or any program contained in that program.

18.5.3 Sharing USE Procedures

The USE statement specifies declarative procedures to handle input/output errors. It also can specify procedures to be executed before the program processes a specific report group.

More than one USE AFTER EXCEPTION procedure in any given program can apply to an input/output operation when there is one procedure for file name and another for the applicable open mode. In this case, only the procedure for file name executes. Figure 18–9 shows that FILE-NAME-PROBLEM SECTION executes.

Figure 18–9: Sharing USE Procedures



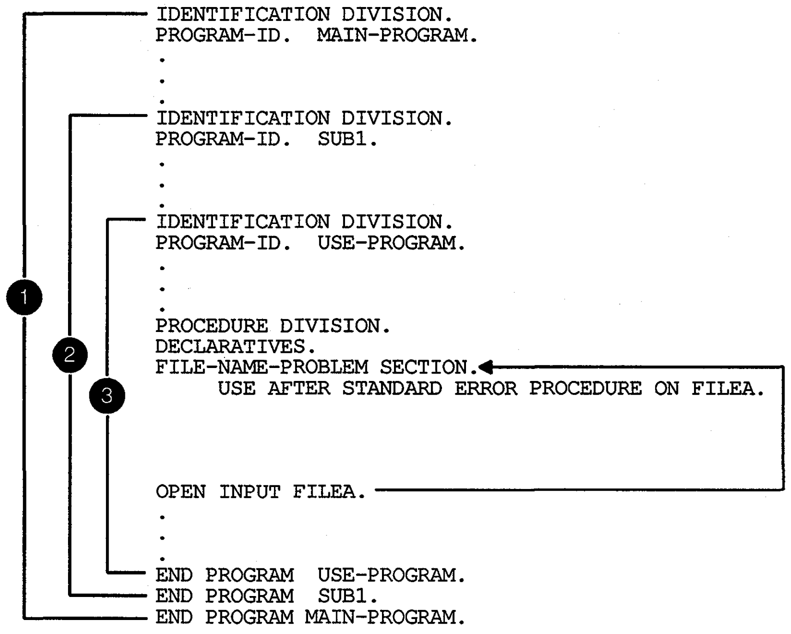
```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROGRAM.
.
.
.
PROCEDURE DIVISION.
DECLARATIVES.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID  SUB1.
.
.
.
PROCEDURE DIVISION.
DECLARATIVES.
FILE-NAME-PROBLEM SECTION.
  USE AFTER STANDARD ERROR PROCEDURE ON FILE-NAME.
.
.
.
FILE-INPUT-PROBLEM SECTION.
  USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
.
.
.
END DECLARATIVES.
000-BEGIN.
    OPEN INPUT FILE-NAME.
.
.
.
END PROGRAM  SUB1
END PROGRAM MAIN-PROGRAM.
```

ZK-1429A-GE

At run time, two special precedence rules apply for the selection of a declarative when programs are contained in other programs. In applying these rules, only the first qualifying declarative is selected for execution. The order of precedence for the selection of a declarative follows:

1. RULE 1—The declarative that executes first is the declarative within the program containing the statement that caused the qualifying condition. In Figure 18–10, FILE-NAME-PROBLEM procedure executes.

Figure 18–10: Executing Declaratives with Contained Programs (Rule 1)



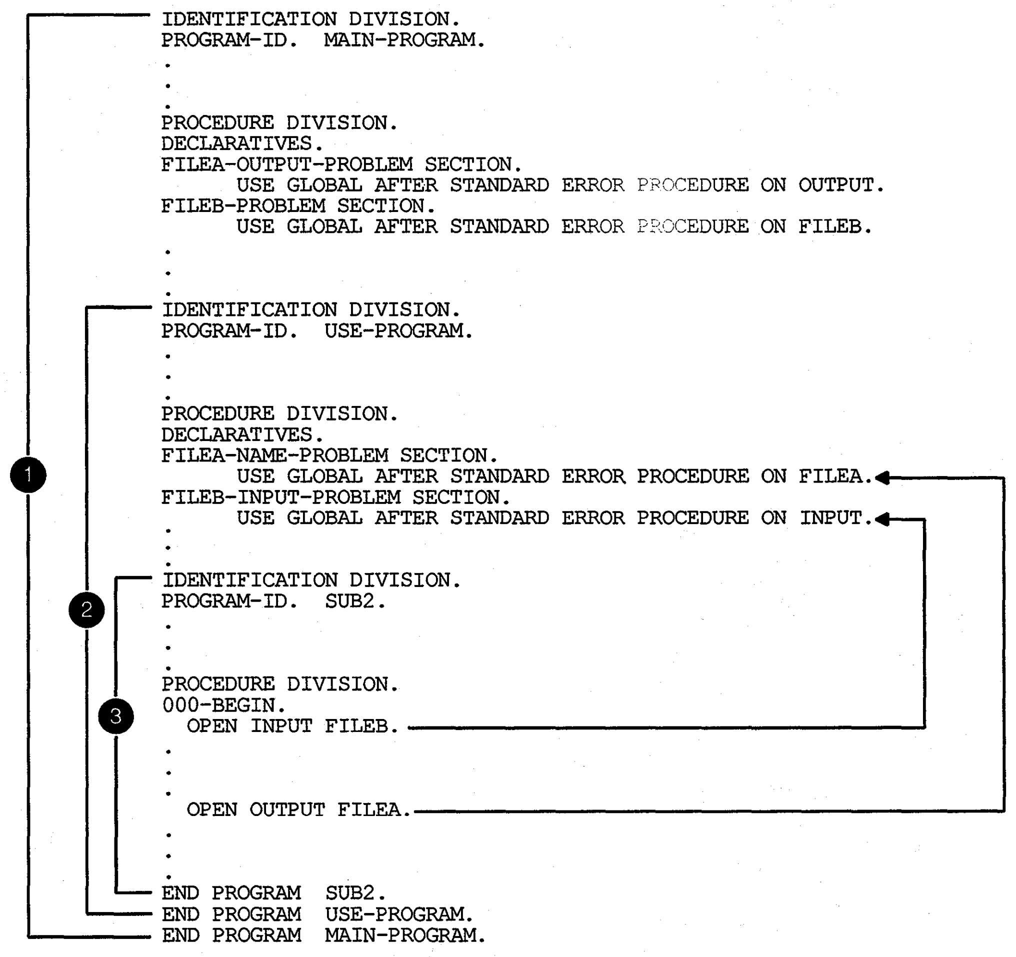
```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAIN-PROGRAM.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB1.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID.  USE-PROGRAM.
.
.
.
PROCEDURE DIVISION.
DECLARATIVES.
FILE-NAME-PROBLEM SECTION.
     USE AFTER STANDARD ERROR PROCEDURE ON FILEA.


OPEN INPUT FILEA.
.
.
.
END PROGRAM  USE-PROGRAM.
END PROGRAM  SUB1.
END PROGRAM  MAIN-PROGRAM.
```

ZK-1428A-GE

2. RULE 2—If a declarative is not found using rule 1, the Run-Time System searches all programs directly or indirectly containing that program for a global use procedure. This search continues until the Run-Time System either: (1) finds an applicable USE GLOBAL declarative, or (2) finds the outermost containing program. Either condition terminates the search; the second condition terminates both the search and the run unit.

Figure 18–11 shows applicable USE GLOBAL declaratives found in a containing program before the outermost containing program. Note that the first OPEN goes to the mode-specific procedure in the USE-PROGRAM rather than the file-specific procedure in the MAIN-PROGRAM.

Figure 18–11: Executing Declaratives Within Contained Programs (Rule 2)



```
IDENTIFICATION DIVISION.
PROGRAM-ID.  MAIN-PROGRAM.
.
.
.
PROCEDURE DIVISION.
DECLARATIVES.
FILEA-OUTPUT-PROBLEM SECTION.
      USE GLOBAL AFTER STANDARD ERROR PROCEDURE ON OUTPUT.
FILEB-PROBLEM SECTION.
      USE GLOBAL AFTER STANDARD ERROR PROCEDURE ON FILEB.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID.  USE-PROGRAM.
.
.
.
PROCEDURE DIVISION.
DECLARATIVES.
FILEA-NAME-PROBLEM SECTION.
      USE GLOBAL AFTER STANDARD ERROR PROCEDURE ON FILEA.
FILEB-INPUT-PROBLEM SECTION.
      USE GLOBAL AFTER STANDARD ERROR PROCEDURE ON INPUT.
.
.
.
IDENTIFICATION DIVISION.
PROGRAM-ID.  SUB2.
.
.
.
PROCEDURE DIVISION.
000-BEGIN.
  OPEN INPUT FILEB.
.
.
.
  OPEN OUTPUT FILEA.
.
.
.
END PROGRAM  SUB2.
END PROGRAM  USE-PROGRAM.
END PROGRAM  MAIN-PROGRAM.
```

18.6 Including Non-COBOL Programs in the Run Unit

Because the VAX COBOL compiler is part of the VMS common language environment, a VAX COBOL program can call a procedure written in another VAX language. This communication among high-level languages exists because VAX languages adhere to the VAX Procedure Calling and Condition Handling Standard when generating a call to a procedure. The standard states that any call to a procedure must be handled using the CALLS or the CALLG VAX instructions. Section 18.8 briefly describes the standard. For detailed information, refer to the VMS documentation on system routines, which also describes error handling by the different languages.

Calling a procedure written in another language allows you to take advantage of features in other languages. Example 18–3 and Example 18–5 demonstrate how to call non-COBOL programs in the run unit.

Example 18–3 shows how to call a BASIC program from a VAX COBOL program.

Example 18–3: Calling a BASIC Program from VAX COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    APPL.
******************************************************
* This VAX COBOL program accepts credit application  *
* information and passes this information to a BASIC *
* program that performs a credit analysis. Notice    *
* that the data passed to the BASIC program is in    *
* the standard VAX binary format.                    *
******************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01  APPLICATION-NUMBER   PIC 999.
01  C-APPLICATION-NUMBER PIC 9(9) COMP.
01  ANNUAL-SALARY        PIC 9(9).
01  C-ANNUAL-SALARY      PIC 9(9) COMP.
01  MORTGAGE-RENT        PIC 999.
01  C-MORTGAGE-RENT      PIC 9(9) COMP.
01  YEARS-EMPLOYED       PIC 99.
01  C-YEARS-EMPLOYED     PIC 9(9) COMP.
01  YEARS-AT-ADDRESS     PIC 99.
01  C-YEARS-AT-ADDRESS   PIC 9(9) COMP.
PROCEDURE DIVISION.
010-BEGIN.
    DISPLAY "Enter 3 digit application number".
    ACCEPT APPLICATION-NUMBER.
    IF APPLICATION-NUMBER = 999
    DISPLAY "All applicants processed" STOP RUN.
    MOVE APPLICATION-NUMBER TO C-APPLICATION-NUMBER.
    DISPLAY "Enter 5 digit annual salary".
    ACCEPT ANNUAL-SALARY.
    MOVE ANNUAL-SALARY TO C-ANNUAL-SALARY.
    DISPLAY "Enter 3 digit mortgage/rent".
    ACCEPT MORTGAGE-RENT.
    MOVE MORTGAGE-RENT TO C-MORTGAGE-RENT.
    DISPLAY "Enter 2 digit years employed by current employer".
    ACCEPT YEARS-EMPLOYED.
    MOVE YEARS-EMPLOYED TO C-YEARS-EMPLOYED.
    DISPLAY "Enter 2 digit years at present address".
    ACCEPT YEARS-AT-ADDRESS.
    MOVE YEARS-AT-ADDRESS TO C-YEARS-AT-ADDRESS.
    CALL "APP" USING BY REFERENCE C-APPLICATION-NUMBER
    C-ANNUAL-SALARY C-MORTGAGE-RENT
    C-YEARS-EMPLOYED C-YEARS-AT-ADDRESS.
    GO TO 010-BEGIN.
```

Example 18–4 shows the BASIC program APP called in Example 18–3, and sample output from the program's execution.

Example 18–4: BASIC Program APP and Output Data

```
10 SUB APP (A%,B%,C%,D%,E%)
40 IF A% = 999 THEN 999
50 IF B% => 26000 THEN 800
60 IF B% => 18000 THEN 600
70 IF B% > 15000 THEN 500
80 IF B% => 10000 THEN 400
90 GO TO 700
400 IF E% < 4 THEN 800
410 IF D% < 2 THEN 800
420 GO TO 800
500 IF E% < 4 THEN 700
510 GO TO 800
600 LET X% = B% / 12
650 IF C% => X%/4 THEN 670
660 GO TO 800
670 IF E% => 4 THEN 800
700 PRINT TAB(1);"APPLICANT NUMBER ";A%; " REJECTED"
710 GO TO 999
800 PRINT TAB(1);"APPLICANT NUMBER ";A%;" ACCEPTED"
999 SUBEND
```

Sample Run of APPL

```
$ RUN APPL
Enter 3 digit application number
376 RET
Enter 5 digit annual salary
35000 RET
Enter 3 digit mortgage/rent
461 RET
Enter 2 digit years employed by current employer
03 RET
Enter 2 digit years at present address
05 RET
APPLICANT NUMBER  376  ACCEPTED
Enter 3 digit application number
999 RET
All applicants processed
```

Example 18–5 shows how to call a FORTRAN program from a VAX COBOL program.

Example 18–5: Calling a FORTRAN Program from VAX COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   GETROOT.
*****************************************************
* This program accepts a value from the terminal,  *
* calls the FORTRAN subroutine SQROOT, and passes   *
* the value as a character string. Program          *
* SQROOT returns the square root of the value.      *
*****************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01  INPUT-NUMBER.
    03  INTEGER       PIC 9(5).
    03  DEC-POINT     PIC X(1).
    03  DECIMAL       PIC 9(8).
01  WORK-NUMBER.
    03  INTEGER       PIC 9(5).
    03  DECIMAL       PIC 9(8).
01  WORK-NUMBER-A REDEFINES WORK-NUMBER  PIC 9(5)V9(8).
01  DISPLAY-NUMBER    PIC ZZ,ZZ9.9999.
PROCEDURE DIVISION.
STARTER SECTION.
SBEGIN.
   MOVE SPACES TO INPUT-NUMBER.
   DISPLAY "Enter number (with decimal point): "
     NO ADVANCING.
   ACCEPT INPUT-NUMBER.
   IF INPUT-NUMBER = SPACES
     GO TO ENDJOB.
   CALL "SQROOT" USING BY DESCRIPTOR INPUT-NUMBER.
   IF INPUT-NUMBER = ALL "*"
     DISPLAY "** INVALID ARGUMENT FOR SQUARE ROOT"
   ELSE
     DISPLAY "The square root is: " INPUT-NUMBER
     INSPECT INPUT-NUMBER
       REPLACING ALL " " BY "0"
     MOVE CORRESPONDING INPUT-NUMBER TO WORK-NUMBER
     WORK-NUMBER-A TO DISPLAY-NUMBER
     DISPLAY DISPLAY-NUMBER.
   GO TO SBEGIN.
ENDJOB.
   STOP RUN.
```

Example 18–6 shows the FORTRAN program SQROOT called by the program in Example 18–5, and sample output from the programs' execution.

The SQROOT subroutine accepts a 14-character string and decodes it into a real variable (DECODE is analogous to an internal READ). It then calls the SQRT function in the statement that encodes the result into the 14-character argument.

Example 18–6: FORTRAN Subroutine SQROOT

```
      SUBROUTINE SQROOT(ARG)
      CHARACTER*14 ARG
      DECODE(14,10,ARG,ERR=20)VAL
10    FORMAT(F12.6)
      IF(VAL.LE.0.)GO TO 20
      ENCODE(14,10,ARG)SQRT(VAL)
999   RETURN
20    ARG='**************'
      GO TO 999
      END
```

Sample Run of GETROOT:

```
$ RUN GETROOT RET
Enter number (with decimal point): 25. RET
The square root is:     5.000000
     5.0000
Enter number (with decimal point): HELLO RET
** INVALID ARGUMENT FOR SQUARE ROOT
Enter number (with decimal point): 1000000. RET
The square root is:  1000.000000
1,000.0000
Enter number (with decimal point): 2. RET
The square root is:     1.414214
     1.4142
Enter number (with decimal point): RET
$
```

18.7 Using VAX COBOL in the Common Language Environment

The VAX COBOL compiler is part of the VMS common language environment. This environment defines certain calling procedures and guidelines that allow you to call programs written in different languages or prewritten system routines from VAX COBOL. You can call the following routine types from VAX COBOL:

- Subprograms written in other VAX languages
- VMS Run-Time Library routines
- VMS system services

The terms routine, procedure, and function are used throughout this chapter. A **routine** is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name) and optionally an argument list. Procedures and functions are specific types of routines: a **procedure** is a routine that does not return a value, whereas a **function** is a routine that returns a value by assigning that value to the function's identifier. In COBOL, routines are also referred to as subprograms and called programs.

System routines are prewritten VMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that VAX COBOL supports the data structures required to call the routine. The system routines used most often are VMS Run-Time Library routines and system services. For more information on system routines, refer to the VMS documentation on Run-Time Library routines and VMS system services.

18.8 The VAX Procedure Calling and Condition Handling Standard

The VAX Procedure Calling and Condition Handling Standard describes the concepts used by all VAX languages for invoking routines and passing data between them. The following attributes are specified by the VAX Procedure Calling and Condition Handling Standard:

- Register usage
- Stack usage
- Function value return
- Argument list

The following sections discuss these attributes in more detail. The VAX Procedure Calling and Condition Handling Standard also defines such attributes as the calling sequence, the argument data types and descriptor formats, condition handling, and stack unwinding. These attributes are discussed in detail in the VMS documentation on system services.

18.8.1 Register and Stack Usage

The VAX Procedure Calling and Condition Handling Standard defines several registers and their uses, as listed in Table 18–1.

Table 18–1: VAX Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Current stack frame pointer
AP	Argument pointer
R1	Environment value (when necessary)
R0, R1	Function value return registers

By definition, any called routine can use registers R2 to R11 for computation and the AP register as a temporary register.

In the VAX Procedure Calling and Condition Handling Standard, a **stack** is defined as a LIFO (last-in/first-out) temporary storage area that the system allocates for every user process. The system keeps information about each routine call in the current image on the call stack. Then, each time you call a routine, the system creates a structure known as the **call frame** on this call stack. The call frame for each active process contains the following:

- A pointer to the call frame of the previous routine call. This pointer corresponds to the frame pointer (FP).
- The argument pointer (AP) of the previous routine call.
- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC).

- The contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When a routine completes execution, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

18.8.2 Return of the Function Value

A function is a routine that returns a single value to the calling routine. The **function value** represents the return value that is assigned to the function's identifier during execution. According to the VAX Procedure Calling and Condition Handling Standard, a function value may be returned as either an actual value or a condition value that indicates success or failure.

18.8.3 The Argument List

The VAX Procedure Calling and Condition Handling Standard also defines a data structure called the argument list. An argument list passes information to a routine and receives results. An **argument list** is a collection of longwords in memory that represents a routine parameter list and possibly includes a function value. To pass data between routines that are not written in the same language, you must specify an argument-passing mechanism. Section 18.4.1 describes the argument-passing mechanisms for VAX COBOL.

For additional information on the VAX Procedure Calling and Condition Handling Standard, see the VMS documentation on system services routines.

18.9 VMS Run-Time Library Routines

The VMS Run-Time Library is a library of prewritten, commonly used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular VMS Run-Time Library facility. Table 18–2 lists all the language-independent Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 18–2: Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control a DIGITAL DECtalk device
LIB$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations

(continued on next page)

Table 18–2 (Cont.): Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
OTS$	General-purpose routines that perform tasks such as data type conversions as part of a compiler's generated code
SMG$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings
PPL$	Parallel processing routines that help you implement concurrent programs on single-CPU and multiprocessor systems

18.10 VMS System Services Routines

System services are prewritten system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the VMS Run-Time Library routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). However, these services are logically divided into groups that perform similar tasks. Table 18–3 describes these groups.

Table 18–3: System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of asynchronous system traps
Change Mode	Changes the access mode of particular routines
Condition Handling	Designates condition handlers for special purposes
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters
Information	Returns information about the system, queues, jobs, processes, locks, and devices
Input/Output	Performs I/O directly, without going through RMS
Lock Management	Enables processes to coordinate access to shareable system resources
Logical Names	Provides methods of accessing and maintaining pairs of character-string logical names and equivalence names
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data
Process Control	Creates, deletes, and controls execution of processes
Security	Enhances the security of VMS systems
Time and Time Conversion	Schedules events and obtains and formats binary time values

18.11 Calling Routines

The basic steps for calling routines are the same whether you are calling a routine (subprogram) written in VAX COBOL, a routine written in some other VAX language, a system service, or a VMS Run-Time Library routine. There are five steps required to call any system routine:

1. Determining the type of call
2. Defining the arguments
3. Calling the routine or service
4. Checking the condition value, if applicable
5. Locating the result

The following sections outline the steps for calling non-VAX COBOL routines.

18.11.1 Determining the Type of Call

Before you call an external routine, you must first determine whether the call should be a procedure call or a function call. In VAX COBOL, a routine that does not return a value should be called as a procedure call. A routine that returns a value should be called as a function call. Thus, a function call returns one of the following:

- A function value (a COMP integer, COMP-1, or COMP-2 number). For example, the call LIB$INDEX returns an integer value.
- A return status, which is a longword (PIC 9(5) to 9(9) USAGE IS COMP) condition value that indicates the program has either successfully executed or failed. For example, LIB$GET_INPUT returns a return status.

Although you can call most system routines as a procedure call, it is recommended that you do so *only* when the system routine does not return a value. By checking the condition value, you can avoid errors. The VMS documentation on system services and Run-Time Library routines contains descriptions of each system routine and a description of the condition values returned. For example, the RETURNS section for the system routine LIB$STAT_TIMER follows:

RETURNS

VMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism	by value

Because LIB$STAT_TIMER returns a value, it should be called as a function. If a system routine contains the following description under the RETURNS section, you should call the system routine as a procedure call:

RETURNS

None.

18.11.2 Defining the Argument

Most system routines have one or more arguments. These arguments are used to pass information to the system routine and to obtain information from it. Arguments can be either required or optional, and each argument has the following characteristics:

- Access type (read, write, modify...)
- Data type (floating point, longword...)
- Passing mechanisms (by value, by reference, by descriptor...)
- Argument form (scalar, array, string...)

To determine which arguments are required by a routine, check the format description of the routine in the VMS documentation on system services or Run-Time Library routines. For example, the format for LIB$STAT_TIMER is as follows:

```
LIB$STAT_TIMER code, value [,handle-adr]
```

The handle-adr argument appears in square brackets ([]), indicating that it is an optional argument. Hence, when you call the system routine LIB$STAT_TIMER, only the first two arguments are required.

Once you have determined which arguments you need, read the argument description for information on how to call that system routine. For example, the system routine LIB$STAT_TIMER provides the following description of the code argument:

code

VMS Usage:	longword_signed
type:	longword integer (signed)
access:	read only
mechanism:	by reference

```
Code that specifies the statistic to be returned. The code
argument contains the address of a signed longword
integer that is this code. It must be an integer from 1 to 5.
```

After you check the argument description, refer to Table 18–4 for the VAX COBOL equivalent of the argument description. For example, the code argument description lists the VMS usage entry longword_signed. To define the code argument, use the VAX COBOL equivalent of longword_signed:

```
01 LWS    PIC S9(9) COMP.
```

Follow the same procedure for the value argument. The description of value contains the following information:

value

VMS Usage:	varying_arg
type:	unspecified
access:	write only
mechanism:	by reference

```
The statistic returned by LIB$STAT_TIMER. The value
argument contains the address of a longword or quadword
that is this statistic.  All statistics are longword integers
except elapsed time, which is a quadword.
```

For the value argument, the VMS usage, varying_arg, indicates that the data type returned by the routine is dependent on other factors. In this case, the data type returned is dependent upon which statistic you want to return. For example, if the statistic you want to return is code 5, page fault count, you must use a signed longword integer. Refer to Table 18–4 to find the following definition for a longword_signed:

```
01 LWS    PIC S9(9) COMP.
```

Regardless of which Run-Time Library routine or system service you call, you can find the definition statements for the arguments in the VMS usage in Table 18–4.

18.11.3 Calling the External Routine

Once you have decided which routine you want to call, you can access the routine using the CALL statement. You set up the call to the routine or service the same way you set up any call in VAX COBOL. To determine the syntax of the CALL statement for a function call or a procedure call, see the *VAX COBOL Reference Manual*, and refer to the examples in this chapter.

Remember, you must specify the name of the routine being called and all parameters required for that routine. Make sure the data types and passing mechanisms for the parameters you are passing coincide with those defined in the routine.

18.11.4 Calling System Routines

The basic steps for calling system routines are the same as those for calling any routine. However, when calling system routines, you need to provide some additional information discussed in the following sections.

18.11.4.1 System Routine Arguments

All system routine arguments are described in terms of the following information:

- VMS usage
- Data type
- Type of access allowed
- Passing mechanism

VMS usages are data structures layered on the standard VMS data types. For example, the VMS usage mask_longword signifies an unsigned longword integer used as a bit mask, and the VMS usage floating_point represents any VMS floating-point data type. Table 18–4 lists the VMS usages and the VAX COBOL statements you need to implement them.

Table 18–4: VAX COBOL Implementation

VMS Data Type	VAX COBOL Definition
access_bit_names	NA . . . PIC X(128).[1]
access_mode	NA . . . PIC X.[1] access_mode is usually passed BY VALUE as PIC 9(9) COMP.
address	USAGE POINTER.
address_range	01 ADDRESS-RANGE. 02 BEGINNING-ADDRESS USAGE POINTER. 02 ENDING-ADDRESS USAGE POINTER.
arg_list	NA . . . Constructed by the compiler as a result of using the COBOL CALL statement. An argument list may be created as follows, but may not be referenced by the COBOL CALL statement. 01 ARG-LIST. 02 ARG-COUNT PIC S9(9) COMP. 02 ARG-BY-VALUE PIC S9(9) COMP. 02 ARG-BY-REFERENCE USAGE POINTER 02 VALUE REFERENCE ARG-NAME. . . . continue as needed
ast_procedure	01 AST-PROC PIC 9(9) COMP.[2]
boolean	01 BOOLEAN-VALUE PIC 9(9) COMP.[2]
byte_signed	NA . . . PIC X.[1]
byte_unsigned	NA . . . PIC X.[1]
channel	01 CHANNEL PIC 9(4) COMP.[2]
char_string	01 CHAR-STRING PIC X to PIC X(65535).
complex_number	NA . . . PIC X(n) where n is length.[1]
cond_value	01 COND-VALUE PIC 9(9) COMP.[2]
context	01 CONTEXT PIC 9(9) COMP.[2]
date_time	NA . . . PIC X(8).[1]
device_name	01 DEVICE-NAME PIC X(n) where n is length.
ef_cluster_name	01 CLUSTER-NAME PIC X(n) where n is length.
ef_number	01 EF-NO PIC 9(9) COMP.[2]
exit_handler_block	NA . . . PIC X(n) where n is length.[1]
fab	NA . . . Too complex for general COBOL use. Most of a FAB structure can be described by a lengthy COBOL record description, but such a FAB cannot then be referenced by a COBOL I-O statement. It is much simpler to do the I-O completely within COBOL, and let the COBOL compiler generate the FAB structure, or do the I-O in another language.
file_protection	01 FILE-PROT PIC 9(4) COMP.[2]

[1] Most VMS data types not directly supported in VAX COBOL can be represented as an alphanumeric data item of a certain number of bytes. While VAX COBOL does not interpret the data type, it may be used to pass objects from one language to another.

[2] Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

(continued on next page)

Table 18–4 (Cont.): VAX COBOL Implementation

VMS Data Type	VAX COBOL Definition
floating_point	01 F-FLOAT USAGE COMP-1. 01 D-FLOAT USAGE COMP-2. * g-float and h-float are not supported in VAX COBOL.
function_code	01 FUNCTION-CODE. 02 MAJOR-FUNCTION PIC 9(4) COMP.[2] 02 SUB-FUNCTION PIC 9(4) COMP.[2]
identifier	01 ID PIC 9(9) COMP.[2]
io_status_block	01 IOSB. 02 COND-VAL PIC 9(4) COMP.[2] 02 BYTE-CNT PIC 9(4) COMP.[2] 02 DEV-INFO PIC 9(9) COMP.[2]
item_list_2	01 ITEM-LIST-TWO. 02 ITEM-LIST OCCURS n TIMES. 04 COMP-LENGTH PIC S9(4) COMP. 04 ITEM-CODE PIC S9(4) COMP. 04 COMP-ADDRESS PIC S9(9) COMP. 02 TERMINATOR PIC S9(9) COMP VALUE 0.
item_list_3	01 ITEM-LIST-3. 02 ITEM-LIST OCCURS n TIMES. 04 BUF-LEN PIC S9(4) COMP. 04 ITEM-CODE PIC S9(4) COMP. 04 BUFFER-ADDRESS PIC S9(9) COMP. 04 LENGTH-ADDRESS PIC S9(9) COMP. 02 TERMINATOR PIC S9(9) COMP VALUE 0.
item_list_pair	01 ITEM-LIST-PAIR. 02 ITEM-LIST OCCURS n TIMES. 04 ITEM-CODE PIC S9(9) COMP. 04 ITEM-VALUE PIC S9(9) COMP. 02 TERMINATOR PIC S9(9) COMP VALUE 0.
item_quota_list	NA
lock_id	01 LOCK-ID PIC 9(9) COMP.[2]
lock_status_block	NA
lock_value_block	NA
logical_name	01 LOG-NAME PIC X TO X(255).
longword_signed	01 LWS PIC S9(9) COMP.
longword_unsigned	01 LWU PIC 9(9) COMP.[2]
mask_byte	NA . . . PIC X.[1]
mask_longword	01 MLW PIC 9(9) COMP.[2]
mask_quadword	01 MQW PIC 9(18) COMP.[2]
mask_word	01 MW PIC 9(4) COMP.[2]

[1] Most VMS data types not directly supported in VAX COBOL can be represented as an alphanumeric data item of a certain number of bytes. While VAX COBOL does not interpret the data type, it may be used to pass objects from one language to another.

[2] Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

(continued on next page)

Table 18–4 (Cont.): VAX COBOL Implementation

VMS Data Type	VAX COBOL Definition
null_arg	CALL . . . USING OMITTED or PIC S9(9) COMP VALUE 0 passed USING BY VALUE.
octaword_signed	NA
octaword_unsigned	NA
page_protection	01 PAGE-PROT PIC 9(9) COMP.[2]
procedure	01 ENTRY-MASK PIC 9(9) COMP.[2]
process_id	01 PID PIC 9(9) COMP.[2]
process_name	01 PROCESS-NAME PIC X TO X(15).
quadword_signed	01 QWS PIC S9(18) COMP.
quadword_unsigned	01 QWU PIC 9(18) COMP.[2]
rights_holder	01 RIGHTS-HOLDER. 02 RIGHTS-ID PIC 9(9) COMP.[2] 02 ACCESS-RIGHTS PIC 9(9) COMP.[2]
rights_id	01 RIGHTS-ID PIC 9(9) COMP.[2]
rab	NA . . . Too complex for general COBOL use. Most of a RAB structure can be described by a lengthy COBOL record description, but such a RAB cannot then be referenced by a COBOL I-O statement. It is much simpler to do the I-O completely within COBOL, and let the COBOL compiler generate the RAB structure, or do the I-O in another language.
section_id	01 SECTION-ID PIC 9(18) COMP.[2]
section_name	01 SECTION-NAME PIC X to X(43).
system_access_id	01 SECTION-ACCESS-ID PIC 9(18) COMP.[2]
time_name	01 TIME-NAME PIC X(n) where n is the length.
uic	01 UIC PIC 9(9) COMP.[2]
user_arg	01 USER-ARG PIC 9(9) COMP.[2]
varying_arg	Dependent upon application.
vector_byte_signed	NA . . . [3]
vector_byte_unsigned	NA . . . [3]
vector_longword_signed	NA . . . [3]
vector_longword_unsigned	NA . . . [3]
vector_quadword_signed	NA . . . [3]
vector_quadword_unsigned	NA . . . [3]
vector_word_signed	NA . . . [3]
vector_word_unsigned	NA . . . [3]
word_signed	01 WS PIC S9(4) COMP.
word_unsigned	01 WS PIC 9(4) COMP.[2]

[2] Although unsigned computational data structures are not directly supported in VAX COBOL, you may substitute the signed equivalent provided you do not exceed the range of the signed data structure.

[3] VAX COBOL does not permit the passing of variable-length data structures.

18.11.4.2 Calling a System Routine in a Function Call

In the following example, LIB$STAT_TIMER returns a condition value called RET-STATUS. To call this system routine, use the FORMAT of the function call described in the VMS documentation on system services or Run-Time Library routines. In this case, the format is as follows:

```
01 ARG-CODE   PIC S9(9) COMP.
01 ARG-VALUE  PIC S9(9) COMP.
01 RET-STATUS PIC S9(9) COMP.
       .
       .
       .
    CALL "LIB$STAT_TIMER"
          USING BY REFERENCE ARG-CODE, ARG-VALUE
          GIVING RET-STATUS.
```

As stated earlier, you are not using the optional handle-arg argument. In a CALL statement, you can specify an optional argument in one of two ways:

```
[,optional-argument]
```

or

```
,[optional-argument]
```

If the comma appears outside of the brackets, you must pass a zero by value or use the OMITTED phrase to indicate the place of the omitted argument.

If the comma appears inside the brackets, you can omit the argument as long as it is the last argument in the list. For example, look at the optional arguments of a hypothetical routine, LIB$EXAMPLE_ROUTINE:

```
LIB$EXAMPLE_ROUTINE arg1 [,arg2] [,arg3] [,arg4]
```

You can omit the optional arguments without using a placeholder:

```
CALL "LIB$EXAMPLE_ROUTINE"
      USING ARG1
      GIVING RET-STATUS.
```

However, if you omit an optional argument in the middle of the argument list, you must insert a placeholder:

```
CALL "LIB$EXAMPLE_ROUTINE"
      USING ARG1, OMITTED, ARG3
      GIVING RET-STATUS.
```

In general, Run-Time Library routines use the [,optional-argument] format, while system services use the ,[optional-argument] format.

In passing arguments to the procedure, you must define the passing mechanism required if it is not the default. The default passing mechanism for all VAX COBOL data types is BY REFERENCE.

The passing mechanism required for a system routine argument is indicated in the argument description. The passing mechanisms allowed in system routines are those listed in the VAX Procedure Calling and Condition Handling Standard in the VMS documentation on system services routines.

If the passing mechanism expected by the routine or service differs from the default mechanism in VAX COBOL, you must override the default. To force an argument to be passed by a specific mechanism, refer to the following list:

- If the argument is described as "the address of," use BY REFERENCE, which is the default.
- If the argument is described as "the value of," use BY VALUE.

- If the argument is described as "address of descriptor," use BY DESCRIPTOR.

NOTE

> If a routine requires a passing mechanism that is not supported by VAX COBOL, calling that routine from VAX COBOL is not possible.

Even when you use the default passing mechanism, you can include the passing mechanism that is used. For example, to call LIB$STAT_TIMER, you can use either of the following definitions:

```
CALL "LIB$STAT_TIMER"
      USING ARG-CODE, ARG-VALUE
      GIVING RET-STATUS.

CALL "LIB$STAT_TIMER"
     USING BY REFERENCE ARG-CODE, ARG-VALUE
     GIVING RET-STATUS.
```

18.11.4.3 Calling a System Routine in a Procedure Call

If the routine or service you are calling does not return a function value or condition value, you can call the system routine as a procedure. The same rules apply to optional arguments; you must follow the calling sequence presented in the FORMAT section of the VMS documentation on system services or Run-Time Library routines.

One system routine that does not return a condition value or function value is the Run-Time Library routine LIB$SIGNAL. LIB$SIGNAL should always be called as a procedure, as shown in the following example:

```
01 ARG-VALUE PIC S9(5) COMP VALUE 90.
    .
    .
    .
    CALL "LIB$SIGNAL" USING BY VALUE ARG-VALUE.
```

18.11.5 Checking the Condition Value

Many system routines return a condition value that indicates success or failure; this value can be either returned or signaled. In general, system routines return a condition value with the following exceptions:

- The system routine returns a function value.
- The CONDITION VALUES RETURNED is None.
- There is no CONDITION VALUES RETURNED description, but rather a CONDITION VALUES SIGNALED description. (Success conditions are not signaled.)
- The call to the routine was made as a procedure call.

If any of these conditions apply, there is no condition value to check.

If there is a condition value, you must check this value to make sure that it indicates successful completion. All success condition values are listed in the CONDITION VALUES RETURNED description.

Condition values indicating success always appear first in the list of condition values for a particular routine, and success codes always have odd values. A success code common to many system routines is the condition value SS$_NORMAL, which indicates that the routine completed normally and successfully. You can reference the condition values symbolically in your COBOL program by specifying them in the EXTERNAL phrase of the VALUE IS clause. Symbolic names specified in VALUE IS EXTERNAL become link-time constants; that is, the evaluation of the symbolic name is deferred until link time because it is known only at link time. For example:

```
01 SS$_NORMAL PIC S9(5) COMP VALUE EXTERNAL SS$_NORMAL
     .
     .
     .
     CALL "LIB$STAT_TIMER" USING ARG-CODE, ARG-VALUE GIVING RET-STATUS.
     IF RET-STATUS NOT EQUAL SS$_NORMAL...
```

Because all success codes have odd values, you can check a return status for any success code. For example, you can cause execution to continue only if a success code is returned by including the following statement in your program.

```
IF RET-STATUS IS SUCCESS ...
```

Sometimes several success condition values are possible. You may only want to continue execution on specific success codes. For example, the $SETEF system service returns one of two success values: SS$_WASSET or SS$_WASCLR. If you want to continue only when the success code SS$_WASSET is returned, you can check for this condition value as follows and branch accordingly:

```
IF RET-STATUS EQUAL SS$_WASSET ...
```

or

```
EVALUATE RET-STATUS
     WHEN SS$_WASSET ...
```

If the condition value returned is not a success condition, then the routine did not complete normally, and the information it should have returned may be missing, incomplete, or incorrect.

You can also check for specific error conditions as follows:

```
WORKING-STORAGE SECTION.
01 USER-LINE     PIC X(30).
01 PROMPT-STR    PIC X(16) VALUE IS "Type Your Name".
01 OUT-LEN       PIC S9(4) USAGE IS COMP.
01 COND-VALUE    PIC S9(9) USAGE IS COMP.
88 LIB$_INPSTRTRU VALUE IS EXTERNAL LIB$_INPSTRTRU.
                     .
                     .
                     .
PROCEDURE DIVISION.
P0.
    CALL "LIB$GET_INPUT" USING BY DESCRIPTOR USER-LINE PROMPT-STR
                               BY REFERENCE OUT-LEN
                               GIVING COND-VALUE.
    EVALUATE TRUE
        WHEN LIB$_INPSTRTRU
             DISPLAY "User name too long"
        WHEN COND-VALUE IS FAILURE
             DISPLAY "More serious error".
                    .
                    .
                    .
```

18.11.5.1 Library Return Status and Condition Value Symbols

Library return status and condition value symbols have the following general form:

fac$_abcmnoxyz

where:

fac	is a 2- or 3-letter facility symbol (LIB, MTH, STR, OTS, BAS, COB, FOR, SS).
abc	are the first 3 letters of the first word of the associated message.
mno	are the first 3 letters of the next word.
xyz	are the first 3 letters of the third word, if any.

Articles and prepositions are not considered significant words in this format. If a significant word is only two letters long, an underscore character is used to fill out the third space. The VMS normal or success code is used to indicate successful completion. Some examples of this code are as follows:

RETURN Status	Meaning
LIB$_INSVIRMEM	Insufficient virtual memory
FOR$_NO_SUCDEV	No such device
MTH$_FLOOVEMAT	Floating overflow in Math Library procedure
BAS$_SUBOUTRAN	Subscript out of range

18.11.6 Locating the Result

Once you have defined the arguments, called the procedure, and checked the condition value, you are ready to locate the result. To find out where the result is returned, look at the description of the system routine you are calling.

For example, in the following call to MTH$ACOS the result is written into the variable COS:

```
01 ARG-CODE PIC S9(9) COMP VALUE 1.
01 COS                COMP1 VALUE 0.
         .
         .
         .
    CALL "MTH$ACOS" USING BY REFERENCE ARG-CODE GIVING COS.
```

This result is described in the VMS documentation on system services and Run-Time Library routines, under the description of the system routine.

18.12 Calling Shareable Images

When calling a subprogram installed as a shareable image, the program name specified in the CALL statement must be a literal.

VAX COBOL programs installed as shareable images cannot contain external files.

For more information on shareable images refer to Chapter 2 and the VMS Linker documentation.

18.13 Examples

This section provides examples that demonstrate how to call system routines from VAX COBOL programs.

Example 18–7 shows a procedure call and gives a sample run of the program RUNTIME. It calls MTH$RANDOM, a random number generator from the Run-Time Library, and generates 10 random numbers. To obtain different random sequences on separate runs, change the value of data item SEED for each run.

Example 18–7: Random Number Generator

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  RUNTIME.
*****************************************************
*  This program calls MTH$RANDOM, a random number   *
*  generator from the Run-Time Library.             *
*****************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SEED   PIC 9(5) COMP VALUE 967.
01 A-NUM  COMP-1.
01 C-NUM  PIC Z(5).
PROCEDURE DIVISION.
GET-RANDOM-NO.
    PERFORM 10 TIMES
       CALL "MTH$RANDOM" USING SEED GIVING A-NUM
       MULTIPLY A-NUM BY 100 GIVING C-NUM
       DISPLAY "Random Number is  " C-NUM
    END-PERFORM.
```

Example 18–8 shows output from a sample run of the RUNTIME program in Example 18–7.

Example 18–8: Sample Run of RUNTIME

```
Random Number is      1
Random Number is      7
Random Number is     92
Random Number is     90
Random Number is     22
Random Number is     29
Random Number is     65
Random Number is     38
Random Number is     32
Random Number is     40
```

Example 18–9: Using SYS$SETDIR

```
01 DIRECTORY  PIC X(24) VALUE  "[MYACCOUNT.SUBDIRECTORY]".
01 STAT PIC S9(9) COMP.
         .
         .
         .
    CALL "SYS$SETDIR" USING BY DESCRIPTOR DIRECTORY
                            OMITTED
                            OMITTED
                            GIVING STAT.
```

Example 18–9 shows a program fragment that calls the SYS$SETDIR system service.

Example 18–10 calls the System Service routine $ASCTIM.

Example 18–10: Using $ASCTIM

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   CALLTIME.
****************************************************
*  This program calls the system service routine   *
*  $ASCTIM which converts binary time to an ASCII  *
*  string representation.                          *
****************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TIMLEN              PIC 9999  COMP VALUE 0.
01  D-TIMLEN            PIC 9999  VALUE 0.
01  TIMBUF              PIC X(24) VALUE SPACES.
01  RETURN-VALUE        PIC S9(9) COMP VALUE 999999999.
PROCEDURE DIVISION.
000-GET-TIME.
      DISPLAY "CALL SYS$ASCTIM".
      CALL "SYS$ASCTIM" USING BY REFERENCE TIMLEN
                              BY DESCRIPTOR TIMBUF
                              OMITTED
                              GIVING RETURN-VALUE.
      IF RETURN-VALUE IS SUCCESS
         THEN
             DISPLAY "DATE/TIME " TIMBUF
             MOVE TIMLEN TO D-TIMLEN
             DISPLAY "LENGTH OF RETURNED = " D-TIMLEN
         ELSE
             DISPLAY "ERROR".
      STOP RUN.
```

Example 18–11 shows output from a sample run of the CALLTIME program in Example 18–10.

Example 18–11: Sample Run of CALLTIME

```
CALL SYS$ASCTIM
DATE/TIME 21-APR-1987 09:34:33.45
LENGTH OF RETURNED = 0023
```

The following example shows how to call the procedure that inserts a variable bit field (LIB$INSV) from a COBOL program. The format of the LIB$INSV procedure is explained in the VMS documentation on Run-Time Library routines. To set the low order three bits of BIT-RESULT to 4, you would code the following:

```
WORKING-STORAGE SECTION.
01   SRC             PIC S9(9) USAGE IS COMP.
01   POS             PIC S9(9) USAGE IS COMP.
01   SIZ             PIC S9(9) USAGE IS COMP.
01   BIT-RESULT      PIC S9(9) USAGE IS COMP.
             .
             .
             .
PROCEDURE DIVISION.
             .
             .
             .
P0.
     MOVE 4 TO SRC.
     MOVE 0 TO POS.
     MOVE 3 TO SIZ.

     CALL "LIB$INSV" USING SRC, POS, SIZ, BIT-RESULT.
```

The following example shows how to call the procedure that enables and disables detection of floating-point underflow (LIB$FLT_UNDER) from a COBOL program. The format of the LIB$FLT_UNDER procedure is explained in the VMS documentation on Run-Time Library routines.

```
WORKING-STORAGE SECTION.
01   NEW-SET         PIC S9(9) USAGE IS COMP.
01   OLD-SET         PIC S9(9) USAGE IS COMP.
             .
             .
             .
PROCEDURE DIVISION.
             .
             .
             .
P0.
     MOVE 1 TO NEW-SET.
     CALL "LIB$FLT_UNDER" USING NEW-SET GIVING OLD-SET.
```

The following example shows how to call the procedure that finds the first clear bit in a given bit field (LIB$FFC). This procedure returns a COMP longword condition value, represented in the example as RETURN-STATUS.

```
WORKING-STORAGE SECTION.
01   START-POS       PIC S9(9) USAGE IS COMP VALUE 0.
01   SIZ             PIC S9(9) USAGE IS COMP VALUE 32.
01   BITS            PIC S9(9) USAGE IS COMP VALUE 0.
01   POS             PIC S9(9) USAGE IS COMP VALUE 0.
01   RETURN-STATUS   PIC S9(9) USAGE IS COMP.
             .
             .
             .
PROCEDURE DIVISION.
             .
             .
             .
        CALL "LIB$FFC" USING START-POS,
                             SIZ,
                             BITS,
                             POS
                       GIVING RETURN-STATUS.

        IF RETURN-STATUS IS FAILURE
           THEN GO TO error-proc.
```

18.14 Additional Information

For more detailed information on system services and Run-Time Library routines, refer to the VMS documentation on system service routines and creating modular programs.

The VMS documentation on system routines contains the VAX Procedure Calling and Condition Handling Standard. The VMS Modular Programming Standard can be found in the VMS documentation on creating modular procedures.

All manuals listed in this chapter can be found in the VMS documentation set.

Part III
VAX COBOL Programming Options and Performance Considerations

Chapter 19

Using the REFORMAT Utility

The REFORMAT Utility converts terminal format source programs to conventional ANSI format and vice versa.

Consider the two formats:

- Terminal format is compatible with the VAX system. Terminal format eliminates the line-number and identification fields of ANSI format and allows horizontal tab characters and short lines. It saves disk space and decreases compile time.
- Conventional ANSI format produces source programs compatible with the reference format of other COBOL compilers.

The *VAX COBOL Reference Manual* discusses both formats in detail.

19.1 ANSI-to-Terminal Format Conversion

REFORMAT converts each ANSI format source line to terminal format by:

- Removing the 6-character sequence field in the first six character positions of the ANSI format line.
- Moving any continuation symbol (-) or comment symbols (* or /) from character position 7 into character position 1.
- Moving the conditional compilation character (if any) from the ANSI format indicator area into character position 2 and inserting a backslash character (\ ) into character position 1 of the terminal format line.
- Replacing spaces with horizontal tabs immediately to the right of Margin B and every eight character positions thereafter until the end of the line. This does not occur in source lines containing a nonnumeric literal.
- Removing the identification entry field in character positions 73 to 80 of the ANSI format line.
- Removing insignificant trailing spaces before character position 73 of the ANSI format line.
- Replacing every form-feed record with a line containing a slash (/) in character position 1.
- Placing the converted code in positions 1 to the end of the line, thereby creating a terminal format line.

19.1.1 ANSI-to-Terminal REFORMAT Command String

To run REFORMAT, enter this command:

```
$ RUN SYS$SYSTEM:REFORMAT
```

REFORMAT executes and prompts you with this message:

```
REFORMAT - ANSI-to-terminal conversion mode [ Y / N ]?
```

For an ANSI-to-terminal conversion, type Y and press RETURN. REFORMAT confirms your choice with this message:

```
REFORMAT - ANSI-to-terminal format selected
```

REFORMAT then asks for input and output file specifications:

```
REFORMAT -       ANSI-format input file spec :
REFORMAT -  Terminal-format output file spec :
```

REFORMAT reads the input file and writes a terminal format output file. After processing the last record, REFORMAT displays these messages:

```
REFORMAT - n ANSI COBOL source records converted to terminal format
REFORMAT - ANSI-to-terminal format conversion mode [ Y / N ]?
```

The first message indicates the number of input records converted to terminal format; the second message prompts you for conversion of another file. Enter the next file to convert, or type CTRL/Z to end execution.

19.2 Terminal-to-ANSI Format Conversion

REFORMAT converts each terminal format source line to ANSI format by:

- Placing a 6-character line number (000010) in the first six character positions of the first line and increasing it by 000010 for each subsequent line.
- Moving any continuation symbol (-), or the comment symbols (* or /) from character position 1 into character position 7.
- Removing the backslash character (\ ), if any, from character position 1 in terminal format and moving the following conditional compilation character into character position 7 of the ANSI format line.
- Replacing horizontal tabs with space characters at every eighth character position, starting at character position 5 and ending at the end of the line.
- Moving spaces into remaining character positions after the last character of code and before character position 73.
- Expanding a terminal line with more than 65 characters into two or more ANSI format lines and right-justifying these lines at character position 72.
- Placing either identification characters (if supplied at program initialization) or spaces into character positions 73 to 80.
- Right-justifying (at position 72) the first line of a continued nonnumeric literal. This ensures that the literal remains the same length as it was in the default format.
- Replacing every form-feed record with a line containing a slash (/) in position 7 and space characters in positions 8 to 72.
- Placing the converted code in character positions 8 to 72, thereby creating one or more ANSI format lines.

Note that it is possible to construct a terminal format line that converts to an invalid ANSI formatted line. Consider the case of a conditional compilation line with a long nonnumeric literal:

```
\A   01   FOO   PIC X(80)  VALUE A ... A.
```

This statement cannot be reformatted to a valid ANSI statement. The literal is 80 characters long, which indicates that the literal must be continued on the next line by placing a continuation symbol (-) in the indicator area. The line is also a conditional compilation line, which indicates that the A is to be placed in the indicator area. Clearly both characters cannot be placed in the indicator area. VAX COBOL continues the conditional compilation line by placing the A in the indicator area. This means the program remains valid if conditionals are not used in the compilation because the lines become comment lines. If conditionals are used, you must locate and correct these invalid lines. The reformat program is a text processor and does not perform the level of checking required by lines such as these. You detect this error during a compile operation.

19.2.1 Terminal-to-ANSI REFORMAT Command String

To run REFORMAT, enter this command:

```
$ RUN SYS$SYSTEM:REFORMAT
```

REFORMAT prompts you with this message:

```
REFORMAT - ANSI-to-terminal conversion mode [ Y / N ]?
```

For a terminal-to-ANSI conversion, type N and press RETURN. REFORMAT confirms your choice with this message:

```
REFORMAT - Terminal-to-ANSI format selected
```

REFORMAT then asks for input and output file specifications:

```
REFORMAT -   Terminal-format input file spec:
REFORMAT -      ANSI-format output file spec:
```

After you enter the file specifications, REFORMAT asks for an identification entry in columns 73 to 80:

```
REFORMAT - Columns 73 to 80:
```

If you want an identification entry, type from one to eight characters. REFORMAT places these characters, left-justified, in columns 73 to 80 of each output line. Otherwise, press RETURN.

REFORMAT reads the input file and writes the output file in 80-character ANSI format records. After processing the last record, REFORMAT displays these messages:

```
REFORMAT - n Terminal COBOL source records converted to ANSI format
REFORMAT - ANSI-to-terminal format conversion mode [ Y / N]?
```

The first message indicates the number of input records converted to ANSI format; the second message prompts you for conversion of another file. Type CTRL/Z to end execution.

19.3 REFORMAT Error Messages

If any of your responses to the prompts are incorrect, REFORMAT displays error messages. It replaces the parentheses and the parenthetical text with the appropriate format type you specified.

```
REFORMAT - Error in opening (ANSI or terminal) format input file:
REFORMAT -        (ANSI or terminal) format input file spec:
```

The system could not open the input file; either the file is not on the specified device or you typed the file name incorrectly. The default device is SYS$DISK.

To continue processing, examine the input file specification and type a corrected version. To process another file, type a new input file specification. To end execution, type CTRL/Z.

```
REFORMAT - Error in opening (ANSI or terminal) format output file:
REFORMAT -   (ANSI or terminal) format output file spec:
```

The system could not open the output file. An incorrectly typed file specification usually causes this error. The default device is SYS$DISK.

To continue, examine the output file specification and type a corrected version. To end execution, type CTRL/Z.

```
REFORMAT - (ANSI or terminal) format input file is empty
REFORMAT -     (ANSI or terminal) format input file spec:
```

The system opened an empty input file. To continue, type a new input file specification. To end execution, type CTRL/Z.

```
REFORMAT - Error in reading (ANSI or terminal) format input file
REFORMAT - Reformating aborted
REFORMAT - n (ANSI or terminal) COBOL  source records converted to
                 (ANSI or terminal) format
REFORMAT - ANSI-to-terminal format conversion mode [ Y or N ]?
```

REFORMAT failed to read a record from the input file. This error ends the conversion process. REFORMAT closes both files and displays the number of converted input records.

You can either convert another file or end the session by typing CTRL/Z.

```
REFORMAT - Error in writing (ANSI or terminal) format output file
REFORMAT - Reformatting aborted
REFORMAT - n (ANSI or terminal) COBOL source records converted to
                 (ANSI or terminal) format
REFORMAT - ANSI-to-terminal format conversion mode [ Y or N ]?
```

REFORMAT failed in an attempt to write an output record. It ends execution and closes both files.

To process another file, type a new input file specification and continue the prompting message sequence. To end execution, type CTRL/Z.

Chapter 20

Optimizing Your VAX COBOL Program

You can decrease processing time and save storage space by using compiler optimization features when you write VAX COBOL programs. This chapter shows how certain numeric data-type and Procedure Division statements can help you optimize your VAX COBOL programs. Refer to Section D.2 for information about optimizing your VAX COBOL programs for certain VAX processors.

The information in this chapter should be seen as guidelines only, and may not be appropriate in all cases. When using the suggestions in this chapter, you should use only those suggestions that fit your needs.

20.1 Numeric Data Representation

Optimizing numeric data is one way to improve program performance. In general, for arithmetic operations, BINARY (COMP) and PACKED-DECIMAL (COMP-3) items run faster than a numeric data item with a USAGE IS DISPLAY clause. The compiler must convert USAGE IS DISPLAY numeric data items to PACKED-DECIMAL items before performing arithmetic operations. More than one conversion is necessary. In addition, PACKED-DECIMAL and fixed BINARY items use less disk space than numeric display items.

For example, notice the storage space used by data items with a USAGE BINARY statement:

PICTURE Range	**Storage**	
S9 TO S9(4)	1 word	(2 bytes)
S9(5) to S9(9)	1 longword	(4 bytes)
S9(10) to S9(18)	1 quadword	(8 bytes)

In general, describe a numeric data item as USAGE BINARY when:

- The data item is part of an arithmetic operation and is less than one quadword.
- The data item is used as a subscript. In this case, allocate a longword by specifying PIC S9(5) to PIC S9(9).

Although not as storage-efficient as USAGE BINARY, data items with a USAGE PACKED-DECIMAL statement let you store two digits per byte rather than one digit per byte for USAGE DISPLAY items, as shown in the following example:

USAGE DISPLAY		USAGE PACKED-DECIMAL	
PICTURE	**Storage**	**PICTURE**	**Storage**
S9(5)V99	7 bytes	S9(5)V99	4 bytes
S9(12)V9	13 bytes	S9(12)V9	7 bytes

To calculate the number of storage bytes for a PACKED-DECIMAL item, divide the PICTURE size by 2 (without rounding) and add 1 to the result. You can check the allocation with the compiler qualifier /MAP.

In general, describe a numeric data item as USAGE PACKED-DECIMAL when:

- The receiving field is a numeric display or edited data item.
- The data item is part of a file's record description frequently used in arithmetic operations.

20.1.1 Scaling and Mixing Data-Types

Scaling is the process of aligning decimal points for numeric data items. Avoid mixing scale-factors and data-types in arithmetic operations.

In general, when you do numeric operations, it is better to use operands of the same usage and scale.

20.1.2 Using Significant Digits

In general, the fewer significant digits in an item, the better the performance (except as described in Section 20.1.1). For example, for a numeric data item to contain a number from 1 to 999, declare it as PIC 9(3), not PIC 9(10).

20.2 Choices in Procedure Division Statements

Some Procedure Division statements make better use of the VAX COBOL compiler than others. This section discusses these statements and shows how to use them.

20.2.1 Using ADD, SUBTRACT, MULTIPLY, and DIVIDE Instead of COMPUTE

The ADD, SUBTRACT, MULTIPLY, and DIVIDE statements are generally faster than the COMPUTE statement and use fewer intermediate temporaries; they usually execute fewer instructions. For example:

Record definitions:

```
01 A      PIC S9(4)V99 PACKED-DECIMAL.
01 B      PIC S9(4)V99 PACKED-DECIMAL.
01 C      PIC S9(4)V99 PACKED-DECIMAL.
01 D      PIC S9(4)V99 PACKED-DECIMAL.
01 E      PIC S9(4)V99 PACKED-DECIMAL.
01 F      PIC S9(4)V99 PACKED-DECIMAL.
01 TEMP1  PIC S9(4)V99 PACKED-DECIMAL.
01 TEMP2  PIC S9(4)V99 PACKED-DECIMAL.
```

Compute statement:

```
COMPUTE F = ((((A + B) - (C + D)) / 2) * E).
```

Separate arithmetic statements:

```
ADD A, B GIVING TEMP1.
ADD C, D GIVING TEMP2.
SUBTRACT TEMP2 FROM TEMP1.
DIVIDE TEMP1 BY 2 GIVING TEMP2.
MULTIPLY TEMP2 BY E GIVING F.
```

20.2.2 Using GO TO DEPENDING ON Instead of IF, GO TO

The GO TO DEPENDING ON statement generates fewer instructions than a sequence of IF and GO TO statements; it can also improve a program's readability. For example:

```
GO TO 100-PROCESS-MARRIED
   200-PROCESS-SINGLE
   300-PROCESS-DIVORCED
   400-PROCESS-WIDOWED
       DEPENDING ON MARITAL-STATUS.
```

The previous example generates fewer instructions and is easier to read than the following:

```
IF MARITAL-STATUS = 1
   GO TO 100-PROCESS-MARRIED.
IF MARITAL-STATUS = 2
   GO TO 200-PROCESS-SINGLE.
IF MARITAL-STATUS = 3
   GO TO 300-PROCESS-DIVORCED.
IF MARITAL-STATUS = 4
   GO TO 400-PROCESS-WIDOWED.
```

Remember, data items referenced by the DEPENDING ON clause must contain a numeric value that is: (1) greater than zero, and (2) not greater than the number of procedure names in the statement. Otherwise, control passes to the next executable statement.

20.2.3 Using Indexing Instead of Subscripting

Using index names for table handling is generally more efficient than using PACKED-DECIMAL or numeric DISPLAY subscripts, since the compiler declares index names as longword binary data items. Subscript data items described in the Working-Storage Section as longword binary items are as efficient as index items. Indexing also provides more flexibility in table-handling operations, since it allows you to use the SEARCH statement for sequential and binary searches.

The efficiency order for indexing and subscripting is as follows:

1. Index names, or subscript data items described as longword BINARY
2. Subscript data items described as 1-word BINARY
3. Subscript data items described as PACKED-DECIMAL
4. Subscript data items described as numeric DISPLAY

These two examples are equally efficient:

Example 1

```
WORKING-STORAGE SECTION.
01  TABLE-SIZE.
    03  FILLER                      PIC X(300).
01  THE-TABLE REDEFINES TABLE-SIZE.
    03  TABLE-ENTRY OCCURS 30 TIMES PIC X(10).
01  SUB1          PIC S9(5) BINARY VALUE ZEROES.
```

Example 2

```
WORKING-STORAGE SECTION.
01  TABLE-SIZE.
    03  FILLER                      PIC X(300).
01  THE-TABLE REDEFINES TABLE-SIZE.
    03  TABLE-ENTRY OCCURS 30 TIMES PIC X(10)
                    INDEXED BY IND-1.
```

If applicable, use a numeric literal to access a table. For example:

```
MOVE TABLE-ENTRY (numeric literal) TO ...
```

Using a numeric literal is faster than using either a subscript or an index:

```
MOVE TABLE-ENTRY (SUB1) TO ...
```

```
MOVE TABLE-ENTRY (IND-1) TO ...
```

20.2.4 Using PERFORM n TIMES Instead of PERFORM VARYING

If possible, use PERFORM n TIMES. It executes fewer instructions than PERFORM VARYING. For example:

```
PERFORM 050-MONTHLY-ANALYSIS 12 TIMES.
```

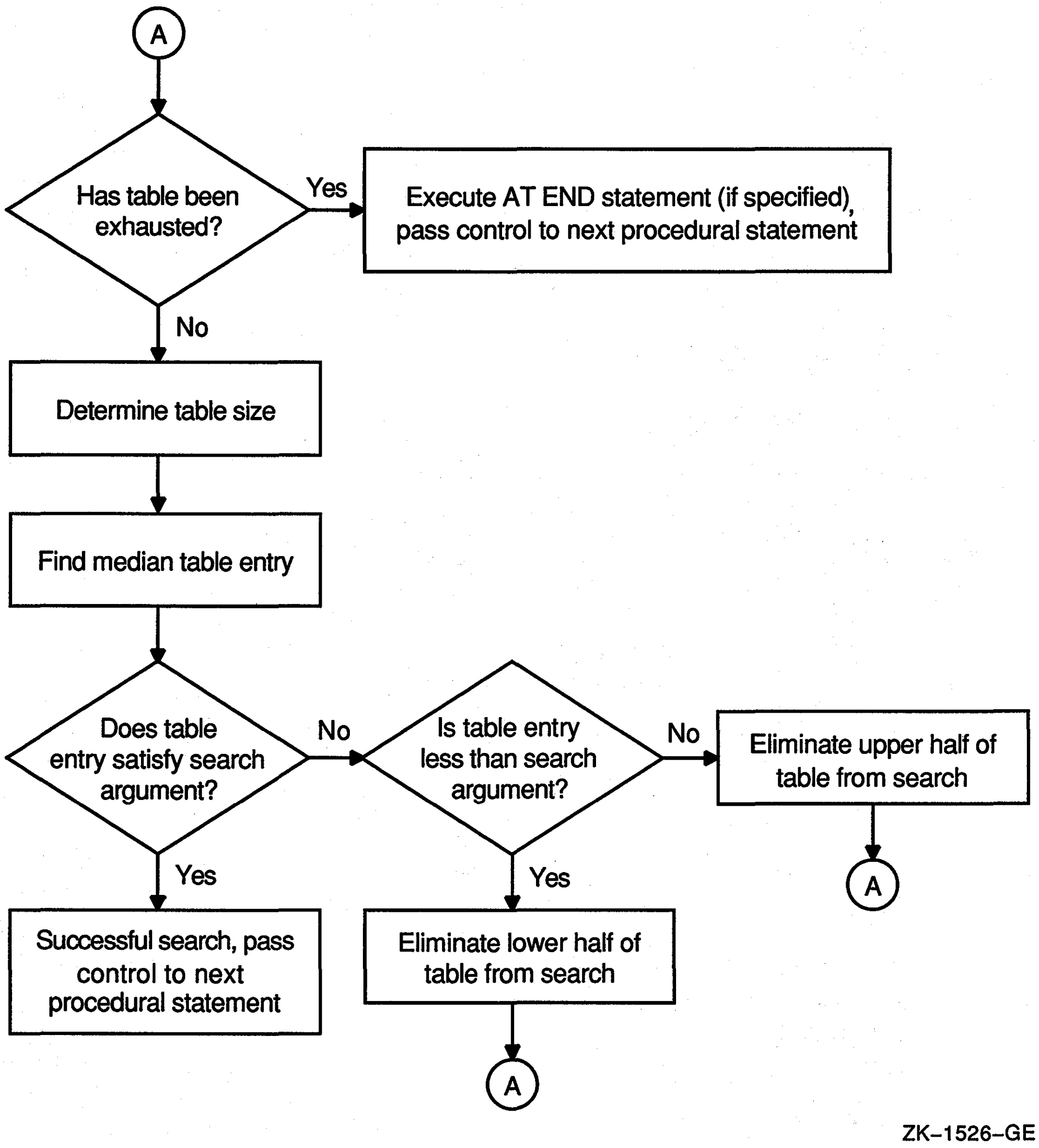
This is more efficient than:

```
PERFORM 050-MONTHLY-ANALYSIS
        VARYING A-COUNTER FROM 1 BY 1
        UNTIL A-COUNTER IS GREATER THAN 12.
```

20.2.5 Using SEARCH ALL Instead of SEARCH

When performing table look-up operations, SEARCH ALL, a binary search operation, is usually faster than SEARCH, a sequential search operation. A binary search determines a table's size, finds the median table entry, and searches the table in sections, by using compare processes. A sequential search manipulates the contents of an index to search the table sequentially (Figure 20–1 shows execution of a SEARCH ALL statement). However, SEARCH ALL requires the table to be in ascending or descending order by search key, while SEARCH imposes no restrictions on table organization. Section 6.4.8 contains examples of binary and sequential table-handling operations.

Figure 20–1: Execution of a SEARCH ALL Statement



A
Has table been exhausted?
Yes
Execute AT END statement (if specified), pass control to next procedural statement
No
Determine table size
Find median table entry
Does table entry satisfy search argument?
No
Is table entry less than search argument?
No
Eliminate upper half of table from search
A
Yes
Successful search, pass control to next procedural statement
Yes
Eliminate lower half of table from search
A
ZK-1526-GE

20.3 Using VAX COBOL for I/O Operations

VAX COBOL provides methods of controlling RMS actions during I/O operations. You have the choice of accepting the defaults RMS provides or using these optional methods to make your program more efficient.

The VAX COBOL language elements that can specify alternatives to the RMS defaults are as follows:

- The APPLY clause in the I-O-CONTROL paragraph
- The RESERVE n AREAS clause in the FILE-CONTROL paragraph

- The SAME RECORD AREA clause in the I-O-CONTROL paragraph
- The BLOCK CONTAINS clause in the FD entry

For additional information on the RMS terms and concepts included in this section, see the RMS reference documentation and the VMS documentation on RMS tuning.

20.3.1 Using the APPLY Clause

The APPLY clause in the I-O-CONTROL paragraph of the Environment Division provides phrases that you can use to improve I/O processing. Examine the following example of the APPLY clause:

```
      [ CONTIGUOUS          ]
APPLY [ CONTIGUOUS-BEST-TRY ]
                PREALLOCATION preall-amt ON { file-name } ...
```

20.3.1.1 Using the PREALLOCATION Phrase of the APPLY Clause

By default, the system does not preallocate disk blocks. As a result, files can require multiple extensions of disk blocks. Although file extension is transparent to your program, it takes time. To reduce this time, use the APPLY PREALLOCATION clause to preallocate disk blocks. Specifying APPLY PREALLOCATION preallocates noncontiguous disk blocks.

When you specify the CONTIGUOUS-BEST-TRY phrase, RMS makes up to three attempts to allocate as many contiguous disk blocks as it can; it then preallocates remaining blocks noncontiguously. The CONTIGUOUS-BEST-TRY phrase minimizes disk space fragmentation and gives better system throughput than CONTIGUOUS.

The APPLY CONTIGUOUS (physically adjacent) clause makes one attempt at contiguous preallocation; if it fails, the OPEN operation fails. Use APPLY CONTIGUOUS if you require a specific physical space on disk.

Contiguous files can reduce or eliminate window turning. When you access a file, the file system maps virtual block numbers to logical block numbers. This map is a window to the file. It contains one pointer for each file extent. The file system cannot map a large noncontiguous file: the file system may have to turn the window to access records in another extent. However, a contiguous file is one extent. It needs one map pointer only, and window turning does not take place after you open the file.

The following statements create a file (after OPEN/WRITE) and preallocate 150 contiguous blocks:

```
ENVIRONMENT DIVISION.
FILE-CONTROL.
    SELECT A-FILE ASSIGN TO "MYFILE".
                  .
                  .
                  .
I-O-CONTROL.
    APPLY CONTIGUOUS PREALLOCATION 150 ON A-FILE.
                  .
                  .
                  .
```

20.3.1.2 Using the EXTENSION Phrase of the APPLY Clause

APPLY EXTENSION extend-amt ON { file-name } ...

The APPLY EXTENSION clause is another way to reduce I/O allocation and extension time. Adding records to a file whose current extent is full causes the file system to extend the file by one disk cluster. (A disk cluster is a set of contiguous blocks; its size is determined when you initialize the volume with the DCL INITIALIZE command or when the volume is mounted with the DCL MOUNT qualifier: /EXTENSION=n.)

You can override the default extension by specifying the number of blocks in the APPLY EXTENSION clause. The APPLY EXTENSION integer becomes a file attribute stored with the file.

20.3.1.3 Using the DEFERRED-WRITE Phrase of the APPLY Clause

APPLY DEFERRED-WRITE ON { file-name } ...

Each WRITE or REWRITE statement can cause an I/O operation. The APPLY DEFERRED-WRITE clause permits writes to a file only when the buffer is full. Reducing the number of WRITE operations reduces file access time. However, the APPLY DEFERRED-WRITE clause can affect file integrity: records in the I/O buffer are not written to the file if the system crashes or the program aborts. You cannot use DEFERRED-WRITE on write-shared files.

20.3.1.4 Using the FILL-SIZE ON Phrase of the APPLY Clause

APPLY FULL-SIZE ON { file-name } ...

Use the APPLY FILL-SIZE clause to populate (load) the file and force COBOL to write records into the bucket area not reserved by the fill number. Routine record insertion uses the fill space, thereby reducing bucket splitting and the resulting overhead.

Do not use the APPLY FILL-SIZE clause for routine record insertion; it prohibits the use of bucket fill space and creates unnecessary buckets.

20.3.1.5 Using the WINDOW Phrase of the APPLY Clause

APPLY WINDOW ON { file-name } ...

Window size is the number of file mapping pointers stored in memory. A large window improves I/O because the system spends less time remapping the file.

When a disk is initialized, the default window size is set by specifying the /WINDOW qualifier. You can override this qualifier with the APPLY WINDOW clause. However, avoid specifying too large a window size. Window size is part of the system's pool space, and a large window size could affect system performance.

20.3.2 Using Multiple Buffers

Multibuffering can increase the speed of I/O operations by reducing the number of file accesses. When a program accesses a record already in the I/O buffer, the system moves the record to the current record area without executing an I/O operation.

You can specify multiple buffering by using the RESERVE clause in the SELECT statement of the Environment Division. The RESERVE clause specification overrides the system default. (The system default is usually set by means of the DCL command SET RMS_DEFAULT). This example reserves six areas for FILE-A:

```
SELECT FILE-A ASSIGN TO "FILE-A"
       RESERVE 6 AREAS.
```

You can specify up to 127 areas in the RESERVE clause. In general, specifying from 2 to 10 areas is best.

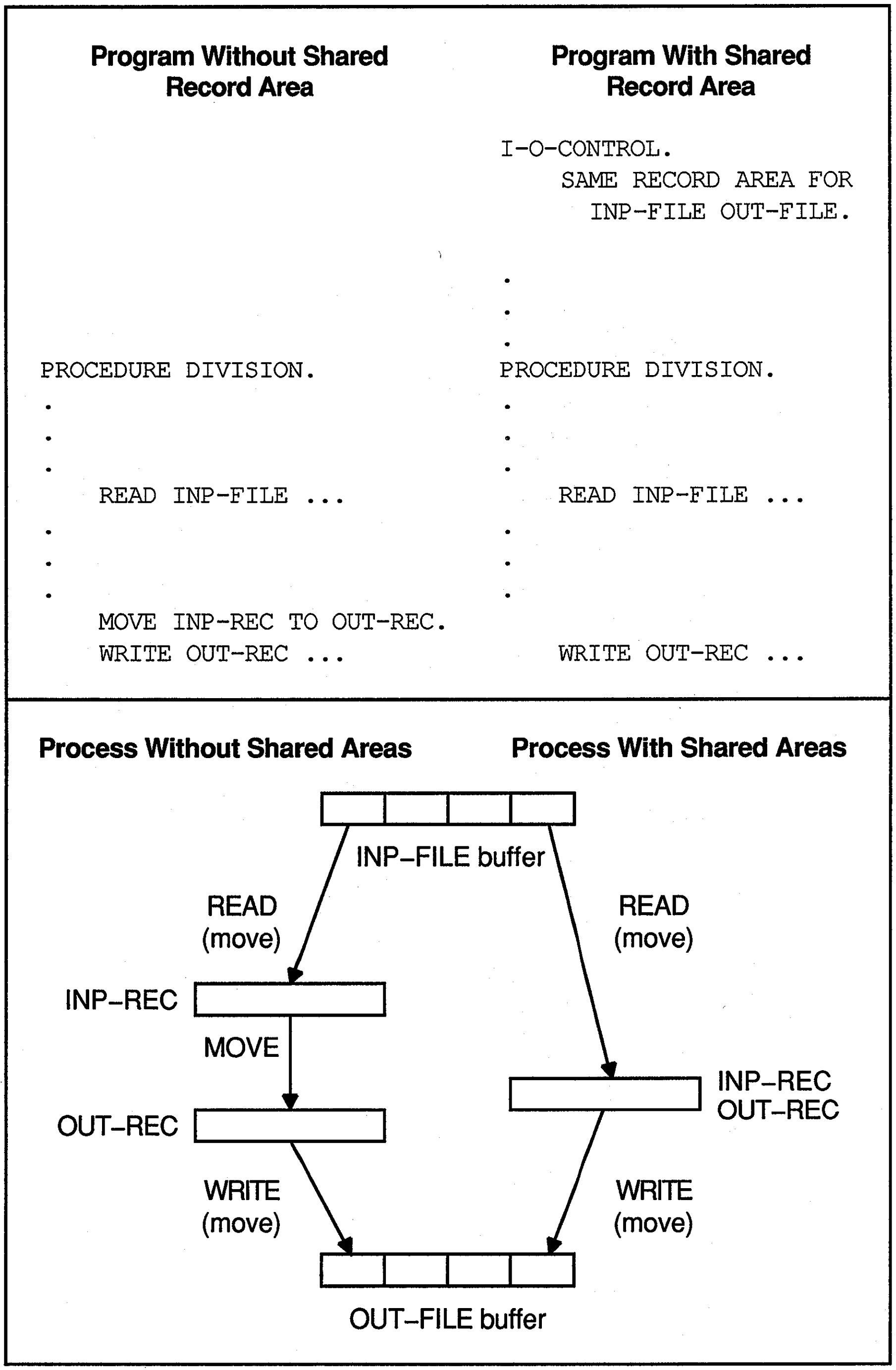
20.3.3 Sharing Record Areas

The compiler allocates unique storage space in the Data Division for each file's current record area. Transferring records between files requires an intermediate buffer area and adds to a program's processing requirements.

To reduce address space and processing overhead, files can share current record areas. Specify the SAME RECORD AREA clause in the I-O-CONTROL paragraph of the Environment Division. Records need not be the same size, nor must the maximum size of each current record area be the same.

Figure 20–2 shows the effect of current record area sharing in a program that reads records from one file and writes them to another. However, it also shows a drawback: current record area sharing is equivalent to implicit redefinition. The records do not exist separately. Therefore, if the program changes a record defined for the output file, the input file record is no longer available.

Figure 20–2: Sharing Record Areas



```
  Program Without Shared              Program With Shared
       Record Area                        Record Area

                                    I-O-CONTROL.
                                        SAME RECORD AREA FOR
                                          INP-FILE OUT-FILE.

                                    .
                                    .
                                    .
PROCEDURE DIVISION.                 PROCEDURE DIVISION.
.                                   .
.                                   .
.                                   .
    READ INP-FILE ...                   READ INP-FILE ...
.                                   .
.                                   .
.                                   .
    MOVE INP-REC TO OUT-REC.
    WRITE OUT-REC ...                   WRITE OUT-REC ...
```

ZK–1539–GE

20.4 Optimizing File Design

This section provides information on how to optimize the following file types:

- Sequential
- Relative
- Indexed

20.4.1 Sequential Files

Sequential files have the simplest structure and the fewest options for definition, population, and handling. You can reduce the number of disk accesses by minimizing record length.

With a sequential disk file, you can use multiblocking to access a buffer area larger than the default. Because the system transfers disk data in 512-byte blocks, a blocking factor with a multiple of 512 bytes improves I/O access time. In the following example, the multiblock count (four) causes reads and writes to FILE-A to access a buffer area of four physical blocks:

```
FILE SECTION.
FD   FILE-A .
     BLOCK CONTAINS 2048 CHARACTERS
                         .
                         .
                         .
```

If you do not want to calculate the buffer size, but want to specify the number of records in each buffer, use the BLOCK CONTAINS n RECORDS clause. The following example specifies a buffer large enough to hold 15 records:

```
BLOCK CONTAINS 15 RECORDS
```

When using the BLOCK CONTAINS n RECORDS clause for sequential files on disk, RMS calculates the buffer size by using the maximum record unit size and rounding up to a multiple of 512 bytes. Consequently, the buffer could hold more records than you specify.

In the following example, the BLOCK CONTAINS clause specifies five records. RMS calculates the block size as eight records, or 512 bytes.

```
FILE SECTION.
FD  FILE-A
    BLOCK CONTAINS 5 RECORDS.
01  FILE-A-REC     PIC X(64).
                   .
                   .
                   .
```

20.4.2 Relative Files

I/O optimization of a relative file depends on four concepts:

- Maximum record number—The highest numbered record written to a relative file.
- Cell size—The unit of disk space needed to store a record unit size (record unit size = record + record overhead).

- Bucket size—The number of blocks read or written in one I/O operation (equivalent to buffer size). A bucket contains from 1 to 63 physical blocks.
- File size—The number of blocks used to preallocate the file.

20.4.2.1 Maximum Record Number (MRN)

If you create a relative file with a VAX COBOL program, the system sets the maximum record number (MRN) to 0, allowing the file to expand to any size.

If you create a relative file with the CREATE/FDL Utility, select a realistic MRN, since an attempt to insert a record with a number higher than the MRN will fail.

20.4.2.2 Cell Size

The system calculates cell size. (However, you can specify a different cell size when you create the file by using the RECORD CONTAINS clause in the file description.) You cannot write records larger than the specified cell size.

Avoid selecting a cell size larger than necessary since this wastes disk space. To optimize the packing of cells into buckets, cell size should be evenly divisible into bucket size.

The system calculates cell size using these formulas:

Fixed-length records:	**cell size = 1 + record size**
Variable-length records:	**cell size = 3 + record size**

For fixed-length records, the overhead byte is a record deletion indicator. Variable-length records use two additional overhead bytes to indicate record length. The following example calculates a cell size of 101 for fixed-length records:

```
FD   A-FILE
     RECORD CONTAINS 100 CHARACTERS
                        .
                        .
                        .
```

20.4.2.3 Bucket Size

A bucket's size is from 1 to 63 blocks. A large bucket improves sequential access to a relative file. You can prevent wasted space between the last cell and the end of a bucket by specifying a bucket size that is a multiple of cell size.

If you omit the BLOCK CONTAINS clause, the system calculates a bucket size large enough to hold at least one cell or 512 bytes, whichever is larger (that is, large enough to hold a record and its overhead bytes). Records cannot cross bucket boundaries, although they can cross block boundaries.

Use the BLOCK CONTAINS n CHARACTERS clause of the file description to set your own bucket size (in bytes per bucket). Consider the following example:

```
FILE-CONTROL.
    SELECT A-FILE
      ORGANIZATION IS RELATIVE.
         .
         .
         .
DATA DIVISION.
FILE SECTION.
FD   A-FILE
     RECORD CONTAINS 60 CHARACTERS
     BLOCK CONTAINS 1536 CHARACTERS
         .
         .
         .
```

In the preceding example, the bucket size is 3. Each bucket contains:

25 records (25 x 60)	= 1500 bytes
1 overhead byte per record (1 x 25)	= 25 bytes
11 bytes of wasted space	= 11 bytes
TOTAL	= 1536 bytes

If you use the BLOCK CONTAINS CHARACTERS clause and specify a value that is not a multiple of 512, RMS rounds the value to the next higher multiple of 512.

In the following example, the BLOCK CONTAINS clause specifies one record per bucket. Since the cell needs only 61 bytes, there are 451 wasted bytes in each bucket.

```
FILE-CONTROL.
    SELECT B-FILE
      ORGANIZATION IS RELATIVE.
         .
         .
         .
DATA DIVISION.
FILE SECTION.
FD  A-FILE
    RECORD CONTAINS 60 CHARACTERS
    BLOCK CONTAINS 1 RECORD.
         .
         .
         .
```

To improve I/O access time: (1) specify a small bucket size, and (2) use the BLOCK CONTAINS n RECORDS clause to specify the number of records (cells) in each bucket. This example creates buckets that contain eight records.

```
FD   A-FILE
     RECORD CONTAINS 60 CHARACTERS
     BLOCK CONTAINS 8 RECORDS.
         .
         .
         .
```

In the preceding example, the bucket size is one 512-byte block. Each bucket contains:

8 records (8 x 60)	= 480 bytes
1 overhead byte per record (1 x 8)	= 8 bytes
24 bytes of wasted space	= 24 bytes
TOTAL	= 512 bytes

20.4.2.4 File Size

Calculating a file's size helps you determine its space requirements. A file's size is a function of its bucket size. When you create a relative file, use these calculations to determine the number of blocks that you need:

$$file\ size\ (in\ blocks) = \frac{511 + (number\ of\ buckets * bytes\ per\ bucket)}{512}$$

$$number\ of\ buckets = \frac{number\ of\ records\ in\ the\ file}{number\ of\ cells\ per\ bucket}$$

Assume that you want to create a relative file able to hold 3,000 records. The records are 255 bytes long (plus 1 byte per record for overhead), with 4 cells to a bucket (BLOCK CONTAINS 4 RECORDS). To determine file size:

1. Calculate the number of buckets:

$$750 = \frac{3000}{4}$$

2. Calculate bucket size (see Section 20.4.2.3):

$$2 = \frac{4 * (1 + 255)}{512}$$

3. Calculate bytes per bucket = (bucket size * number of bytes in a block):

$$1024 = 2 * 512$$

4. Calculate file size:

$$1500 = \frac{511 + (750 * 1024)}{512}$$

$$file\ size = 1500\ physical\ blocks$$

To allocate the 1500 calculated blocks to populate the entire file, use the APPLY CONTIGUOUS-BEST-TRY PREALLOCATION clause; otherwise, allocate fewer blocks.

Before writing a record to a relative file, RMS must have formatted all buckets up to and including the bucket to contain the record. Each time bucket reformatting occurs, response time suffers. Therefore, writing the highest-numbered record first forces formatting of the entire file only once. However, this technique can waste disk space if the file is only partially loaded and not preallocated.

20.4.3 Indexed Files

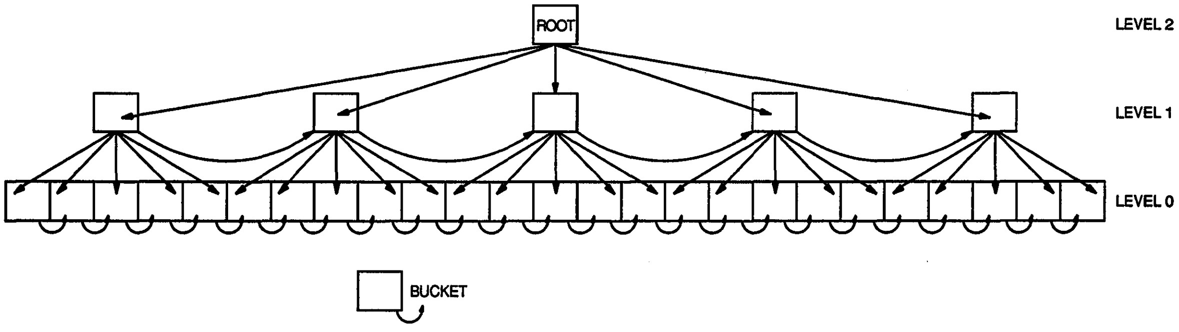
An indexed file contains data records and pointers to facilitate record access.

All data records and record pointers are stored in buckets. A bucket contains an integral number of contiguous, 512-byte blocks. The number of blocks is the bucket size.

Every indexed file must have a primary key, a field in the record description that contains a unique value for each record. When RMS writes records to the indexed file, it collates them according to increasing primary key value in a series of chained buckets. Thus, you can access the records sequentially by specifying ACCESS SEQUENTIAL.

As RMS writes records, it builds and maintains a tree-like structure of key-value and location pointers. The highest level of the index is a single bucket, called the root bucket. RMS scans one bucket at each level until it reaches the bottom, or data level. In a primary key index, this level contains actual data records. Buckets in each higher level, called index levels, contain index records. Successive levels of an index file are numbered. The data level is level zero. The number of levels above level zero is called the index depth. Figure 20–3 shows a 2-level primary index.

Figure 20–3: Two-Level Primary Index



ZK-1540-GE

RMS also builds an index for each alternate key that you define for the file. Like the primary index, alternate key indexes are contained in the file. However, they do not contain data records at the data level; instead, they contain pointers, or secondary index data records (SIDRs), to data records in the data level of the primary index.

Each random access request begins by comparing a key value to the root bucket's entries. It seeks the first root bucket entry whose key value equals or exceeds the value of the access request key. (This search is always successful, because the root bucket's highest key value is the highest possible value that the key field can contain.) Having located that key value, RMS uses its bucket pointer to bring the target bucket on the next lower level into memory. This process is repeated for each level of the index.

RMS searches one bucket at each level of the index until it reaches a target bucket at the data level. It then determines the data record's location, enabling it to retrieve or delete a record or write a new record. There can be no duplicate primary key values. If a record insertion causes a duplicate primary key value, the attempted write produces an exception condition.

A data level bucket may not be large enough to contain a new record. In this case, RMS inserts a new bucket in the chain, moving enough records from the old bucket to preserve the key value sequence. This is known as a bucket split.

20.4.3.1 Optimizing Indexed File I/O

I/O optimization of an indexed file depends on five concepts:

- Records—The size and format of the data records can affect the disk space used by the file.
- Keys—The number of keys and existence of duplicate key values can affect disk space and processing time.
- Buckets—Bucket size can affect disk space and processing time. Index depth and file activity can affect bucket size.
- Index depth—The depth of the index can affect bucket size and processing time.
- File size—The length of files affects space and access time.

Records

Variable-length records can save file space: you need write only the primary record key data item (plus alternate keys, if any) for each record. In contrast, fixed-length records require that all records be equal in length.

For example, assume that you are designing an employee master file. A variable-length record file lets you write a long record for a senior employee with a large amount of historical data, and a short record for a new employee with less historical data.

In the following example of a variable-length record description, integer 10 of the RECORD VARYING clause represents the length of the primary record key, while integer 80 describes the length of the longest record in A-FILE.

```
FILE-CONTROL.
    SELECT A-FILE ASSIGN TO "AMAST"
            ORGANIZATION IS INDEXED.
DATA DIVISION.
FILE SECTION.
FD  A-FILE
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS A-KEY
    RECORD VARYING FROM 10 TO 80 CHARACTERS.
01  A-REC.
    03  A-KEY          PIC X(10).
    03  A-REST-OF-REC  PIC X(70).
        .
        .
        .
```

Buckets must contain enough room for record insertion, or bucket splitting occurs. In this case, a new bucket contains records moved from the original one. For each record moved, a 7-byte pointer to the new record location remains in the original bucket. Thus, bucket splits can accumulate overhead and possibly reduce usable space so much that the original bucket can no longer receive records.

Record deletions can also accumulate storage overhead. However, most of the space is available for reuse. Because there can be no duplicate primary keys, RMS can reclaim all but 2 bytes of the deleted record space. This 2-byte field is a record deletion flag.

There are several ways to minimize overhead accumulation. First, determine or estimate the frequency of certain operations. For example, if you expect to add or delete 100 records of a 100,000-record file, your database is stable enough to allow some wasted space for record additions and deletions. However, if you expect frequent additions and deletions, try to:

- Choose a bucket size that allows for overhead accumulation, if possible. Avoid bucket sizes that are an exact or near multiple of your record size.
- Optimize record insertion by using the RMS DEFINE Utility to define the file with fill numbers; use the APPLY FILL-SIZE clause when loading the file.

Alternate Keys

Each alternate key requires the creation and maintenance of a separate index structure. The more keys you define, the longer each WRITE, REWRITE, and DELETE operation takes. (READ operations are not affected by multiple keys.)

If your application requires alternate keys, you can minimize I/O processing time if you avoid duplicate alternate keys. Duplicate keys can create long record pointer arrays, which fill bucket space and increase access time.

Bucket Size

Bucket size selection can influence indexed file performance.

To the system, bucket size is an integral number of physical blocks, each 512 bytes long. Thus, a bucket size of 1 specifies a 512-byte bucket, while a bucket size of 2 specifies a 1024-byte bucket, and so on.

The VAX COBOL compiler passes bucket size values to RMS based on what you specify in the BLOCK CONTAINS clause. In this case, you express bucket size in terms of records or characters.

If you specify block size in records, the bucket can contain more records than you specify, but never fewer. For example, assume that your file contains fixed-length, 100-byte records, and you want each bucket to contain five records, as follows:

```
BLOCK CONTAINS 5 RECORDS
```

This appears to define a bucket as a 512-byte block, containing five records of 100 bytes each. However, the compiler adds RMS record and bucket overhead to each bucket, as follows:

Bucket overhead	**= 15 bytes per bucket**
Record overhead	**= 7 bytes per record (fixed-length)** **9 bytes per record (variable-length)**

Thus, in this example, the bucket size calculation is:

Bucket overhead	= 15 bytes	
Total record space is (100 + 7) * 5	= 535 bytes	(Record size is 100 bytes, record overhead is 7 bytes for each of 5 records)
TOTAL	= 550 bytes	

Because blocks are 512 bytes long, and buckets are always an integral number of blocks, the smallest bucket size possible (the system default) in this case is two blocks. The system, however, puts in as many records as fit into each bucket. Thus, the bucket actually contains nine records, not five.

The CHARACTERS option of the BLOCK CONTAINS clause lets you specify bucket size more directly. For example:

```
BLOCK CONTAINS 2048 CHARACTERS
```

This specifies a bucket size of four 512-byte blocks. The number of characters in a bucket is always a multiple of 512. If not, RMS rounds it to the next higher multiple of 512.

Index Depth

The length of data records, key fields, and buckets in the file determines the depth of the index. Index depth, in turn, determines the number of disk accesses needed to retrieve a record. The smaller the index depth, the better the performance. In general, an index depth of 3 or 4 gives satisfactory performance. If your calculated index depth is greater than 4, you should consider redesigning the file.

You can optimize your file's index depth after you have determined file, record, and key size. Calculating index depth is an iterative process, with bucket size as the variable. Keep in mind that the highest level (root level) can contain only one bucket. An example of index depth calculation follows the information in File Status calculation:

File Size

When you calculate file size:

- Every bucket in an indexed file contains 15 bytes of overhead.
- Every bucket in an indexed file contains records. Only record type and size differ.
- Data records are only in level 0 buckets of the primary index.
- Index records are in level 1 and higher-numbered buckets.
- If you use alternate keys, secondary index data records (SIDRs) are only in level 0 buckets of alternate indexes.

Use these calculations to determine data and index record size:

- **Data records:**

$$Fixed - length\ record\ size = actual\ record\ size + 7$$

$$Variable - length\ record\ size = actual\ record\ size + 9$$

- **Index records:**

$$Record\ size = key\ size + 3$$

If a file has more than 65,536 blocks, the 3-byte index record overhead could increase to 5 bytes.

Use these calculations to determine SIDR record length:

- **No duplicates allowed:**

$$Record\ size = key\ size + 9$$

- **Duplicates allowed:**

$$Record\ size = key\ size + 8 + 5 * (number\ of\ duplicate\ records)$$

NOTE

> Bucket packing efficiency determines how well bucket space is used. A packing efficiency of 1 means the buckets of an index are full. A packing efficiency of .5 means that, on the average, the buckets are half full.

Consider an indexed file with these attributes:

- 100,000 fixed-length records of 200 characters each
- Primary key = 20 characters
- Alternate key = 8 characters, no duplicates allowed
- Bucket size = 3 (an arbitrary value)
- No fill number

Primary key index level calculations:

Level 0 (data level buckets):

$$data\ records\ per\ bucket = \frac{bytes\ per\ bucket - 15}{record\ size + 7}$$

$$= \frac{1536 - 15}{200 + 7} = 7\ data\ records\ per\ bucket$$

$$number\ of\ data\ buckets = \frac{number\ of\ data\ records}{records\ per\ bucket}$$

$$= \frac{100,000}{7} = 14,286\ buckets\ to\ contain\ all\ data\ records.$$

Level 1 (index buckets):

$$index\ records\ per\ bucket = \frac{bytes\ per\ bucket - 15}{key\ size + 3}$$

$$= \frac{1536 - 15}{20 + 3} = 66\ index\ records\ per\ bucket$$

$$number\ of\ index\ buckets = \frac{no.\ of\ buckets\ from\ level\ n - 1}{index\ records\ per\ bucket}$$

$$= \frac{14,286}{66} = 216\ level\ 1\ buckets\ to\ address\ all\ data\ buckets\ at\ level\ 0$$

Continue calculating index depth until you reach the root level—that is, when the number of buckets needed to address the buckets from the previous level equals 1.

Level 2 (index buckets):

$$number\ of\ buckets = \frac{216}{66} = 4\ level\ 2\ buckets\ to\ address\ all\ level\ 1\ buckets$$

Level 3 (index buckets):

$$number\ of\ buckets = \frac{4}{66} = 1\ level\ 3\ bucket\ to\ address\ all$$

$$level\ 2\ buckets\ (Level\ 3\ is\ the\ root\ bucket\ for\ the\ primary\ index.)$$

20.4.3.2 Calculating Key Index Levels

If you allow duplicate keys in alternate indexes, the number and size of SIDRs depend on the number of duplicate key values in the file. (For duplicate key alternate index calculations, see the RMS reference documentation.) Since alternate index records are usually inserted in random order, the bucket packing efficiency ranges from about .5 to about .65. The following example uses an average efficiency of .55.

Level 0 (data level buckets—no duplicate alternate keys):

$$SIDRs\ per\ bucket = \frac{bytes\ per\ bucket - 15}{key\ size + 9}$$

$$= \frac{1536 - 15}{8 + 9} = 89\ SIDRs\ per\ bucket$$

$$number\ of\ buckets = \frac{number\ of\ records}{records\ per\ bucket}$$

$$= \frac{100,000}{89} = 1123\ level\ 0\ alternate\ index\ buckets$$

Level 1 (index buckets):

$$records\ per\ bucket = \frac{1536 - 15}{8 + 3} = 138\ index\ records\ per\ bucket\ number\ of\ buckets$$

$$= \frac{1123}{138} = 9\ level\ 1\ buckets\ to\ address\ data\ buckets\ (SIDRs)\ at\ level\ 0$$

Level 2 (index buckets):

$$number\ of\ buckets = \frac{9}{138} = 1\ level\ 2\ bucket\ to\ address\ data\ buckets$$

$$at\ level\ 1\ (level\ 2\ is\ the\ root\ level)$$

20.4.3.3 Caching Index Roots

The system requires at least two buffers to process an indexed file: one for a data bucket, the other for an index bucket. Each buffer is large enough to contain a single bucket. If your program does not contain a RESERVE n AREAS clause, or if you do not use the SET RMS_DEFAULT DCL command, the system sets the default.

A RESERVE n AREAS clause creates additional buffers for processing an indexed file. At run time, the system retains (caches) in memory the roots of one or more indexes of the file. Random access to any record through that index requires one less I/O operation.

You can also use the SET RMS_DEFAULT/BUFFER_COUNT=count to create additional buffers.

The following rules apply for caching index roots:

- Allocate one buffer for each key that your program uses to access file records, in addition to the two required buffers. For example, if the file contains a primary key and two alternate keys, and you use all these keys to access records, allocate a total of five buffers. If you use only one key, you need only one additional buffer area, or a total of three.

- Use the RESERVE n AREAS clause to obtain allocation, where n is two more than the number of distinct keys used for access. For example, the RESERVE 5 AREAS clause allocates two required buffers, plus three buffer areas for caching the roots of three distinct file access keys.
- Use the SET RMS_DEFAULT/BUFFER_COUNT=count DCL command if you want to allocate buffers without specifying the RESERVE AREA clause in your program, or for buffer allocation on a process or system-wide basis.

The SET RMS DCL commands also apply to sequential and relative files. The SET RMS DCL commands and RESERVE AREA clause provide the same functionality.

Appendix A

Compiler Implementation Limitations

The following list summarizes the VAX COBOL compiler's limitations and restrictions. The compiler issues diagnostic messages whenever you exceed its limits.

1. Run-time storage (generated object code and data) for COBOL programs cannot exceed 2,147,483,647 bytes.
2. The length of an FD's record cannot exceed 32,767 bytes for a sequential file, 32,234 bytes for an indexed file, or 32,255 bytes for a relative file. For SD records the length cannot exceed 32,759 bytes for a sequential file, 32,226 bytes for an indexed file, or 32,247 bytes for a relative file.
3. Bucket size for relative and indexed files cannot be greater than 63.
4. A sequential disk file's multiblock count cannot be greater than 127.
5. The physical block size for a sequential tape file must be from 20 to 65,532 bytes, inclusive.
6. Run-time storage for an indexed file's RECORD KEY or ALTERNATE RECORD KEY data item must not be greater than 255 bytes.
7. The maximum number of EXTERNAL file connectors and/or record definitions is 248. The compiler implements each EXTERNAL file connector and record definition as a separate PSECT, with the EXTERNAL file name and record name serving as the PSECT name. Each index name within an EXTERNAL record is implemented as a separate PSECT.
8. The number of indexed file RECORD KEY and ALTERNATE RECORD KEY data items must not exceed 255.
9. The number of literal phrases specified to define an alphabet in an ALPHABET clause of the SPECIAL-NAMES paragraph must not be greater than 256.
10. The value of a numeric literal in a literal phrase of an ALPHABET clause must not be greater than 255.
11. The value of a switch number in the SWITCH clause of the SPECIAL-NAMES paragraph must be from 1 to 16, inclusive.
12. The value of a numeric literal in the SYMBOLIC CHARACTERS clause must be from 1 to 256, inclusive.
13. The value of an integer in the EXTENSION option of the APPLY clause must be from 0 to 65,535, inclusive.
14. The value of an integer in the WINDOW option of the APPLY clause must be from 0 to 127, inclusive, or equal to 255.

15. The value of the integer in the RESERVE AREAS clause must not be greater than 127.
16. If a data item is allocated more than 65,535 bytes, a COBOL program cannot reference it.
17. Alphanumeric or numeric edited picture character-strings cannot represent more than 255 standard data format characters.
18. Alphanumeric or alphabetic picture character-strings cannot represent more than 65,535 standard data format characters.
19. A nonnumeric literal cannot be greater than 256 characters.
20. A hexadecimal literal cannot be greater than 256 hexadecimal digits.
21. A PICTURE character-string cannot contain more than 256 characters.
22. The number of operands in a single DISPLAY statement cannot be greater than 254.
23. The number of operands in the USING phrase of a CALL statement cannot be greater than 255.
24. The number of USING files in a SORT or MERGE statement cannot exceed 10.
25. The maximum number of characters in a subschema pathname specification is 256.
26. The maximum static nesting depth of contained programs is 256.
27. The maximum number of characters in a user-word in VAX COBOL is 31. The maximum number of characters allowed in a user-word as defined by the ANSI COBOL standard is 30. The compiler issues an informational diagnostic if you use 31-character user-words. The maximum number of characters in an external report file name is 30.
28. The maximum number of strings associated with the /AUDIT compile qualifier is 64.
29. The maximum number of characters in a CDD/Plus pathname specification is 256.
30. The maximum number of levels in a database subschema record definition supported by VAX COBOL is 49.
31. The maximum number of digits in a numeric database data item supported by VAX COBOL is 18.
32. The maximum number of standard data format characters in a character-type database data item is 65,535.
33. If a file is assigned to a magtape media and you use the BLOCK CONTAINS clause in the associated file description, the number of characters in a physical block determined from the BLOCK CONTAINS clause must be an even multiple of 4.
34. If a file is assigned to a disk medium and you use the BLOCK CONTAINS clause in the associated file description, the BLOCK CONTAINS value must be an even multiple of 512.
35. The maximum number of lines in any report file is 999,998,000,001.
36. The maximum subscript value for any subscript or index name is 2,147,483,647.

37. In the OCCURS n TIMES clause of a Data Description entry, the maximum allowable value for n is 2,147,483,647.

38. The maximum static scoping depth of file-specific USE procedures is 82.

39. The maximum static scoping depth of database USE procedures is 84.

40. The maximum number of operands in a given COBOL DML statement is 255.

41. In a PERFORM n TIMES statement, the maximum allowable value for n is 2,147,483,647.

42. The maximum static nesting depth of nested IF statements is 64.

43. The maximum number of levels for subscripts is 48.

44. The maximum number of files in a MULTIPLE FILE TAPE clause is 255.

Appendix B

Error Messages

This appendix discusses run-time errors, program run errors, and run-time I/O errors. Also included in this appendix is a partial list of compiler messages returned by the VAX COBOL compiler.

B.1 Run-Time Errors

Faulty program logic can cause abnormal termination. If errors occur at run time, the Run-Time Library (RTL) displays a message with the same general format as system error messages. In addition, the system TRACEBACK facility displays a list of routines active when the error occurred.

The following DCL command displays a list of COBOL run-time errors:

```
$  HELP COBOL ERRORS
```

You can then access help for any of the run-time errors displayed.

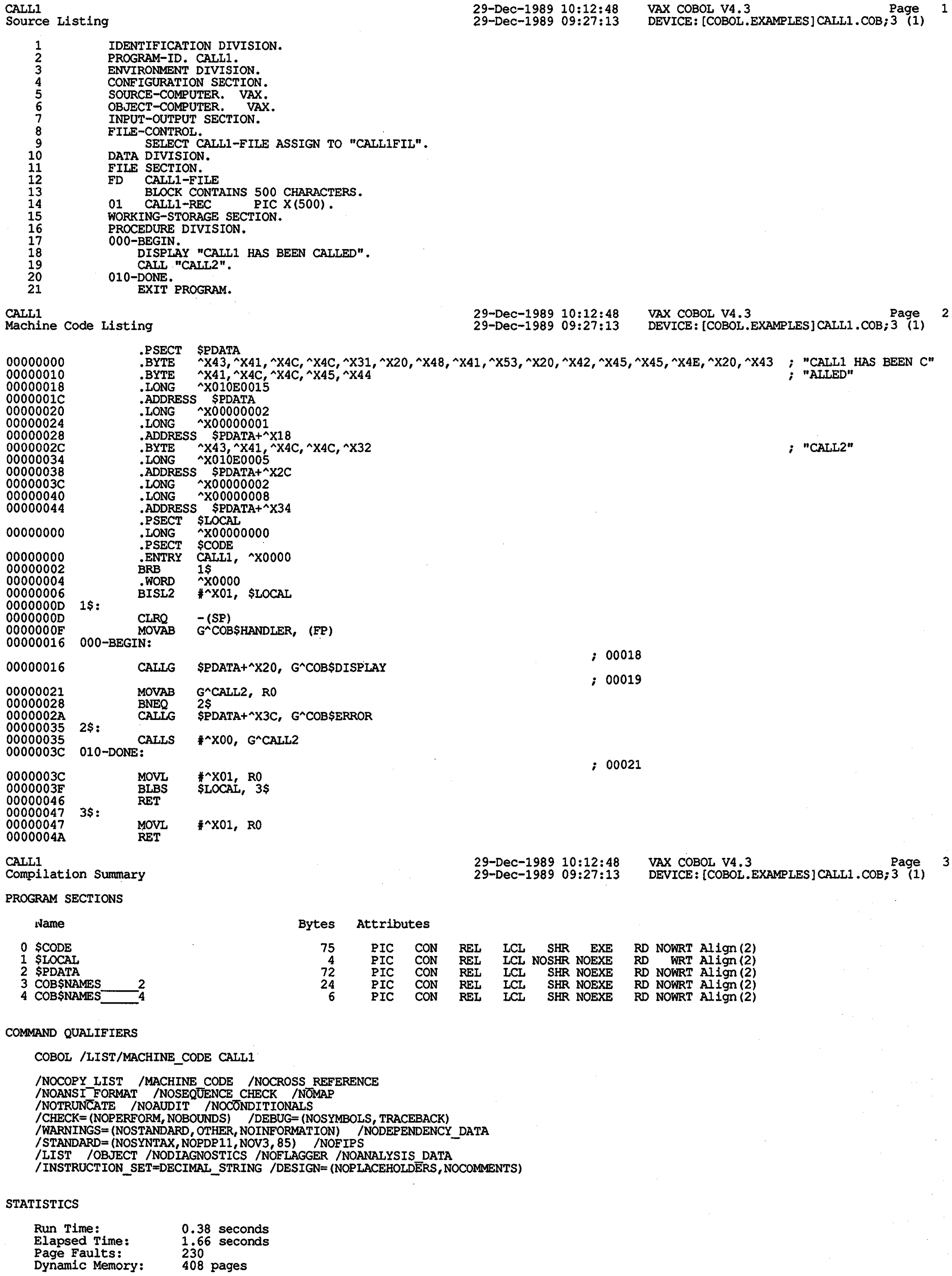
B.1.1 Sample Run-Time Error

Notice that line 45 in program TRBLE in Figure 2–2 calls subprogram CALL1. CALL1 in turn calls CALL2. (CALL1 and CALL2 are listed in Figure B–1 and Figure B–2, respectively.) Program CALL2 has a logic error at line 25 where it attempts to add the alphanumeric contents of data-item ALPHA to the numeric data-item NUMA. Running program TRBLE produces these results:

```
$ RUN TRBLE
CALL 1 HAS BEEN CALLED
CALL 2 HAS BEEN CALLED
%COB-F-INVDECDAT, invalid decimal data
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name     line  relative PC  absolute PC

CALL2           CALL2              24     00000035     0000145D
CALL1           CALL1              19     0000003C     00001418
TRBLE           TRBLE              43     00000185     00001185
```

When an error occurs, TRACEBACK produces a symbolic dump of the active call frames. A call frame represents one execution of a routine. For each call frame, TRACEBACK displays the following information: (1) the module name, (program-id), (2) the routine name (program-id), (3) the source listing line number where the error or CALL occurred, and (4) program-counter (PC) information.

Figure B–1: Listing of Program CALL1



```
CALL1                                                            29-Dec-1989 10:12:48     VAX COBOL V4.3                  Page   1
Source Listing                                                   29-Dec-1989 09:27:13     DEVICE:[COBOL.EXAMPLES]CALL1.COB;3 (1)

     1         IDENTIFICATION DIVISION.
     2         PROGRAM-ID. CALL1.
     3         ENVIRONMENT DIVISION.
     4         CONFIGURATION SECTION.
     5         SOURCE-COMPUTER.  VAX.
     6         OBJECT-COMPUTER.   VAX.
     7         INPUT-OUTPUT SECTION.
     8         FILE-CONTROL.
     9              SELECT CALL1-FILE ASSIGN TO "CALL1FIL".
    10         DATA DIVISION.
    11         FILE SECTION.
    12         FD   CALL1-FILE
    13              BLOCK CONTAINS 500 CHARACTERS.
    14         01   CALL1-REC       PIC X(500).
    15         WORKING-STORAGE SECTION.
    16         PROCEDURE DIVISION.
    17         000-BEGIN.
    18             DISPLAY "CALL1 HAS BEEN CALLED".
    19             CALL "CALL2".
    20         010-DONE.
    21             EXIT PROGRAM.
CALL1                                                            29-Dec-1989 10:12:48     VAX COBOL V4.3                  Page   2
Machine Code Listing                                             29-Dec-1989 09:27:13     DEVICE:[COBOL.EXAMPLES]CALL1.COB;3 (1)

                    .PSECT  $PDATA
00000000            .BYTE   ^X43,^X41,^X4C,^X4C,^X31,^X20,^X48,^X41,^X53,^X20,^X42,^X45,^X45,^X4E,^X20,^X43  ; "CALL1 HAS BEEN C"
00000010            .BYTE   ^X41,^X4C,^X4C,^X45,^X44                                                          ; "ALLED"
00000018            .LONG   ^X010E0015
0000001C            .ADDRESS  $PDATA
00000020            .LONG   ^X00000002
00000024            .LONG   ^X00000001
00000028            .ADDRESS  $PDATA+^X18
0000002C            .BYTE   ^X43,^X41,^X4C,^X4C,^X32                                                          ; "CALL2"
00000034            .LONG   ^X010E0005
00000038            .ADDRESS  $PDATA+^X2C
0000003C            .LONG   ^X00000002
00000040            .LONG   ^X00000008
00000044            .ADDRESS  $PDATA+^X34
                    .PSECT  $LOCAL
00000000            .LONG   ^X00000000
                    .PSECT  $CODE
00000000            .ENTRY  CALL1, ^X0000
00000002            BRB     1$
00000004            .WORD   ^X0000
00000006            BISL2   #^X01, $LOCAL
0000000D   1$:
0000000D            CLRQ    -(SP)
0000000F            MOVAB   G^COB$HANDLER, (FP)
00000016   000-BEGIN:
                                                             ; 00018
00000016            CALLG   $PDATA+^X20, G^COB$DISPLAY
                                                             ; 00019
00000021            MOVAB   G^CALL2, R0
00000028            BNEQ    2$
0000002A            CALLG   $PDATA+^X3C, G^COB$ERROR
00000035   2$:
00000035            CALLS   #^X00, G^CALL2
0000003C   010-DONE:
                                                             ; 00021
0000003C            MOVL    #^X01, R0
0000003F            BLBS    $LOCAL, 3$
00000046            RET
00000047   3$:
00000047            MOVL    #^X01, R0
0000004A            RET
CALL1                                                            29-Dec-1989 10:12:48     VAX COBOL V4.3                  Page   3
Compilation Summary                                              29-Dec-1989 09:27:13     DEVICE:[COBOL.EXAMPLES]CALL1.COB;3 (1)

PROGRAM SECTIONS

    Name                                     Bytes   Attributes

  0 $CODE                                       75     PIC    CON    REL    LCL    SHR   EXE    RD NOWRT Align(2)
  1 $LOCAL                                       4     PIC    CON    REL    LCL NOSHR NOEXE    RD   WRT Align(2)
  2 $PDATA                                      72     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)
  3 COB$NAMES_____2                             24     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)
  4 COB$NAMES_____4                              6     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)

COMMAND QUALIFIERS

    COBOL /LIST/MACHINE_CODE CALL1

    /NOCOPY_LIST  /MACHINE_CODE  /NOCROSS_REFERENCE
    /NOANSI_FORMAT  /NOSEQUENCE_CHECK  /NOMAP
    /NOTRUNCATE  /NOAUDIT  /NOCONDITIONALS
    /CHECK=(NOPERFORM,NOBOUNDS)  /DEBUG=(NOSYMBOLS,TRACEBACK)
    /WARNINGS=(NOSTANDARD,OTHER,NOINFORMATION)  /NODEPENDENCY_DATA
    /STANDARD=(NOSYNTAX,NOPDP11,NOV3,85)  /NOFIPS
    /LIST  /OBJECT /NODIAGNOSTICS /NOFLAGGER /NOANALYSIS_DATA
    /INSTRUCTION_SET=DECIMAL_STRING /DESIGN=(NOPLACEHOLDERS,NOCOMMENTS)

STATISTICS

    Run Time:           0.38 seconds
    Elapsed Time:       1.66 seconds
    Page Faults:        230
    Dynamic Memory:     408 pages
```

ZK-6447-GE

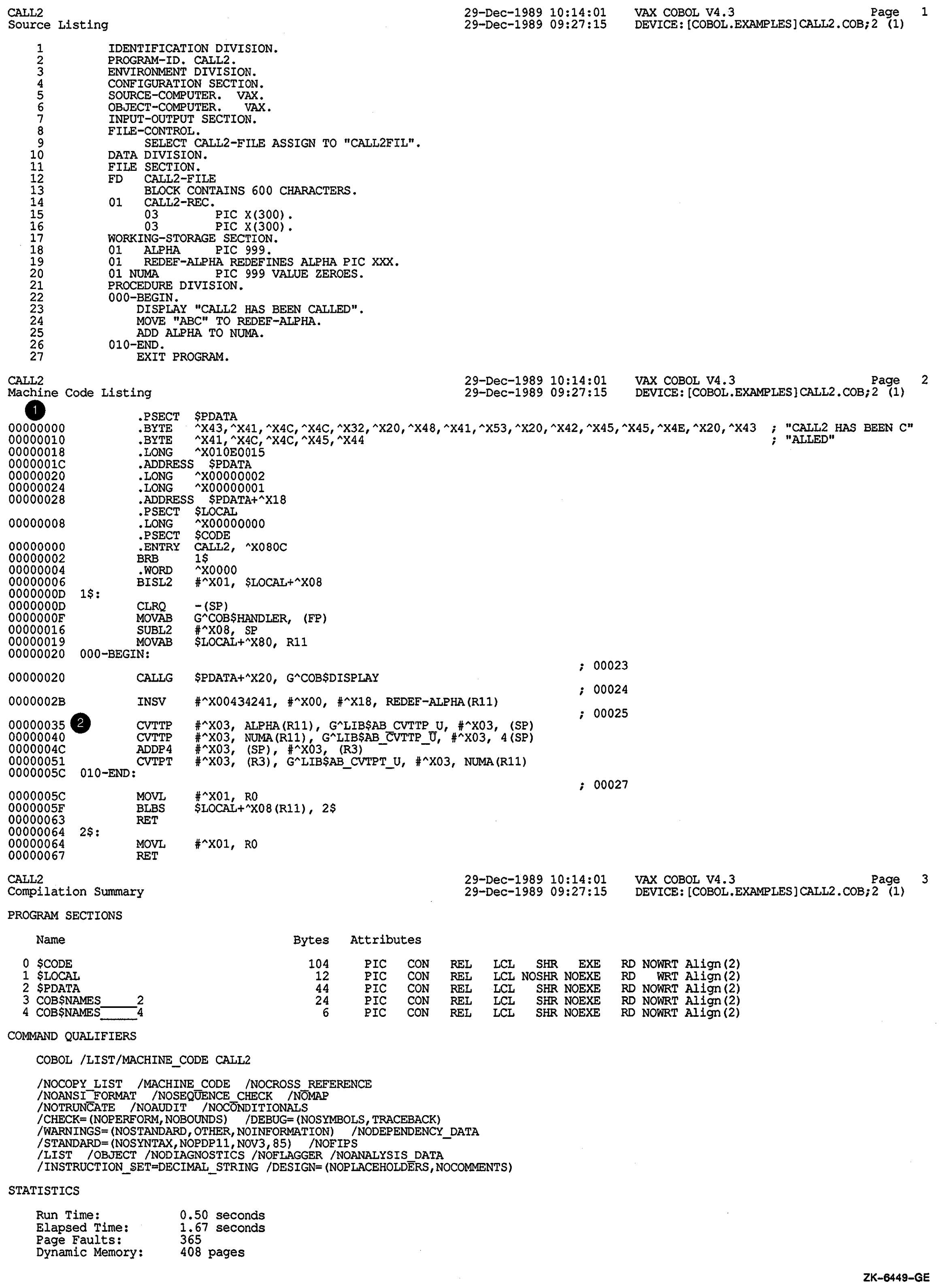
From the TRACEBACK information you can determine that the last line to execute successfully was line 24 of the program module CALL2 (labeled ❶ in Figure B–2).

The TRACEBACK information also gives you the relative and absolute PC of the instruction (labeled ❷ in Figure B–2). Relative PC is the location in memory containing the instruction that failed to execute. If you specify the /LIST and /MACHINE_CODE qualifiers when you compile the program, the listing shows relative PC 00000035 is in the .PSECT $CODE area of the machine code listing. Absolute PC is the absolute memory address of the instruction that failed.

You can also use the VMS Debugger to examine the machine code instruction. To do this, compile and link the program using the /DEBUG qualifier. When you run the program, you automatically enter the debugger. Once in the debugger, you could use the EXAMINE/INSTRUCTION command to examine the contents of the failed instruction. You could also use the debugger in screen mode, which would indicate where the error occurred. For more information on the debugger, refer to Chapter 3 and the VMS Debugger documentation.

```
EXAMINE/INSTRUCTION ^X00001322
```

Figure B–2: Listing of Program CALL2



```
CALL2                                                           29-Dec-1989 10:14:01    VAX COBOL V4.3                 Page   1
Source Listing                                                  29-Dec-1989 09:27:15    DEVICE:[COBOL.EXAMPLES]CALL2.COB;2 (1)

    1         IDENTIFICATION DIVISION.
    2         PROGRAM-ID. CALL2.
    3         ENVIRONMENT DIVISION.
    4         CONFIGURATION SECTION.
    5         SOURCE-COMPUTER.  VAX.
    6         OBJECT-COMPUTER.   VAX.
    7         INPUT-OUTPUT SECTION.
    8         FILE-CONTROL.
    9              SELECT CALL2-FILE ASSIGN TO "CALL2FIL".
   10         DATA DIVISION.
   11         FILE SECTION.
   12         FD   CALL2-FILE
   13              BLOCK CONTAINS 600 CHARACTERS.
   14         01   CALL2-REC.
   15              03        PIC X(300).
   16              03        PIC X(300).
   17         WORKING-STORAGE SECTION.
   18         01   ALPHA     PIC 999.
   19         01   REDEF-ALPHA REDEFINES ALPHA PIC XXX.
   20         01 NUMA        PIC 999 VALUE ZEROES.
   21         PROCEDURE DIVISION.
   22         000-BEGIN.
   23             DISPLAY "CALL2 HAS BEEN CALLED".
   24             MOVE "ABC" TO REDEF-ALPHA.
   25             ADD ALPHA TO NUMA.
   26         010-END.
   27             EXIT PROGRAM.

CALL2                                                           29-Dec-1989 10:14:01    VAX COBOL V4.3                 Page   2
Machine Code Listing                                            29-Dec-1989 09:27:15    DEVICE:[COBOL.EXAMPLES]CALL2.COB;2 (1)
   ❶
                    .PSECT  $PDATA
00000000            .BYTE   ^X43,^X41,^X4C,^X4C,^X32,^X20,^X48,^X41,^X53,^X20,^X42,^X45,^X45,^X4E,^X20,^X43  ; "CALL2 HAS BEEN C"
00000010            .BYTE   ^X41,^X4C,^X4C,^X45,^X44                                                         ; "ALLED"
00000018            .LONG   ^X010E0015
0000001C            .ADDRESS  $PDATA
00000020            .LONG   ^X00000002
00000024            .LONG   ^X00000001
00000028            .ADDRESS  $PDATA+^X18
                    .PSECT  $LOCAL
00000008            .LONG   ^X00000000
                    .PSECT  $CODE
00000000            .ENTRY  CALL2, ^X080C
00000002            BRB     1$
00000004            .WORD   ^X0000
00000006            BISL2   #^X01, $LOCAL+^X08
0000000D  1$:
0000000D            CLRQ    -(SP)
0000000F            MOVAB   G^COB$HANDLER, (FP)
00000016            SUBL2   #^X08, SP
00000019            MOVAB   $LOCAL+^X80, R11
00000020  000-BEGIN:
                                                                        ; 00023
00000020            CALLG   $PDATA+^X20, G^COB$DISPLAY
                                                                        ; 00024
0000002B            INSV    #^X00434241, #^X00, #^X18, REDEF-ALPHA(R11)
                                                                        ; 00025
00000035 ❷          CVTTP   #^X03, ALPHA(R11), G^LIB$AB_CVTTP_U, #^X03, (SP)
00000040            CVTTP   #^X03, NUMA(R11), G^LIB$AB_CVTTP_U, #^X03, 4(SP)
0000004C            ADDP4   #^X03, (SP), #^X03, (R3)
00000051            CVTPT   #^X03, (R3), G^LIB$AB_CVTPT_U, #^X03, NUMA(R11)
0000005C  010-END:
                                                                        ; 00027
0000005C            MOVL    #^X01, R0
0000005F            BLBS    $LOCAL+^X08(R11), 2$
00000063            RET
00000064  2$:
00000064            MOVL    #^X01, R0
00000067            RET

CALL2                                                           29-Dec-1989 10:14:01    VAX COBOL V4.3                 Page   3
Compilation Summary                                             29-Dec-1989 09:27:15    DEVICE:[COBOL.EXAMPLES]CALL2.COB;2 (1)

PROGRAM SECTIONS

    Name                                 Bytes   Attributes

  0 $CODE                                  104     PIC    CON    REL    LCL   SHR    EXE    RD NOWRT Align(2)
  1 $LOCAL                                  12     PIC    CON    REL    LCL NOSHR NOEXE    RD   WRT Align(2)
  2 $PDATA                                  44     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)
  3 COB$NAMES_____2                         24     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)
  4 COB$NAMES_____4                          6     PIC    CON    REL    LCL   SHR NOEXE    RD NOWRT Align(2)

COMMAND QUALIFIERS

    COBOL /LIST/MACHINE_CODE CALL2

    /NOCOPY_LIST  /MACHINE_CODE  /NOCROSS_REFERENCE
    /NOANSI_FORMAT  /NOSEQUENCE_CHECK  /NOMAP
    /NOTRUNCATE  /NOAUDIT  /NOCONDITIONALS
    /CHECK=(NOPERFORM,NOBOUNDS)  /DEBUG=(NOSYMBOLS,TRACEBACK)
    /WARNINGS=(NOSTANDARD,OTHER,NOINFORMATION)  /NODEPENDENCY_DATA
    /STANDARD=(NOSYNTAX,NOPDP11,NOV3,85)  /NOFIPS
    /LIST  /OBJECT /NODIAGNOSTICS /NOFLAGGER /NOANALYSIS_DATA
    /INSTRUCTION_SET=DECIMAL_STRING /DESIGN=(NOPLACEHOLDERS,NOCOMMENTS)

STATISTICS

    Run Time:           0.50 seconds
    Elapsed Time:       1.67 seconds
    Page Faults:        365
    Dynamic Memory:     408 pages
```

ZK-6449-GE

B.2 Program Run Errors

Incorrect or undesirable program results are usually caused by data errors or program logic errors. You can resolve most of these errors by desk-checking your program and by using the VMS Debugger.

B.2.1 Faulty Data

Faulty or incorrectly defined data often produce incorrect results. Data errors can sometimes be attributed to:

- Incorrect picture size. If the picture size of a receiving data item is too small, data may be truncated.

```
77   COUNTER   PIC 9.
     .
     .
     .
PROCEDURE DIVISION.
     .
     .
     .
     LOOP.
         ADD 1 TO COUNTER
         IF COUNTER < 10 GO TO LOOP.
```

- Incorrect file definition. The block size specified when accessing a file should be the same block size used when creating the file.
- Incorrect record field position. The record field positions that you specify in your program may not agree with a file's record field positions. For example, a file could have this record description:

```
01  PAY-RECORD.
    03  P-NUMBER       PIC X(5).
    03  P-WEEKLY-AMT   PIC S9(5)V99  COMP-3.
    03  P-MONTHLY-AMT  PIC S9(5)V99  COMP-3.
    03  P-YEARLY-AMT   PIC S9(5)V99  COMP-3.
        .
        .
        .
```

Incorrectly positioning these fields can produce faulty data.

In the following example, a program references the file incorrectly. The field described as P-YEARLY-AMT actually contains P-MONTHLY-AMT data, and vice versa.

```
01  PAY-RECORD.
    03  P-NUMBER       PIC X(5).
    03  P-WEEKLY-AMT   PIC S9(5)V99  COMP-3.
    03  P-YEARLY-AMT   PIC S9(5)V99  COMP-3.
    03  P-MONTHLY-AMT  PIC S9(5)V99  COMP-3.
        .
        .
        .
PROCEDURE DIVISION.
ADD-TOTALS.
    ADD P-MONTHLY-AMT TO TOTAL-MONTHLY-AMT.
        .
        .
        .
```

You can minimize file definition and record field position errors by writing your file and record descriptions in a library file and then using the COPY or COPY FROM DICTIONARY statement in your programs.

Your choice of test data can minimize faulty data problems. Rather than using actual or ideal data, use test files that include data extremes.

Determining when a program produces incorrect results can often help your debugging effort. You can do this by maintaining audit counts (such as total master in = nnn, total transactions in = nnn, total deletions = nnn, total master out = nnn) and displaying the audit counts when the program ends. Conditional compilation lines (explained in this manual) used in your source program can also help your debugging effort.

B.2.2 Program Logic Errors

When checking your program for logic errors, first examine your program for some of the more obvious bugs, such as the following:

- Hidden periods. Periods inadvertently placed in a statement usually produce unexpected results. For example:

```
050-DO-WEEKLY-TOTALS.
    IF W-CODE = "W"
       PERFORM 100-WEEKLY-SUMMARY
       ADD WEEKLY-AMT TO WEEKLY-TOTALS.
       GO TO 000-READ-A-MASTER.
    WRITE NEW-MASTER-REC.
```

  The period at the end of ADD WEEKLY-AMT TO WEEKLY-TOTALS terminates the scope of the IF statement. The GO TO is not within the IF scope and will always be executed. This changes the logic of the statement by transforming GO TO 000-READ-A-MASTER from a conditional to an unconditional GO TO. In addition, the statement following the GO TO will never be executed.

- Tests for equality, rather than inequality. Executing a procedure until a test condition is met can cause errors:

```
PERFORM ABC-ROUTINE UNTIL A-COUNTER = 10.
```

  If, during execution, the program increments A-COUNTER by a value other than 1 (2 or 1.5, for example), A-COUNTER might never equal 10, causing a loop in ABC-ROUTINE. You can prevent this type of error by changing the statement to:

```
PERFORM ABC-ROUTINE UNTIL A-COUNTER > 9
```

- Two negative test conditions combined with an OR. The object of the following statement is to execute GO TO 200-PRINT-REPORT when TEST-FIELD contains other than an A or B. However, the GO TO always executes because no matter what TEST-FIELD contains, one of the conditions is always true.

```
IF TEST-FIELD NOT = "A" OR NOT = "B"
   GO TO 200-PRINT-REPORT.
   .
   .
   .
```

You can correct this logic error by changing the statement to:

```
IF TEST-FIELD NOT = "A" AND NOT = "B"
   GO TO 200-PRINT-REPORT.
   .
   .
   .
```

B.3 Run-Time Input/Output Errors

An I/O error is a condition that causes an I/O statement to fail. I/O errors are detected by the VAX Record Management Services (RMS) or the VAX Run-Time Library (RTL). You can use the RMS special registers, RMS-STS, RMS-STV, and RMS-FILENAME as well as RMS-CURRENT-STS, RMS-CURRENT-STV, and RMS-CURRENT-FILENAME, to detect errors. Example B–1 and Example B–2 show how you can use RMS special registers to detect errors.

Example B–1: Using RMS Special Registers to Detect Errors

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RMS-SPEC-REGISTERS.
*
* This program demonstrates the use of RMS special registers to
* implement a different recovery for RMS file errors
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT OPTIONAL EMP-FILE
    ASSIGN TO "SYS$DISK:FOO.DAT".
SELECT REPORT-FILE
    ASSIGN TO "SYS$OUTPUT".
DATA DIVISION.
FILE SECTION.
FD      EMP-FILE
        VALUE OF ID IS VAL-OF-ID.
01      EMP-RECORD.
        02       EMP-ID       PIC 9(7).
        02       EMP-NAME     PIC X(15).
        02       EMP-ADDRESS  PIC X(30).
```

(continued on next page)

Example B–1 (Cont.): Using RMS Special Registers to Detect Errors

```
FD      REPORT-FILE          REPORT IS RPT.
WORKING-STORAGE SECTION.
01      VAL-OF-ID            PIC X(20).
01      END-OF-FILE          PIC S9(9)  COMP VALUE EXTERNAL RMS$_EOF.
01      BADNAME              PIC S9(9)  COMP VALUE EXTERNAL SS$_BADFILENAME.
01      FILE-NOT-FOUND       PIC S9(9)  COMP VALUE EXTERNAL RMS$_FNF.
01      DIR-NOT-FOUND        PIC S9(9)  COMP VALUE EXTERNAL RMS$_DNF.
01      INV-DEVICE           PIC S9(9)  COMP VALUE EXTERNAL RMS$_DEV.
01      INV-FILE-ID          PIC S9(9)  COMP VALUE EXTERNAL RMS$_IFI.
01      RMS-ERR              PIC S9(9)  COMP VALUE EXTERNAL SHR$_RMSERROR.
01      D-DATE               PIC 9(6).
01      EOF-SW               PIC X.
        88       E-O-F                  VALUE "E".
        88       NOT-E-O-F              VALUE "N".
01      VAL-OP-SW            PIC X.
        88                   VALID-OP   VALUE "V".
        88                   OP-FAILED  VALUE "F".
01      OP                   PIC X.
        88                   OP-OPEN    VALUE "O".
        88                   OP-CLOSE   VALUE "C".
        88                   OP-READ    VALUE "R".
REPORT SECTION.
RD      RPT     PAGE 26 LINES HEADING 1  FIRST DETAIL 5.
01              TYPE IS PAGE HEADING.
    02  LINE IS PLUS 1.
        03      COLUMN 1    PIC X(16)    VALUE "Employee File on".
        03      COLUMN 18   PIC Z9/99/99 SOURCE D-DATE.
    02  LINE IS PLUS 2.
        03      COLUMN 2    PIC X(5)     VALUE "emp  ".
        03      COLUMN 22   PIC X(4)     VALUE "name".
        03      COLUMN 42   PIC X(7)     VALUE "address".
        03      COLUMN 70   PIC ZZ9      SOURCE PAGE-COUNTER.
01      REPORT-LINE         TYPE IS DETAIL.
    02  LINE IS PLUS 1.
        03      COLUMN IS 1 PIC 9(7)     SOURCE EMP-ID.
        03      COLUMN IS 20 PIC X(15)   SOURCE IS EMP-NAME.
        03      COLUMN IS 42 PIC X(30)   SOURCE IS EMP-ADDRESS.
PROCEDURE DIVISION.
DECLARATIVES.
USE-SECT SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON EMP-FILE.
```

(continued on next page)

Example B–1 (Cont.): Using RMS Special Registers to Detect Errors

```
CHECK-RMS-SPECIAL-REGISTERS.
    SET OP-FAILED TO TRUE.
    EVALUATE RMS-STS OF EMP-FILE           TRUE
         WHEN (END-OF-FILE)                OP-READ
                  SET VALID-OP TO TRUE
                  SET E-O-F TO TRUE
         WHEN (BADNAME)                    OP-OPEN
         WHEN (FILE-NOT-FOUND)             OP-OPEN
         WHEN (DIR-NOT-FOUND)              OP-OPEN
         WHEN (INV-DEVICE)                 OP-OPEN
                  DISPLAY    "File cannot be found or file spec is invalid"
                  DISPLAY RMS-FILENAME OF EMP-FILE
                  DISPLAY "Enter corrected file (cntrl-z to STOP RUN): "
                              WITH NO ADVANCING
                  ACCEPT VAL-OF-ID AT END STOP RUN END-ACCEPT
         WHEN ANY                          OP-CLOSE
                  CONTINUE
         WHEN ANY                          RMS-STS OF EMP-FILE IS SUCCESS
                  SET VALID-OP TO TRUE
         WHEN OTHER
                  IF RMS-STV OF EMP-FILE NOT = ZERO
                  THEN
                    CALL "LIB$STOP" USING
                       BY VALUE RMS-STS OF EMP-FILE,
                       BY VALUE RMS-STV OF EMP-FILE
                  ELSE
                      CALL "LIB$STOP" USING
                       BY VALUE RMS-STS OF EMP-FILE
                  END-IF
    END-EVALUATE.
END DECLARATIVES.
MAIN-PROG SECTION.
000-DRIVER.
    PERFORM 100-INITIALIZE.
    PERFORM WITH TEST AFTER UNTIL E-O-F
        GENERATE REPORT-LINE
        READ EMP-FILE
    END-PERFORM
    PERFORM 200-CLEANUP.
    STOP RUN.
100-INITIALIZE.
    ACCEPT D-DATE FROM DATE.
    DISPLAY "Enter file spec of employee file: " WITH NO ADVANCING.
    ACCEPT VAL-OF-ID.
    PERFORM WITH TEST AFTER UNTIL VALID-OP
        SET VALID-OP TO TRUE
        SET OP-OPEN TO TRUE
        OPEN INPUT EMP-FILE
        IF OP-FAILED
        THEN
            SET OP-CLOSE TO TRUE
            CLOSE EMP-FILE
        END-IF
    END-PERFORM.
    OPEN OUTPUT REPORT-FILE.
    INITIATE RPT.
    SET NOT-E-O-F TO TRUE.
    SET OP-READ TO TRUE.
    READ EMP-FILE.
```

(continued on next page)

Example B–1 (Cont.): Using RMS Special Registers to Detect Errors

```
200-CLEANUP.
    TERMINATE RPT.
    SET OP-CLOSE TO TRUE.
    CLOSE EMP-FILE REPORT-FILE.
END PROGRAM RMS-SPEC-REGISTERS.
```

Example B–2: Using RMS-CURRENT Special Registers to Detect Errors

```
IDENTIFICATION DIVISION.
PROGRAM ID. RMS-CURRENT-SPEC-REGISTERS.
*
* This program demonstrates the use of RMS-CURRENT special registers
* to implement a single recovery for RMS file errors with multiple files
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILE-1
        ASSIGN TO "SYS$DISK:FOO_1.DAT".
SELECT FILE-2
        ASSIGN TO "SYS$DISK:FOO_2.DAT".
SELECT FILE-3
        ASSIGN TO "SYS$DISK:FOO_3.DAT".
DATA DIVISION.
FILE SECTION.
FD      FILE-1.
01      FILE-1-REC.
        02      F1-REC-FIELD    PIC 9(9).
FD      FILE-2.
01      FILE-2-REC.
        02      F2-REC-FIELD    PIC 9(9).
FD      FILE-3.
01      FILE-3-REC.
        02      F3-REC-FIELD    PIC 9(9).
PROCEDURE DIVISION.
DECLARATIVES.
USE-SECT SECTION.
        USE AFTER STANDARD EXCEPTION PROCEDURE ON INPUT.
CHECK-RMS-CURRENT-REGISTERS.
        DISPLAY "************** ERROR **************".
        DISPLAY "Error on file: " RMS-CURRENT-FILENAME.
        DISPLAY "Status Values:".
        DISPLAY "      RMS-STS = " RMS-CURRENT-STS WITH CONVERSION.
        DISPLAY "      RMS-STV = " RMS-CURRENT-STV WITH CONVERSION.
        DISPLAY "***********************************".
END DECLARATIVES.
MAIN-PROG SECTION.
MAIN-PARA.
        OPEN INPUT FILE-1.
        OPEN INPUT FILE-2.
        OPEN INPUT FILE-3.
        .
        .
        .
        CLOSE FILE-1.
        CLOSE FILE-2.
```

(continued on next page)

Example B–2 (Cont.): Using RMS-CURRENT Special Registers to Detect Errors

```
        CLOSE FILE-3.
        STOP RUN.
END-PROGRAM RMS-CURRENT-SPEC-REGISTERS.
```

You can check a file's I/O status by using File Status data items. Each time an I/O operation occurs, the system generates a 2-character file status key value. Checking a file's status during a USE procedure or in an INVALID KEY imperative condition can help you determine the nature of an I/O error. For example:

```
FD  INDEXED-MASTER
    ACCESS MODE IS DYNAMIC
    FILE STATUS IS MASTER-STATUS
    RECORD KEY IN IND-KEY.
  .
  .
  .
WORKING-STORAGE SECTION.
01  MASTER-STATUS      PIC XX  VALUE SPACES.
  .
  .
  .
PROCEDURE DIVISION.
  .
  .
  .
050-READ-MASTER.
    READ INDEXED-MASTER
      INVALID KEY PERFORM 100-CHECK-STATUS
      GO TO 200-INVALID-READ.
      .
      .
      .
100-CHECK-STATUS.
    IF MASTER-STATUS = "23"
       DISPLAY "RECORD NOT IN FILE".
    IF MASTER-STATUS = "90"
       DISPLAY "RECORD LOCKED BY ANOTHER USER".
      .
      .
      .
```

The *VAX COBOL Reference Manual* contains a list of File Status key values.

If your program contains a USE procedure for a file and an I/O operation for that file fails, the RTL performs the procedure, but the system does not display an RMS error message. A USE procedure can sometimes avoid program termination. For example, a File Status of 91 indicates that the file is locked by another program; rather than terminate your program, you could perform other procedures and then try re-opening the file.

If program continuation is not desirable, the USE procedure can perform housekeeping functions, such as saving data or displaying program-generated error messages.

If you specify an INVALID KEY phrase for a file and the I/O operation causes an INVALID KEY condition, the RTL performs the associated imperative statement. The RTL performs no other file processing for the current statement. The USE procedure (if any) is not performed. The INVALID KEY phrase processes I/O errors due to invalid key conditions only.

If you do not specify an INVALID KEY phrase but declare a USE procedure for the file, RTL performs the USE procedure and returns control to the program.

B.4 Compiler Messages

When the VAX COBOL compiler detects an error, a diagnostic message is generated by the compiler and is displayed online.

Some compile-time error messages need more explanation than can be provided online. This appendix contains those diagnostic messages that require additional detail.

If the compiler detects an error, it displays the erroneous source line, an error pointer, possibly an asterisk, and an error message. An asterisk immediately preceding the error message means further diagnostic information about a compile-time error is found in this appendix.

Compiler command line qualifiers can suppress informational and warning messages (see the /WARNINGS qualifier in Chapter 2 for diagnostic options.) Error messages are written to SYS$ERROR and, if a listing is specified, embedded in the listing file.

An error message has the following format:

```
%COBOL-a-ERROR bbb,(c) *ddd
```

Table B–1 explains the information contained in an error message.

Table B–1: Information Contained in an Error Message

Symbol	Meaning
%COBOL	identifies a VAX COBOL compiler-generated error.
a	identifies the severity code. The compiler classifies errors by these severity codes (from least to greatest severity): I Information—To get these messages, use the /WARNINGS, /WARNINGS=INFORMATION, or /STANDARD=PDP11 command qualifier. These messages convey observational or advisory information. However, they may indicate program errors that you might want to correct. Information—To get these messages, use the /WARNINGS, /WARNINGS=STANDARD, or /STANDARD=SYNTAX command qualifier. These messages are informational and indicate that you have used a Digital-defined COBOL language feature. This feature may not be transportable to other COBOL implementations.

(continued on next page)

Table B–1 (Cont.): Information Contained in an Error Message

Symbol	Meaning
	W Warning—Warnings indicate an error condition for which the compiler can take corrective action. Check this action to make sure it is what you wanted. Otherwise, the program might produce unexpected results.
	E Error—Error messages indicate error conditions that are not fatal, but are usually not executable by the compiler. The compiler still creates an object file, but the program might not properly execute. Therefore, you should correct the error and recompile the program.
	F Error—A fatal error indicates that the compiler cannot take any corrective action or create an object file. Therefore, you must correct the error and recompile the program.
bbb	indicates the diagnostic error message number listed in this appendix.
c	(Error Pointer Reference) references the error message to the closest approximation to where the error occurred.
*	indicates diagnostic information is found in the compile-time diagnostic appendix.
ddd	indicates the diagnostic error message. A brief description of the error identified by the error pointer.

The selected VAX COBOL compiler error numbers, severity codes, and messages are as follows:

Number	Code	Message
033	E	*Integer value is outside valid range—results are undefined. The value is either: (1) greater than that allowed by RMS, (2) not less than 2**31, or (3) equal to zero. The compiler truncates the value, and the results are undefined.
054	E	*VALUE and condition-names invalid with subordinate JUSTIFIED, SYNCHRONIZED, or non-DISPLAY usage. The compiler accepts the VALUE clause.
061	E	*Missing level 01 or 77 entry before this item. If the current item is a condition-name, the compiler ignores the definition; otherwise, the compiler treats the current item as an 01 level item.
091	E	*Incorrect data-name in REDEFINES clause. The compiler assumes the correct data-name, if possible. Otherwise, the clause is ignored.
209	F	*Invalid ADVANCING operand. The operand must be either: (1) PAGE, (2) a mnemonic-name, (3) an unsigned integer data item, or (4) an unsigned integer literal.
291	F	*READ statement required for OPEN I-O on file. The program opens the file in I-O mode, but does not contain a READ statement. A DELETE or REWRITE operation on a sequentially accessed file requires a previously executed READ or START statement.
388	F	*Node <CDD-node-name> is an invalid CDD object. The dictionary object is corrupt. A required VAX CDD entry was not found. You must reinsert the definition into the dictionary.

Number	Code	Message
391	E	*CDD OCCURS data-item initialization invalid in COBOL—value ignored. This VAX CDD entry is incompatible with VAX COBOL. An item in COBOL with an OCCURS clause cannot have a VALUE clause. The compiler ignores the initial value attribute in this dictionary entry.
392	F	*CDD error at node <CDD-node-name>. The dictionary object is corrupt. A required VAX CDD entry was not found. You must reinsert the record definition into the dictionary.
397	F	*CDD record containing symbolic literals invalid in COBOL. This VAX CDD entry is incompatible with VAX COBOL. The COBOL language does not define symbolic literals. You must do one of the following: • Redefine the record definition in the dictionary, omitting the symbolic literal • Do not use this record definition with VAX COBOL The compiler ignores the symbolic literal. You must redefine and reinsert the record definition into the dictionary.
401	E	*Invalid multidimensional CDD OCCURS. This VAX CDD entry is incompatible with VAX COBOL. VAX CDD supports the definition of multidimensional tables more generally than does VAX COBOL. You must do one of the following: • Redefine the record definition in the dictionary, omitting this multidimensional table definition • Do not use this record definition with VAX COBOL The compiler uses one dimension of information of a multidimensional table definition.
402	F	*Node <CDD-node-name> is a <CDD-node-type>, not a record definition. VAX COBOL requires a pathname to reference a VAX CDD record description. You must use a different pathname in your COPY FROM DICTIONARY statement or correct the VAX CDD entry to describe a record. The compiler ignores the COPY FROM DICTIONARY statement.
404	E	*Redefinition of FILLER invalid in COBOL. The compiler ignores the redefinition and treats the item as a separate data description entry.
411	E	*Length for database record-item <item-name> must be greater than zero. The minimum size for a COBOL data item is 1 byte. In a VAX DBMS Data Description entry, the minimum size for a subschema data item is 0 bytes. You must do one of the following: • Redefine the subschema data item to be at least 1 byte, and recompile the subschema with the VAX DDL Utility • Do not use this subschema with VAX COBOL The compiler treats the record-item as if it has a length of 1 byte. This may produce run-time errors.

Number	Code	Message
412	E	*Length for database record-item "<item-name>" must be multiple of 8 bits. This entry is incompatible with VAX COBOL. The smallest unit of data that can be defined in a VAX COBOL program is 1 byte (8 bits). The compiler rounds the length up to the next multiple of 8 bits.
413	E	*Offset for database record-item "<item-name>" must be multiple of 8 bits. This entry is incompatible with VAX COBOL. The smallest unit of data that can be defined in a VAX COBOL program is 1 byte (8 bits). The compiler rounds the offset up to the next multiple of 8 bits.
414	E	*Length for database record "<record-name>" must be multiple of 8 bits. This entry is incompatible with VAX COBOL. The smallest unit of data that can be defined in a VAX COBOL program is 1 byte (8 bits). The compiler rounds the length up to the next multiple of 8 bits.
418	F	*Error in accessing subschema—DB statement ignored. The VAX COBOL compiler was unsuccessful in its attempt to access the subschema and/or schema specified in your DB statement. You must make sure that the schema and subschema exist in the VAX CDD. Also verify that your logical names resolve to the schema and subschema that you intend to use. The compiler ignores the DB statement.
419	F	*Invalid stride for database record-item "<item-name>". This entry is incompatible with VAX COBOL. The stride attribute describes the separation (in bits) between one occurrence of a table item and the next occurrence. This should be the same as the allocated length of a single occurrence of the item. The compiler issues this diagnostic whenever the two lengths do not agree. You must do the following: (1) redefine your subschema to ensure that these two database record-item lengths are identical, and (2) recompile the subschema containing this record-item with the VAX DBMS DDL Utility. The compiler gives the record-item a stride equal to the length of one occurrence of the table item.
420	F	*Unable to complete subschema processing. The VAX COBOL compiler failed to exit the VAX CDD when it completed subschema processing. Recompile the COBOL program. The compiler terminates subschema processing.
439	F	*Error in accessing record-item "<item-name>" from subschema. The subschema containing the record item is most likely corrupt. Recompile the subschema with the VAX DBMS DDL Utility.
441	F	*Invalid multidimensional database record-item "<item-name>". This entry is incompatible with VAX COBOL. The VAX DBMS Data Definition Language defines multidimensional tables more generally than VAX COBOL. You must do one of the following: • Redefine the record definition in the dictionary, omitting such multidimensional table definitions • Do not use this record definition with VAX COBOL The compiler uses one dimension of information of the multidimensional table definition.
444	E	*Unsupported subschema datatype for "<record-item-name>". The compiler treats the item as if it were alphanumeric with a length equal to the original data type.

Number	Code	Message
458	F	*Invalid length for database record "<record-name>". The subschema containing the record definition with the invalid length is most likely corrupt. Recompile the subschema with the VAX DBMS DDL Utility. The compiler assigns the record a length of one byte. This may produce run-time errors.
466	F	*Offset required for database record-item "<item-name>". The required offset-into-the-record attribute is missing in the entry for this subschema record-item. Most likely the subschema is corrupt. Recompile the subschema using the VAX DBMS DDL Utility.
467	F	*Subschema must have at least one database record. This valid VAX DBMS DDL entry is invalid in COBOL. You must do one of the following: • Redefine the subschema including a record-type definition and recompile the subschema with the VAX DBMS DDL Utility • Do not use such subschema definitions with VAX COBOL The compiler terminates subschema processing.
468	F	*Upper-bound required for database record-item "<item-name>". The compiler encountered a table definition with no upper-bound. Most likely the subschema record description is corrupt. Recompile the subschema with the VAX DBMS DDL Utility.
473	E	*OCCURS DEPENDING ON data-name must be defined in the same DATA DIVISION as the OCCURS DEPENDING ON. The compiler ignores the error.
479	F	*Error in accessing record "<record-name>" from subschema. The subschema containing the record is most likely corrupt. Recompile the subschema using the VAX DBMS DDL Utility.
480	F	*REFERENCE data name defined in an invalid section. Data name must be defined in the Subschema, File, or Working-Storage Sections. The compiler ignores the entire VALUE clause.
484	F	*Error in accessing realm "<realm-name>" from subschema. The subschema containing the realm is most likely corrupt. Recompile the subschema with the VAX DBMS DDL Utility.
493	F	*Invalid GLOBAL clause. This data-item either: (1) does not have an explicit name, (2) is not an 01 or 77 item, or (3) is not defined in the File or Working-Storage Sections.
494	F	*Error in accessing set "<set-name>" from subschema. The subschema containing the set is most likely corrupt. Recompile the subschema with the VAX DBMS DDL Utility.
495	F	*Numeric database record-item "<item-name>" must represent at least one digit. This subschema entry is incompatible with VAX COBOL. The VAX DBMS Data Definition Language allows the definition of numeric record-items to consist of zero digits, VAX COBOL does not. Correct the definition of the numeric record-item and recompile the subschema with the VAX DBMS DDL Utility.
532	F	*OFFSET invalid in a format 3 record selection expression. OFFSET can be used in the database key identifier form of the record selection expression, but not the record search access form.

Number	Code	Message
533	F	*Operand subordinate to another database group operand. The list of record-items for this statement cannot contain any item that is subordinate to another item in the list.
562	F	*Invalid number of selection objects. In the WHEN clause of an EVALUATE statement, the number of selection objects must equal the number of selection subjects.
584	I	*Allocation of this table is incompatible with subset. For information on how to correct this error, see Appendix C.

Appendix C

Using the COBOL–81 SUBSET Flagger

This appendix describes the VAX COBOL compile-time qualifier /STANDARD=PDP11.

C.1 Using VAX COBOL to Produce Compatible COBOL–81 Source Programs

COBOL–81 is the COBOL compiler that runs on the Professional and PDP–11 systems.

COBOL–81 is a subset of VAX COBOL. This superset-subset relationship allows you to create programs that are portable between VAX and PDP–11 systems. Thus, a COBOL source program that contains only language elements supported by COBOL–81 can run on both systems. However, some architectural differences between the PDP–11 and VAX computers cause incompatibilities between COBOL–81 and VAX COBOL that prevent program portability.

The COBOL–81 flagger provided by VAX COBOL enables you to detect and eliminate incompatible language elements in VAX COBOL programs. The flagger identifies all language elements in VAX COBOL source programs that are not part of the COBOL–81 subset. It also identifies source language elements related to the system incompatibilities previously mentioned.

You invoke the COBOL–81 flagger by specifying the /STANDARD=PDP11 command line qualifier. To receive the diagnostics, you must also specify the /WARNINGS=ALL qualifier.

C.2 Using the /STANDARD=PDP11 Qualifier

When you compile a VAX COBOL program with the /STANDARD=PDP11 qualifier, informational diagnostics indicate which language elements prevent the COBOL program from being VAX/PDP–11 compatible.

These informational diagnostics do not prevent the compiler from generating object code. When you invoke the flagger, the compiler issues appropriate diagnostics to the terminal and, if /LIST was specified, to the listing file.

Because the COBOL–81 and VAX COBOL compilers handle certain errors differently, a COBOL program is considered compatible only when it compiles free from all levels of diagnostics. For example, an error diagnosed as W- or E-level by VAX COBOL (that does not prevent object code generation) may be diagnosed as F-level by COBOL–81. In these instances, the flagger does not provide additional diagnostics. Digital extensions that are supported by both VAX COBOL and COBOL–81 are among the few exceptions to this rule.

NOTE

Use the /WARNINGS=ALL command line qualifier with the /STANDARD=PDP11 qualifier to ensure that the compiler does not suppress the generated diagnostics. The /WARNINGS=ALL qualifier causes the compiler to list all warning-level and informational-level diagnostics.

C.3 VAX COBOL Flagging Procedures

This section describes how the COBOL–81 flagger determines and flags COBOL–81 incompatible elements in your COBOL program. Keep this approach in mind when you review your COBOL program compilation listing and revise your COBOL program to make it compatible.

A COBOL source program has two levels where differences may occur:

- The sectional level
- The context-specific level

When the flagger views a program at the sectional level, it sees the source as composed of many COBOL sections (CONFIGURATION SECTION, INPUT-OUTPUT SECTION, etc.). If the COBOL program contains a section that is not supported by COBOL–81, the COBOL–81 flagger issues a single diagnostic on that section. For example, COBOL–81 does not support any portion of the Report Section (for Report Writer). Therefore, the flagger issues a single diagnostic at the beginning of the section, rather than flagging each item defined in the Report Writer Section.

The context-specific level encompasses the lexical, syntactic, and semantic views of the COBOL source program; most diagnostics are context-specific. A context-specific message either describes the error or specifies a corrective action.

C.4 Source Level Differences and Incompatibilities

This section lists source level differences and incompatibilities between VAX COBOL and COBOL–81. The COBOL–81 flagger diagnoses most of these differences.

A COBOL–81 flagger diagnostic indicates a language element that you must remove or alter to make the program compatible. Where the necessary alteration is not obvious, the flagger provides an explanation of the recommended alteration.

The *VAX COBOL Reference Manual* and the *COBOL–81 Reference Manual* contain complete information on VAX COBOL and COBOL–81 syntax, respectively.

C.4.1 General Language Concepts

The following list describes the lexical differences between VAX COBOL and COBOL–81:

- The dollar sign ($) and underscore (_) are invalid characters for COBOL–81 words.
- COBOL–81 words are limited to 30 characters.
- Null nonnumeric literals ("") are not supported by COBOL–81.

- Apostrophes (′) are not valid nonnumeric literal delimiters in COBOL–81.
- Terminal format source lines are limited to 200 characters in COBOL–81.

The following non-division-specific language elements are not supported by COBOL–81:

- The following COPY statement options:
  - text-name OF/IN library-name
  - REPLACING pseudo-text (= =)
  - REPLACING ′identifier′
  - FROM DICTIONARY
- END PROGRAM header
- Contained programs
- More than one source program in a file
- Floating point literals, hexadecimal literals
- REPLACE statements

C.4.2 Unsupported Language Elements by Division

The following sections explain those language elements not supported by COBOL–81. The elements are grouped by division.

IDENTIFICATION DIVISION

The following VAX COBOL language elements in the IDENTIFICATION DIVISION are not supported by COBOL–81:

- COMMON clause
- IDENT clause
- INITIAL clause

ENVIRONMENT DIVISION

The following VAX COBOL language elements in the ENVIRONMENT DIVISION are not supported by COBOL–81:

- SEGMENT LIMIT clause

  In VAX COBOL the SEGMENT LIMIT clause is for documentation only. In COBOL–81, it is enforced.
- Literal option of the ALPHABET clause
- ASCII and EBCDIC as system names
- Symbolic characters
- FILE-CONTROL and file description entry clauses

  In VAX COBOL, some clauses that describe files in COBOL can appear in either the FILE-CONTROL paragraph or the file description entry (FD) of a file, but not in both. COBOL–81 does not allow this. The following can appear only in the FILE-CONTROL paragraph in COBOL–81:
  - ACCESS MODE clause
  - FILE STATUS clause

  - RECORD KEY clause (indexed files)
  - ALTERNATE RECORD KEY clause (indexed files)

  Similarly, the following can appear only in the file description entry (FD) in COBOL–81:

  - BLOCK CONTAINS clause
  - CODE-SET clause

- OPTIONAL phrase for relative and indexed files.
- PADDING CHARACTER phrase
- RECORD DELIMITER IS STANDARD-1 phrase
- RERUN clause in I-O-CONTROL

  In COBOL–81, the RERUN clause in I-O-CONTROL cannot specify a conditional name.

- File-spec in an ASSIGN clause

  In COBOL–81, the file-spec in an ASSIGN clause must be a nonnumeric literal.

- RECORD KEY or ALTERNATE RECORD KEY for an indexed file

  In COBOL–81, the RECORD KEY or ALTERNATE RECORD KEY for indexed files must be alphanumeric. Also duplicate and primary descending keys are not supported.

- CONTIGUOUS-BEST-TRY, C01, and LOCK-HOLDING as system names.
- SAME AREA clause

  In VAX COBOL the SAME AREA clause is used for documentation only. In COBOL–81, it is enforced.

- APPLY clause syntax differences

  In VAX COBOL you must choose one or more options for each APPLY clause. In COBOL–81 you can choose only one option per APPLY clause. The following APPLY clause entry is valid in VAX COBOL, but not in COBOL–81 for the reason just described:

  ```
  APPLY FILL-SIZE MASS-INSERT ON FILE-A
  ```

  To make this clause compatible, expand it into two APPLY clauses:

  ```
  APPLY FILL-SIZE ON FILE-A

  APPLY MASS-INSERT ON FILE-A
  ```

- APPLY PREALLOCATION clause

  The maximum preallocation amount in the APPLY PREALLOCATION clause differs between VAX COBOL and COBOL–81.

- MULTIPLE FILE TAPE clause
- WINDOW phrase

  The values of window-ptrs in the WINDOW phrase differ between VAX COBOL and COBOL–81.

DATA DIVISION

The following VAX COBOL language elements in the DATA DIVISION are not supported by COBOL–81:

- SUB-SCHEMA SECTION
- REPORT SECTION
- EXTERNAL clause in file description entries
- GLOBAL clause in file description entries
- EXTERNAL clause in record description entries
- GLOBAL clause in record description entries
- OCCURS clause as follows:
  - Lower-bound of zero
  - More than three levels of nesting
  - VAX COBOL follows the 1985 ANSI COBOL standard rules for evaluating the size of an OCCURS DEPENDING ON item.
- PICTURE character string

  VAX COBOL follows the 1985 ANSI COBOL standard rules for when PICTURE P positions are zero.

  In COBOL–81, a PICTURE character string cannot exceed 30 characters.
- COMP-1 usage
- COMP-2 usage
- POINTER usage
- USAGE IS INDEX items (size and alignment differences)
- Alignment of COMP data items

  This incompatibility requires careful consideration due to the complexity of data allocation. See Section C.5; it explains the differences between the compilers allocation of COMP data items and presents methods available to correct the incompatibility.
- VALUE clause, except in the WORKING-STORAGE SECTION
- EXTERNAL option on VALUE clause
- REFERENCE option on VALUE clause

PROCEDURE DIVISION

The following VAX COBOL language elements in the PROCEDURE DIVISION are not supported by COBOL–81.

- Nesting parentheses

  Parentheses can be nested to a maximum of 24 levels in COBOL–81. VAX COBOL has no limit for parentheses nesting; however, the size of the entire expression can be too large for the internal parse stack.
- Special registers

  The following COBOL special registers occupy 4 bytes in VAX COBOL and 2 bytes in COBOL–81:
  - RMS-STS

  - RMS-STV
  - LINAGE-COUNTER

  The values returned in the special registers RMS-STS and RMS-STV differ between the VAX and PDP–11 systems.

  COBOL–81 does not recognize the following VAX COBOL special registers:

  - DB-CONDITION
  - DB-CURRENT-RECORD-NAME
  - DB-CURRENT-RECORD-ID
  - RMS-FILENAME
  - RMS-CURRENT-STS
  - RMS-CURRENT-STV
  - RMS-CURRENT-FILENAME

- Subscripting

  COBOL–81 supports a maximum of 3 subscripts.

  In COBOL–81, a subscript cannot be an arithmetic expression or a data name.

  VAX COBOL supports the 1985 ANSI COBOL standard rules for subscript evaluations. Because of the differences in subscript evaluation, differences may result when you use the DIVIDE, STRING, UNSTRING, and INSPECT statements.

- Reference modification
- File Status values

  COBOL–81 does not support File Status value 25. Although COBOL–81 supports File Status value 96, that error does not occur in VAX COBOL programs.

- Record selection expressions for COBOL DML
- Database conditions
- Segment-numbers

  COBOL–81 supports segment-numbers between 0 and 49.

- ACCEPT statement
  - FROM DAY-OF-WEEK phrase
  - AT END phrase
- ALTER statement
- COBOL–81 does not support the following CALL statement options:
  - 'identifier'
  - BY VALUE
  - BY CONTENT
  - 'literal' (illegal characters and truncation)
  - GIVING
  - ON EXCEPTION

  - ON OVERFLOW

  COBOL–81 supports only alphanumeric items as CALL BY DESCRIPTOR arguments.

- CANCEL statement
- COMPUTE statement

  Exponents specified in COMPUTE must be integers in COBOL–81.

- Data manipulation language (DML) statements
- DIVIDE statement

  VAX COBOL supports the 1985 ANSI COBOL standard rules for subscript evaluation in DIVIDE statements.

- EVALUATE statement
- EXIT PROGRAM statement

  In a main program VAX COBOL ignores the EXIT PROGRAM statement.

- GENERATE statement
- Optional procedure name for GO TO
- INITIALIZE statement
- INITIATE statement
- INSPECT statement—CONVERTING phrase

  The BEFORE/AFTER clause syntax of the INSPECT statement in COBOL–81 is a subset of VAX COBOL syntax.

  VAX COBOL supports the 1985 ANSI COBOL standard rules for subscript evaluation in the INSPECT statement.

  VAX COBOL supports multiple arguments for the ALL/LEADING phrase of the INSPECT statement.

- MERGE statement restrictions

  MERGE key size cannot be greater than 255 characters. Total number of characters in MERGE keys cannot exceed 512. MERGE cannot specify more than 16 merge keys. Neither infile nor outfile can describe an indexed file in random access mode.

- MOVE—De-editing of numeric edited items
- OPEN statement

  The following restrictions exist for the OPEN statement in COBOL–81:

  - OPEN EXTEND is allowed only on files with SEQUENTIAL organization.
  - Files specified in a SAME AREA clause cannot be open at the same time.
  - The following clauses of the OPEN statement:
    - ALLOWING WRITERS
    - ALLOWING UPDATERS
    - Optional files may not be opened for OUTPUT, I-O, or EXTEND
- PERFORM
  - Optional procedure name
  - TEST BEFORE/AFTER clause

  - VAX COBOL supports the 1985 ANSI COBOL standard rules for evaluation of identifiers in the AFTER clause.
- Procedure Division Header GIVING phrase
- RECORD statement
- The following clauses of the READ statement:
  - ALLOWING UPDATERS
  - ALLOWING READERS
  - ALLOWING NO OTHERS
  - REGARDLESS OF LOCK
- ALLOWING NO OTHERS clause on the REWRITE statement
- The following clauses of the SET statement:
  - condition-name TO TRUE
  - pointer-id TO REFERENCE
  - status-code-id TO SUCCESS/FAILURE
- The following clauses of the START statement:
  - ALLOWING UPDATERS
  - ALLOWING READERS
  - ALLOWING NO OTHERS
  - REGARDLESS OF LOCK
- STRING statement

  VAX COBOL supports the 1985 ANSI COBOL standard rules for subscript evaluation.
- SUPPRESS statement
- TERMINATE statement
- UNSTRING statement

  VAX COBOL supports the 1985 ANSI COBOL standard rules for subscript evaluation.
- The following clauses of the USE statement:
  - GLOBAL
  - BEFORE REPORTING
  - FOR DB-EXCEPTION
- Mnemonic name following BEFORE/AFTER ADVANCING clause of the WRITE statement
- The VFU-CHANNEL clause

C.5 Alignment of COMP Data Items

COBOL–81 aligns COMPUTATIONAL (COMP) data items on word boundaries. VAX COBOL does not have this alignment restriction, and so aligns COMP data items on the next available byte boundary.

This alignment difference becomes an incompatibility when COMP items are used in files as in group names; however, all COMP items must be allocated compatibly before the program is transferable. The COBOL–81 flagger indicates those COMP items that are incompatibly aligned, that is, any COMP item that does not begin on a word boundary. There are two methods of correcting this incompatibility:

- Specify the SYNCHRONIZED (SYNC) clause for all COMP items in your program.
- Insert FILLER data items before incompatible COMP items to force compatible alignment.

The SYNCHRONIZED clause causes VAX COBOL and COBOL–81 to align COMP items on the same boundaries. The COMP item's size determines the required alignment boundary. The compiler can insert fill bytes before the COMP item to obtain the correct alignment. This method is the preferred corrective method, as it is the simplest way to ensure compatibility.

The second method involves the use of FILLER data items to force incompatible COMP items to word boundaries. Correcting an incompatible record using this method may result in a record that occupies less space than one corrected using the SYNCHRONIZED clause. In addition to identifying incompatible COMP items, the COBOL–81 flagger indicates where FILLER data items should be inserted to force compatible alignment.

The following examples illustrate incompatible items and the use of both methods to correct them.

Example 1—Incompatible Data Item

Incompatible record

```
01   ITEM-A.
     03   ITEM-B  PIC X.
     03   ITEM-C  PIC 9(9) COMP.
```

COBOL–81 flagger diagnosis:

```
    7         01  ITEM-A.
    8             03      ITEM-B  PIC X.
    9             03      ITEM-C  PIC 9(9) COMP.
                          1
%COBOL-I-ERROR  586, (1) Allocation of this item is incompatible
with subset
%COBOL-I-ERROR  584, (1) Insert 1 fill byte before this item
(same level number) for subset compatibility
```

Method 1:

Place the SYNCHRONIZE, or SYNC, clause on the COMP item:

```
FD   FILE-1.
01   ITEM-A.
     03   ITEM-B  PIC X.
     03   ITEM-C  PIC 9(9) COMP SYNC.
```

Method 2:

Refer to the preceding COBOL–81 flagger diagnosis. To correct the record, insert a FILLER data item with PIC X as indicated by the informationals:

```
FD   FILE-1.
01   ITEM-A.
     03   ITEM-B  PIC X.
     03   FILLER  PIC X.
     03   ITEM-C  PIC 9(9) COMP.
```

Example 2—Tables

Incompatible record:

```
FD   FILE-1.
01   ITEM-A.
     03      ITEM-B OCCURS 3 TIMES.
          05      ITEM-C PIC 9(9) COMP.
          05      ITEM-D PIC X.
```

COBOL–81 flagger diagnosis:

```
   21         01  ITEM-A.
   22             03      ITEM-B OCCURS 3 TIMES.
                          1
%COBOL-I-ERROR  585, (1) Allocation of this table is incompatible
with subset
   23                     05      ITEM-C PIC 9(9) COMP.
   24                     05      ITEM-D PIC X.
```

Method 1:

Place the SYNCHRONIZE, or SYNC, clause on the COMP item:

```
FD   FILE-1.
01   ITEM-A.
     03      ITEM-B OCCURS 3 TIMES.
          05      ITEM-C PIC 9(9) COMP SYNC.
          05      ITEM-D PIC X.
```

Method 2:

This table is more difficult to correct using the second method (the manual method).

Because the solution is not easily indicated by informational diagnostics, the flagger does not point out where to insert FILLER data items. The length of the occurring group is odd, and the COMP item is on an even-byte offset relative to the beginning of the record. This causes every other occurrence of ITEM-C (starting with the second) to begin on an odd-byte boundary. The correction should cause each occurrence of ITEM-B to begin on an even-byte boundary, without increasing the size of ITEM-B.

```
FD   FILE-1.
01   ITEM-A.
     03      OCCURS 3 TIMES.
          05   ITEM-B.
               10      ITEM-C PIC 9(9) COMP.
               10      ITEM-D PIC X.
          05   FILLER    PIC X.
```

Example 3—Complex Record

Incompatible record:

```
01   ITEM-A.
     03   ITEM-B    PIC X.
     03   ITEM-C.
          05   ITEM-D    PIC X.
          05   ITEM-E    PIC 9(4) COMP.
```

One of the rules that affects data allocation in both compilers is known as Boundary Equivalence. Refer to the *VAX COBOL Reference Manual*, for a description of Boundary Equivalence. Since the COBOL–81 compiler requires ITEM-E to be word aligned, ITEM-C must be word aligned according to the Boundary Equivalence rule. Thus, since ITEM-C begins on an odd-byte boundary, it is incompatible. Note that the flagger diagnoses this incompatibility as well as that of ITEM-E.

COBOL–81 flagger diagnosis:

```
   38         01  ITEM-A.
   39                     03      ITEM-B  PIC X.
   40                     03      ITEM-C.
                                  1
%COBOL-I-ERROR  586, (1) Allocation of this item is
incompatible with subset
%COBOL-I-ERROR  583, (1) Insert 1 fill byte before this
group item for subset compatibility
   41                     05      ITEM-D  PIC X.
   42                     05      ITEM-E  PIC 9(4) COMP.
                                  1
%COBOL-I-ERROR  586, (1) Allocation of this item is
incompatible with subset
%COBOL-I-ERROR  584, (1) Insert 1 fill byte before this item
(same level number) for subset compatibility
```

Method 1:

Place the SYNCHRONIZE, or SYNC, clause on the COMP item:

```
01   ITEM-A.
     03   ITEM-B    PIC X.
     03   ITEM-C.
          05   ITEM-D    PIC X.
          05   ITEM-E    PIC 9(4) COMP SYNC.
```

Method 2:

Refer to the preceding COBOL–81 flagger diagnosis. To correct the record, insert FILLER data item with PIC X as indicated by the informationals:

```
01   ITEM-A.
     03   ITEM-B    PIC X.
     03   FILLER    PIC X.
     03   ITEM-C.
          05   ITEM-D    PIC X.
          05   FILLER    PIC X.
          05   ITEM-E    PIC 9(4) COMP.
```

Appendix D

Additional Information on COBOL Command Qualifiers

This appendix provides additional information on the following COBOL command line qualifiers:

- /FLAGGER
- /INSTRUCTION_SET
- /STANDARD

D.1 Using the /FLAGGER Qualifier

This section explains how to use the /FLAGGER qualifier and provides the following information:

- Options used with the /FLAGGER qualifier
- FIPS validation levels
- A table indicating FIPS levels

In accordance with the *Federal Information Processing Standard Publication 21-2* (FIPS-PUB 21–2) issued by the U.S. National Bureau of Standards, VAX COBOL allows you to specify a FIPS level of COBOL syntax beyond which informational diagnostics will be generated. To do this, include the /FLAGGER qualifier when you compile your VAX COBOL program. To receive the informational diagnostics, you must also use the /WARNINGS=INFORMATIONAL or /WARNINGS=ALL qualifier. The /FLAGGER qualifier is particularly useful when a target system's compiler has a low level of FIPS syntax support.

When you use the /FLAGGER qualifier with its options, you receive diagnostic messages for syntax in the source program as follows:

- Not within the FIPS validation level you selected when the source program is compiled
- Within the optional module you selected when the source program is compiled
- For obsolete language elements as defined by the ANSI 1985 standard for the COBOL language
- For Digital extensions to the COBOL language

D.1.1 /FLAGGER Options

/FLAGGER=HIGH_FIPS produces informational diagnostics for language constructs in the source program that are above the FIPS high validation level. These are constructs supported by VAX COBOL that are Digital extensions to the COBOL language.

/FLAGGER=INTERMEDIATE_FIPS produces informational diagnostics for language constructs in the source program that are above the FIPS intermediate validation level. These are constructs supported by VAX COBOL that are within the FIPS high validation level or Digital extensions.

/FLAGGER=MINIMUM_FIPS produces informational diagnostics for language constructs in the source program that are above the FIPS minimum validation level. These are constructs supported by VAX COBOL that are within the FIPS high and intermediate validation levels or Digital extensions.

/FLAGGER=OBSOLETE produces informational diagnostics for language constructs that are identified as obsolete by the ANSI 1985 COBOL standard for the COBOL language. If a language construct is in the selected FIPS level or optional module and is also on the obsolete list, only the obsolete diagnostic will be generated if /FLAGGER=OBSOLETE is specified.

/FLAGGER=OPTIONAL_FIPS produces informational diagnostics for language constructs supported by VAX COBOL within FIPS optional modules. VAX COBOL provides support for the optional modules Report Writer and Segmentation.

/FLAGGER=REPORT_WRITER produces informational diagnostics for language constructs supported by VAX COBOL within the FIPS optional module Report Writer.

/FLAGGER=SEGMENTATION provides informational diagnostics for language constructs supported by VAX COBOL within the FIPS optional module Segmentation.

/FLAGGER=SEGMENTATION_1 provides informational diagnostics for language constructs supported by VAX COBOL within level 1 of the FIPS optional module Segmentation.

Any combination of qualifier options is permitted. If more than one validation level (high, intermediate, or minimum) is specified, the lowest level is used. If no FIPS level is specified, but another /FLAGGER option is used, /FLAGGER=HIGH_FIPS is assumed.

The default is /NOFLAGGER. This qualifier cannot be used if /STANDARD=V3 is also used.

D.1.2 FIPS Levels

Table D–1 shows the required functional processing modules and the optional modules supported by VAX COBOL. The table also shows the COBOL subsets that correspond to the FIPS levels of Minimum, Intermediate, and High. The levels numbers (0, 1, and 2) correspond to the levels indicated in the 1985 ANSI COBOL standard.

Table D–1: Relationship Among VAX COBOL Modules, Subsets, and Levels

	COBOL Subsets		
	Minimum	**Intermediate**	**High**
Required Modules			
Nucleus	1	1	2
Sequential I-O	1	1	2
Relative I-O	0	1	2
Indexed I-O	0	1	2
Interprogram Communication	1	1	2
Sort-merge	0	1	1
Source Text Manipulation	0	1	2
Optional Modules			
Report Writer	–, or 1	–, or 1	–, or 1
Segmentation	–, 1, or 2	–, 1, or 2	–, 1, or 2

Table Legend

0—Null level (the module is not included in the subset)
1—First nonnull level
2—Second nonnull level
Dash—Optional

For more information, refer to the *Federal Information Processing Standard Publication 21-2* (FIPS-PUB 21–2) and the 1985 ANSI COBOL standard.

D.2 /INSTRUCTION_SET Qualifier

The /INSTRUCTION_SET qualifier allows you to optimize your VAX COBOL programs for certain VAX processors.

The /INSTRUCTION_SET qualifier has the following options:

- DECIMAL_STRING
- NODECIMAL_STRING
- GENERIC

The default is /INSTRUCTION_SET=DECIMAL_STRING.

NOTE

Regardless of the /INSTRUCTION_SET option you choose, your VAX COBOL program will run on any VAX processor. The /INSTRUCTION_SET qualifier merely allows you to optimize your code for certain processors.

D.2.1 Overview of VAX Architectural Subsetting

The VAX family of processors offers a wide range of processors within a single architectural framework. All VAX processors have the ability to run all of the system software and layered products in the VAX family. In this way, different VAX processors are able to meet a variety of cost and performance needs.

In order to achieve this wide range of processors with a single architecture, the VAX instruction set has been divided into subsets. Processors that do not implement the full instruction set in the hardware can be referred to as *subset processors*.

Subset processors implement various subset classes of instructions through software emulation. This process allows the processors to offer full VAX functionality at some reduction in execution speed.

For more information on VAX architectural subsetting refer to the *VAX Architecture Reference Manual.*

D.2.2 How Subsetting May Affect VAX COBOL Programs

Some VAX COBOL programs may experience performance degradation on processors that emulate the decimal string instructions. Performance degradation occurs because emulating large numbers of instructions takes a processor longer than if the instructions are implemented in the hardware.

The most noticeable performance degradation occurs in VAX COBOL programs that are CPU intensive (that is, programs that require a lot of CPU time as opposed to I/O time). However, since most COBOL programs are I/O intensive, they are likely to experience only slight performance degradation when run on a subset processor.

D.2.3 Determining the Instruction Set

The instructions that affect CPU intensive COBOL programs are the decimal string instructions. You can use the following DCL commands to determine whether or not your VAX processor implements the decimal string instructions in the hardware:

```
$ DECIMAL_EMULATED=F$GETSYI("DECIMAL_EMULATED")
$ SHOW SYMBOL DECIMAL_EMULATED
```

The second command returns a true or false response. If the response is true, your system emulates the decimal string instructions. If the response is false, your system implements the decimal string instructions in the hardware.

D.2.4 Selecting an Option

The /INSTRUCTION_SET qualifier enables you to optimize your VAX COBOL program for the environment your VAX COBOL application will run in. Depending on the environment, you might want to select a specific option.

Using /INSTRUCTION_SET=DECIMAL_STRING

/INSTRUCTION_SET=DECIMAL_STRING optimizes the code for VAX processors that include the decimal string instructions in the hardware. This is the default.

You should use this option if your program will be running on processors that implement the decimal string instructions in the hardware.

Using /INSTRUCTION_SET=NODECIMAL_STRING

/INSTRUCTION_SET=NODECIMAL_STRING instructs the compiler to optimize the code for VAX processors that do not include the decimal string instructions in the hardware. You should consider using this option when both of the following conditions are true:

- The application will only be run on VAX processors that emulate the decimal string instructions
- CPU performance is important

Using this option will improve the performance of most VAX COBOL applications that run on a subset processor. However, you will see the most improvement in applications that are CPU intensive.

Using /INSTRUCTION_SET=GENERIC

/INSTRUCTION_SET=GENERIC offers a compromise between the other two settings. Using this option avoids some of the performance degradation programs may experience when run on a subset processor. However, programs compiled using this option, perform at nearly optimal levels on processors that include the decimal string instructions in the hardware.

You should consider using this option when your VAX COBOL application will run in an environment that includes VAX processors that emulate the decimal string instructions, as well as processors that implement these instructions in the hardware.

Other Considerations

The /INSTRUCTION_SET qualifier instructs the compiler to generate different code sequences for various operations. Although the behavior of the code generated with the various /INSTRUCTION_SET options is the same when the input data is valid, there may be some differences when programs encounter invalid data. Invalid data may be caused by program logic errors or corrupt data files.

When working with DISPLAY or PACKED DECIMAL data items with /INSTRUCTION_SET=DECIMAL_STRING, the instructions generated for operations often check the validity of the input data items being processed. This is a side effect of the instructions being used. If bad data is encountered at run time, the INVALID DECIMAL DATA message is generated and the program terminates execution.

When you use /INSTRUCTION_SET=NODECIMAL_STRING or /INSTRUCTION_SET=GENERIC, some of the operations do not validate the correctness of the data being processed. If your program encounters invalid data items, the program continues to execute; however, the results are unpredictable.

If your program runs in an environment where invalid data is common, you may want to use /INSTRUCTION_SET=DECIMAL_STRING.

If you are developing a large application with several modules, you do not have to compile all the modules with the same qualifier value. For more information on compiling VAX COBOL programs, see Chapter 2.

D.3 Differences Using /STANDARD=85 and /STANDARD=V3

This section explains the differences between the code generated when you compile programs using /STANDARD=85 and /STANDARD=V3.

D.3.1 Overview

Version 4.0 and higher versions of VAX COBOL are based on the ANSI 1985 COBOL standard. Versions prior to Version 4.0 of VAX COBOL were based on the ANSI 1974 COBOL standard. While most of the enhancements made to this version of VAX COBOL are compatible with versions of the compiler prior to Version 4.0, some differences exist. Although these differences do not affect most existing programs, there are some instances where results might vary.

To minimize conflicts with existing programs, VAX COBOL allows you to compile programs according to the rules for either the current version or Version 3.4. To do this you use the /STANDARD qualifier.

/STANDARD=85 instructs the compiler to compile and generate code according to the ANSI 1985 COBOL standard. /STANDARD=85 is the default.

/STANDARD=V3 instructs the compiler to compile and generate code in the manner of VAX COBOL Version 3.4 in specific instances.

If you use the /WARNINGS=ALL qualifier with the /STANDARD qualifier, the compiler generates informational diagnostics for language constructs that depend on the /STANDARD qualifier.

The following statements are affected by the /STANDARD qualifier:

- Subscript evaluation and reference modification in the following instances:
  - Remainder of a DIVIDE statement
  - STRING and UNSTRING statements
  - Identifiers in an INSPECT statement
- The order of evaluation of identifiers for some PERFORM ... VARYING ... AFTER statements.
- PIC P items. These are zero in new cases.
- The size of a variable-length table is different in certain cases.
- EXIT PROGRAM statement in a main program.
- New and revised I/O file status codes.
- Opening nonoptional files in I-O and EXTEND mode.

D.3.2 DIVIDE Statement

The /STANDARD=85 qualifier instructs the compiler to evaluate subscripts associated with the REMAINDER phrase of DIVIDE statements after the result of the divide operation is stored in the identifier associated with the GIVING phrase.

The /STANDARD=V3 qualifier instructs the compiler to evaluate subscripts for the REMAINDER phrase at the beginning of the DIVIDE statement.

The following example highlights the difference in how subscripts for the DIVIDE statement are evaluated:

```
DIVIDE DIVIDEND BY DIVISOR GIVING QUOTIENT
       REMAINDER REM (QUOTIENT).
```

If you use /STANDARD=85, the subscript (QUOTIENT) for REM is evaluated after the value of QUOTIENT is updated.

If you use /STANDARD=V3, the subscript (QUOTIENT) for REM is evaluated before the DIVIDE statement is executed.

D.3.3 STRING Statement

The /STANDARD=85 qualifier instructs the compiler to evaluate subscripts and reference modifications for source strings and delimiters in a STRING statement once, as the first operation of the execution of the statement.

The /STANDARD=V3 qualifier instructs the compiler to evaluate the subscripts and reference modifications for source strings and delimiters prior to examining each source string. Also, the pointer is updated after each examination of a source string.

The following example highlights the difference in how subscripts for the source string in a STRING statement are evaluated:

```
STRING SRC-STRING-A (PTR) SRC-STRING-B (PTR)
   DELIMITED BY DELIM (PTR)
   INTO DEST-STRING
   WITH POINTER PTR
```

If you use /STANDARD=85, the value of PTR is the same for both source strings and the delimiter.

If you use /STANDARD=V3, the subscript, PTR, for SRC-STRING-A and DELIM is evaluated before characters are transferred to DEST-STRING. After the characters are moved, the number of characters moved is added to PTR. The new value of PTR is then used to evaluate SRC-STRING-B (PTR) and DELIM (PTR).

D.3.4 UNSTRING Statement

The /STANDARD=85 qualifier instructs the compiler to evaluate all subscripts and reference modifications in the UNSTRING statement once, as the first operation when the statement is executed.

The /STANDARD=V3 qualifier instructs the compiler to evaluate subscripts and reference modifications for the source string, pointer, and tallying items once, as the first operation when the statement is executed. Subscripts for delimiters are evaluated before each examination of the source string. Subscript evaluation for the destination string, delimiter destination, and the counter occur just before the data is transferred to these items. The pointer and tallying items are updated after each examine and move cycle.

The following example highlights the difference in how subscripts for the UNSTRING statement are evaluated:

```
UNSTRING SRC-STRING DELIMITED BY "A"
   INTO DEST-STRING (CNT)
        DEST-STRING (CNT)
   TALLYING IN CNT
```

If you use /STANDARD=85, the initial value of CNT is used for all subscripts in the statement.

If you use /STANDARD=V3, SRC-STRING is examined and the data in the first INTO clause is transferred. The TALLYING item CNT is then incremented. As the statement continues to execute, any transfers to the items in the second INTO clause use the new value of CNT when their subscripts are evaluated.

D.3.5 INSPECT Statement

The /STANDARD=85 qualifier instructs the compiler to evaluate all subscripts and reference modifications in the INSPECT statement once, as the first operation when the statement is executed.

The /STANDARD=V3 qualifier only differs when you use Format 3 of the INSPECT statement. If you use Format 3, /STANDARD=V3 instructs the compiler to treat the statement as if it were two separate statements. For example, if you use /STANDARD=V3, the single statement in Example D–1 is equivalent to the two statements in Example D–2. In this example, the value of the subscript for SRC-STRING-B is dependent upon the value of TLY after the TALLYING phrase is executed.

Example D–1: INSPECT Statement Using Format 3

```
INSPECT SRC-STRING-A TALLYING TLY FOR CHARACTERS
                     REPLACING CHARACTERS BY SRC-STRING-B (TLY)
```

Example D–2: INSPECT Statement Using Formats 1 and 2

```
INSPECT SRC-STRING-A TALLYING TLY FOR CHARACTERS
INSPECT SRC-STRING-A REPLACING CHARACTERS BY SRC-STRING-B (TLY)
```

For more information, see Format 3 of the INSPECT statement in the *VAX COBOL Reference Manual*.

D.3.6 PERFORM ... VARYING ... AFTER Statement

When you use Format 4 of the PERFORM statement, the /STANDARD=85 qualifier instructs the compiler to augment the identifier associated with the VARYING phrase before setting the identifier associated with the AFTER phrase.

When you use Format 4 of the PERFORM statement, the /STANDARD=V3 qualifier instructs the compiler to set the identifier associated with the AFTER phrase before augmenting the identifier associated with the VARYING phrase.

The following example, in which one VARYING variable depends on another VARYING variable, produces different results using /STANDARD=85 and /STANDARD=V3.

```
PERFORM ... VARYING ID2 FROM 1   BY I UNTIL ID2 > 3
               AFTER ID5 FROM ID2 BY 1 UNTIL ID5 > 3
```

Table D–2 shows the different values for the identifiers in the previous example.

Table D–2: PERFORM ... VARYING ... AFTER Identifier Values

	/STANDARD=85		/STANDARD=V3	
Pass	**ID2**	**ID5**	**ID2**	**ID5**
First	1	1	1	1
Second	1	2	1	2
Third	1	3	1	3
Fourth	2	2	2	1
Fifth	2	3	2	2
Sixth	3	3	2	3
Seventh	N/A	N/A	3	2
Eighth	N/A	N/A	3	3

D.3.7 PIC P Digits

The /STANDARD=85 qualifier instructs the compiler to interpret PIC P digits as zeros in the following cases:

- Operations where the sending operand is numeric
- MOVE statements where the sending operand is numeric and contains P digits
- MOVE statements where the sending operand is numeric edited and contains P digits and the receiving item is numeric or numeric edited
- Comparison operations in which both operands are numeric

The /STANDARD=V3 qualifier does not instruct the compiler to interpret PIC P digits as zeros in numeric to alphanumeric moves.

The following example produces different results using /STANDARD=85 and /STANDARD=V3.

```
01 P    PIC 9P VALUE 10.
01 N    PIC 99.
01 X    PIC XX.
        .
        .
        .
        MOVE P TO N X.
```

If you use /STANDARD=85, both N and X are equal to 10 after the MOVE statement executes. If you use /STANDARD=V3, N is equal to 10 and X is equal to 1 followed by a space.

D.3.8 Size of Variable-Length Tables

The /STANDARD=85 qualifier instructs the compiler to determine the size of a variable-length item containing an OCCURS DEPENDING ON statement involved in a MOVE by the value of the OCCURS DEPENDING ON item. However, when the item is within the data structure containing the OCCURS clause, and the item is the target of a MOVE statement, the maximum size of the table is used.

The /STANDARD=V3 qualifier instructs the compiler to always use the maximum size of the destination table.

The following is an example of how this change produces different results:

```
01 A-TABLE PIC 99.
01 TABLE-1.
   03 B-TABLE OCCURS 1 TO 10 TIMES DEPENDING ON A-TABLE PIC X.
01 C-TABLE PIC X(10) VALUE "0123456789".
     .
     .
     .
     MOVE 5 TO A-TABLE
     MOVE C-TABLE TO TABLE-1.
```

Table D–3 shows the values of the table elements for B-TABLE after the MOVE statement using /STANDARD=85 and /STANDARD=V3.

Table D–3: Table Values After a MOVE Statement

Table Element	/STANDARD=V3	/STANDARD=85
B-TABLE(1)	0	0
B-TABLE(2)	1	1
B-TABLE(3)	2	2
B-TABLE(4)	3	3
B-TABLE(5)	4	4
B-TABLE(6)	5	Undefined
B-TABLE(7)	6	Undefined
B-TABLE(8)	7	Undefined
B-TABLE(9)	8	Undefined
B-TABLE(10)	9	Undefined

D.3.9 EXIT PROGRAM Statement

If you use /STANDARD=85, an EXIT PROGRAM statement in the body of a main program is bypassed and the statements following the EXIT PROGRAM statement are executed. If the program is a subprogram, the EXIT PROGRAM statement acts as a return.

If you use /STANDARD=V3, an EXIT PROGRAM statement is treated as a return in both main programs and subprograms.

D.3.10 New and Revised I-O Status Codes

Table D–4 explains the new and revised I-O status codes for VAX COBOL.

If you use /STANDARD=85 you receive the File Status codes listed in the column labeled 85, and your program acts accordingly.

If you use /STANDARD=V3 you receive the File Status codes listed in the column labeled V3, and your program acts accordingly.

Table D–4: New and Revised I-O Status Codes

I-O Error Condition	Status Code	
	V3	85
READ successful—record shorter than fixed file attribute.	00	04
CLOSE reel/unit attempted on nonreel/unit device.	00	07
READ fails—relative key digits exceed relative key.	00	14
WRITE fails—relative key digits exceed relative key.	00	24
OPEN I-O on file that is not mass storage.	00	37
WRITE fails—attempt to write a record of a different size than in the file description.	00	44
READ fails—no next logical record (EOF detected).	13	10
READ fails—no next logical record (EOF on OPTIONAL file).	15	10
READ fails—no valid next record (already at EOF).	16	10
READ NEXT or sequential READ—no valid next record pointer.	16[1]	46[1]
READ or START fails—optional input file not present.	25	23
READ successful—record longer than fixed file attribute.	30	04
OPEN on relative or indexed file that is not mass storage.	30	37
REWRITE fails—attempt to rewrite record of different size.	30	44
CLOSE fails—file not currently open.	93	42
DELETE or REWRITE fails—previous I-O not successful READ.	93	43
OPEN fails—file previously closed with LOCK.	94	38
OPEN fails—file created with different organization.	94	39
OPEN fails—file created with different prime record key.	94	39
OPEN fails—file created with different alternate record keys.	94	39
OPEN fails—file currently open.	94	41
READ or START fails—file not opened INPUT or I-O.	94	47
WRITE fails—file not opened OUTPUT, EXTEND, or I-O.	94	48
DELETE or REWRITE fails—file not opened I-O.	94	49
OPEN INPUT on a nonoptional file—file not found.	97	35

[1] See Section D.3.10.1.

D.3.10.1 No Valid Next Record Condition

This section describes what happens when you compile your program using /STANDARD=85 or /STANDARD=V3 and all the following conditions exist:

- The no valid next record (NVNR) condition exists.
- Your program attempts a sequential READ statement.
- Your program includes an AT END branch associated with the READ statement.

If you use /STANDARD=85, the following occurs:

- The File Status code variable, if any, for the file is set to 46.
- The statements associated with the AT END statement are not executed.
- The program terminates execution abnormally (unless you have provided for this situation with a USE AFTER STANDARD EXCEPTION procedure).

If you use /STANDARD=V3, the following occurs:

- The File Status code variable, if any, for the file is set to 16.
- The statements associated with the AT END statement are executed.
- The program continues to execute normally.

D.3.11 OPEN I-O and EXTEND Modes

If you use the /STANDARD=85 qualifier, nonoptional files opened in I-O or EXTEND mode are not created, if the file is unavailable, and a run-time error is issued.

If you use the /STANDARD=V3 qualifier, nonoptional files opened in I-O or EXTEND mode are created, if the file is unavailable.

Appendix E

Optional Programming Productivity Tools

This appendix provides an overview of optional programming productivity tools that are not included with the VAX COBOL software or the VMS operating system. Using these tools can increase your productivity as a VAX COBOL programmer. The following products are described in this appendix:

- VAX Language-Sensitive Editor (LSE) and VAX Source Code Analyzer (SCA) (Section E.1)
- VAX CDD/Plus (Section E.2)
- VAX COBOL GENERATOR (Section E.3)
- VAX Data Base Management System (VAX DBMS) (Section E.4)
- VAX DEC/Test Manager (Section E.5)
- VAX DEC/Code Management System(CMS) (Section E.6)

For information on how to purchase these tools, contact your Digital sales representative.

E.1 VAX Language-Sensitive Editor (LSE) and the VAX Source Code Analyzer (SCA)

The VAX Language-Sensitive Editor (LSE) is a powerful and flexible text editor designed specifically for software development. The VAX Source Code Analyzer (SCA) is an interactive tool for program analysis.

These products are closely integrated; generally, you can invoke SCA through LSE. LSE provides additional editing features that make SCA program analysis more efficient. In conjunction with the VAX COBOL compiler, the two tools provide a set of new enhancements supporting source code design and review.

LSE also provides the following software development features:

- Formatted language constructs, or templates, for most VAX programming languages, including VAX COBOL. These templates include the keywords and punctuation used in source programs, and use placeholders to indicate locations in the source code where additional text is optional or required.
- Commands to compile, review, and correct compilation errors from within the editor.
- Integration with VAX DEC/Code Management System (CMS). You can issue CMS commands from within the editor to make source file management more efficient.

SCA performs the following types of program analysis:

- Cross-referencing, which supplies information about program symbols and source files.
- Static analysis, which provides information on how subprograms, symbols, and files are related.

LSE and SCA together, in conjunction with VAX language compilers, provide the following software design features:

- Pseudocode support, which includes a new LSE placeholder for delimiting text that describes algorithms or design decisions. This feature allows you to write source code in shorthand, returning later to fill in code details.
- Placeholder processing, in which language compilers accept LSE placeholders and pseudocode as valid program elements during compilation. This feature allows you to test the validity of algorithms while programs are still in shorthand form.
- Comment processing, which includes design comment information in the SCA library. SCA performs cross-referencing and static analysis on this information in response to user queries.
- View support, which provides a reverse-design facility. LSE commands compress program code into overview line summaries. If you choose to edit these overview lines, the program code reflects the modifications you make.
- A report tool, callable through LSE, which can print views, standard design reports, and customized reports.

The following sections provide entry, exit, and language-specific information on the combined use of LSE and SCA.

For More Information:

- On LSE and SCA–*Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*
- On CMS–*Guide to VAX DEC/Code Management System*

E.1.1 Preparing an SCA Library

SCA stores data generated by the VAX COBOL compiler in an SCA library. The data in an SCA library contains information about all symbols, modules, and files encountered during a specific compilation of the source. You must prepare this library before you enter LSE to invoke SCA by following these steps:

1. Create a VMS directory for your SCA library. For example:

   ```
   $ CREATE/DIRECTORY PROJ:[USER.LIB1]
   ```

2. Initialize and set the library with the SCA CREATE LIBRARY command. For example:

   ```
   $ SCA CREATE LIBRARY [.LIB1]
   ```

   If you have an existing SCA library that has been initialized, you make its contents visible to SCA by setting it with the SCA SET LIBRARY command. For example:

   ```
   $ SCA SET LIBRARY [.EXISTING_SCA_LIBARAY]
   ```

A message appears in the message buffer at the bottom of your screen, indicating whether or not your SCA library selection succeeded.

3. Direct the VAX COBOL compiler to generate data analysis files by appending the /ANALYSIS_DATA qualifier to the COBOL command. For example:

   ```
   $ COBOL/ANALYSIS_DATA PG1,PG2,PG3
   ```

   This command line compiles the input files PG1.COB, PG2.COB and PG3.COB, and generates corresponding output files for each input file, with the file types OBJ and ANA. SCA puts these files in your current default directory.

4. Load the information in the data analysis files into your SCA library with the LOAD command. For example:

   ```
   $ SCA LOAD PG1,PG2,PG3
   ```

   This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

5. Once you have prepared the SCA library, you enter LSE to begin an SCA session. Within this context, the integration of LSE and SCA provides commands that you can use only within LSE.

E.1.2 Starting and Terminating an LSE or an SCA Session

To invoke LSE, issue the following command at the DCL prompt:

```
$ LSEDIT USER.COB
```

To end an LSE session, press CTRL/Z to get the LSE> prompt. If you wish to save modifications to your file, issue the EXIT command. If you do not wish to save the file or any modification to the file, issue the QUIT command.

To invoke SCA from LSE, type the SCA command that you wish to execute at the LSE> prompt, as in the following syntax:

```
LSE> command [parameter] [/qualifier...]
```

To invoke SCA from the DCL command line for the execution of a single command, you can use the following syntax:

```
$ SCA command [parameter] [/qualifier...]
```

If you have several SCA commands to invoke, you may wish to use the SCA subsystem to enter commands, as in the following syntax:

```
$ SCA
SCA> command [parameter] [/qualifier...]
```

Typing EXIT (or pressing CTRL/Z) ends an SCA subsystem session and returns you to the DCL level.

E.1.3 Compiling from Within LSE

To compile a completed VAX COBOL program, issue the following command at the LSE prompt:

```
LSE> COMPILE
```

To compile a VAX COBOL program that contains placeholders and design comments, include the following qualifiers with the previous command:

```
LSE> COMPILE $/ANALYSIS_DATA/DESIGN=(PLACEHOLDERS, COMMENTS)
```

The /ANALYSIS_DATA qualifier causes the compiler to generate a data analysis file containing source code analysis information and to provide this information to the SCA library.

The /DESIGN qualifier instructs the compiler to recognize placeholders and design comments as valid program elements. If you have also specified the /ANALYSIS_DATA qualifier, the compiler includes information on placeholders and design comments in the data analysis file.

LSE provides several commands to help you review errors and examine your source code:

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer. You can issue this command with the /REVIEW qualifier to put LSE in REVIEW mode immediately after the compilation.
REVIEW	None	Puts LSE into REVIEW mode and displays any errors resulting from the last compilation.
END REVIEW	None	Removes the buffer $REVIEW from the screen; returns the cursor to a single window containing the source buffer.
GOTO SOURCE	CTRL/G	Moves the cursor to the source buffer that contains the error.
NEXT STEP	CTRL/F	Moves the cursor to the next error in the buffer $REVIEW.
PREVIOUS STEP	CTRL/B	Moves the cursor to the previous error in the buffer $REVIEW.
	{ Down arrow Up arrow }	Moves the cursor within a buffer.

E.1.4 Notes on VAX COBOL Support

This section describes VAX COBOL-specific information for the following LSE and SCA features:

- Programming language placeholders and tokens
- Placeholder and design comment processing

E.1.4.1 Programming Language Placeholders and Tokens

LSE accepts keywords, or tokens, for all languages with LSE support, but the tokens themselves are language-defined. For example, you can expand the {IDENTIFICATION DIVISION ..} token only when using VAX COBOL.

Likewise, LSE provides placeholders, or prompt markers, for all languages with LSE support, but the specific text or choices these markers call for are language-defined. For example, you see the [DATA DIVISION ..] placeholder only when using VAX COBOL.

NOTE

> Keywords such as IF, OPEN, and WRITE can be tokens as well as placeholders; therefore, any time you are in the VAX COBOL language environment you can type one of these words and press CTRL/E to expand the construct.

Remember that braces ({}) enclose required placeholders; brackets ([ ]) enclose optional placeholders. When you erase an optional placeholder, LSE also deletes any associated text before and after that placeholder.

You can use the SHOW TOKEN and SHOW PLACEHOLDER commands to display a list of all VAX COBOL tokens and placeholders, or a particular token or placeholder. For example:

```
LSE> SHOW TOKEN IF                    {lists the token IF}
LSE> SHOW TOKEN                       {lists all tokens   }
```

To copy the listed information into a separate file, first issue the appropriate SHOW command to put the list into the $SHOW buffer. Then issue the following command:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename.filetype
```

To obtain a hard copy of the list, use the PRINT command at DCL level to print the file you created.

E.1.4.2 Placeholder and Design Comment Processing

While all languages with LSE support provide placeholder processing, each language defines specific contexts in which placeholders can be accepted as valid program code. VAX COBOL defines contexts for ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION placeholders. The following list shows the valid contexts within a VAX COBOL ENVIRONMENT DIVISION.

- The entire ENVIRONMENT DIVISION
- The CONFIGURATION SECTION
- The INPUT-OUTPUT SECTION
- The SOURCE-COMPUTER paragraph
- The OBJECT-COMPUTER paragraph
- The SPECIAL-NAMES paragraph
- The FILE-CONTROL paragraph
- The I-O-CONTROL paragraph
- Any clause in the FILE-CONTROL paragraph
- Any clause in the SPECIAL-NAMES paragraph
- Any clause in the I-O-CONTROL paragraph

The following list shows the valid contexts within a VAX COBOL DATA DIVISION.

- The entire DATA DIVISION
- The SUBSCHEMA SECTION
- The FILE SECTION
- The WORKING-STORAGE SECTION

- The REPORT SECTION
- The DB subschema description
- A complete FD file description
- A complete SD file description
- A complete RD file description
- A complete record description
- A complete report group description

The following list shows the valid contexts within a VAX COBOL PROCEDURE DIVISION.

- Any paragraph or section name
- Any paragraph or section name references in the GO TO, PERFORM, SORT, MERGE, or ALTER statements
- Any conditional expressions referenced in the IF, EVALUATE, PERFORM, or SEARCH statement
- Any complete statement

VAX COBOL accepts optional LSE placeholders in any context where optional syntax is allowed. The following example shows some contexts in which LSE placeholders and design comments might appear in the design of a VAX COBOL program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. paycheck.
*+
* FUNCTIONAL DESCRIPTION:
*
*   This program computes the amount of an employee's salary and
*   prints a paycheck.
*
* FORMAL PARAMETERS:
*
*   name -
*          The name of the employee, lastname first
*
* IMPLICIT INPUT PARAMETERS:
*
*   [description or none]
*
* IMPLICIT OUTPUT PARAMETERS:
*
*   [description or none]
*
* RETURN VALUE:
*
*   [description or none]
*
* SIDE EFFECTS
*
*   [description or none]
*
* DESIGN:
*
*           [tbs]
*
* [logical properties]
*
* [other files required]
*
```

```
* [other tags]
*-

DATA DIVISION.
WORKING-STORAGE SECTION.
01  weekly-salary PIC S9(5)V99.
LINKAGE SECTION.
01  name    PIC X(20).

PROCEDURE DIVISION USING name.
PO.
    «Fetch the employee's record.»

* Compute paychecks differently for salaried and hourly employees.
    IF «employee is salaried»
    THEN
        «Use a fixed weekly salary from the employee's record.»
    ELSE
        «Fetch the number of regular and overtime hours worked.»
        «Compute the weekly pay.»
    END-IF

    «Print the paycheck»
END PROGRAM paycheck.
```

VAX COBOL support for placeholder and design comment processing includes the following language-specific stipulations:

- Pseudocode placeholders are designated with double right- and left-angle brackets, << >> or « ».
- Comment processing is limited to the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, and DATA DIVISION.

Placeholders are either optional or required. Required placeholders, which are delimited by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, which are delimited by brackets ([ ]), represent places in the source code where you can either provide additional constructs or delete the placeholder.

Additionally, when you use VAX COBOL with LSE, the expanded code might include ellipses (. . .), or vertical bars (|). Syntax constructs followed by ellipses indicate that the constructs can be repeated. A vertical bar between constructs indicates that you must choose one of the constructs.

There are three types of LSE placeholders:

Type of Placeholder	Description
Terminal	Provides text that describes valid replacements for the placeholder
Nonterminal	Expands into additional language constructs
Menu	Provides a list of options corresponding to the placeholder

LSE commands allow you to manipulate tokens and placeholders. These commands and their default key bindings are as follows:

Command	Key Binding	Function
EXPAND	CTRL/E	Expands a placeholder
UNEXPAND	PF1-CTRL/E	Reverses the effect of the most recent placeholder expansion
GOTO PLACEHOLDER/FORWARD	CTRL/N	Moves the cursor to the next placeholder

Command	Key Binding	Function
GOTO PLACEHOLDER/REVERSE	CTRL/P	Moves the cursor to the previous placeholder
ERASE PLACEHOLDER/FORWARD	CTRL/K	Erases a placeholder
UNERASE PLACEHOLDER	PF1-CTRL/K	Restores the most recently erased placeholder
None	Down arrow	Moves the indicator down through a menu
None	Up arrow	Moves the indicator up through a menu
None	{ ENTER RETURN }	Selects a menu option

E.1.5 LSE and SCA Examples

The following example shows how you can use the LSE tokens and placeholders to create a PERFORM statement within a VAX COBOL program. The example shows expansions of the following features:

- Data definition
- IF statement

Instructions and explanations precede each example, and an arrow (←) indicates the line in the code where an action has occurred. See Section E.1.4.2 for the commands that manipulate tokens and placeholders.

When you use LSE to create a new VAX COBOL program, the initial string, {program module} appears at the top of the screen. If you expand the initial string, the following appears on your screen:

```
[Module level comments]
{program}...
```

Delete the [Module level comments] placeholder and then expand the {program}... placeholder. When you finish, your screen should look as follows:

```
{IDENTIFICATION DIVISION ..}
[ENVIRONMENT DIVISION ..]
[DATA DIVISION ..]
[PROCEDURE DIVISION ..]
[contained program]...
END PROGRAM {program-name}.
[program]...
```

To begin to create the program EXAMPLE, expand the {IDENTIFICATION DIVISION ..} placeholder and type EXAMPLE over the placeholder {program-name}. At this point, your screen should look as follows:

```
IDENTIFICATION DIVISION.   <--
PROGRAM-ID. EXAMPLE [IS [INITIAL] [COMMON] PROGRAM] [WITH IDENT
{module-vers-string}].  <--
[AUTHOR. [comment-entry]]  <--
[INSTALLATION. [comment-entry]]  <--
[DATE-WRITTEN. [comment-entry]]  <--
[DATE-COMPILED. [comment-entry]] <--
[SECURITY. [comment-entry]]  <--
[Routine level comments]  <--
[ENVIRONMENT DIVISION ..]
[DATA DIVISION ..]
[PROCEDURE DIVISION ..]
[contained-program]...
END PROGRAM {program-name}. <--
[program]...
```

E.1.5.1 Data Definition

This section provides an example of how to create a simple data definition.

Beginning where you were after typing EXAMPLE, erase the placeholders down to [DATA DIVISION ..]. Then expand the [DATA DIVISION ..] placeholder. Erase the placeholders down to [WORKING-STORAGE SECTION] and expand it. Type 01 over the [record-description-entry] placeholder and expand 01. Your screen should then look as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.             <--
WORKING-STORAGE SECTION.   <--
01  [{data-name}|FILLER]   <--
    [REDEFINES {other-data-item}]
    [IS EXTERNAL]
    [IS GLOBAL]
    [PICTURE IS {character-string}]
    [USAGE IS {COMP|COMP-1|COMP-2|COMP-3|DISPLAY|INDEX|POINTER}]
    [SIGN IS {LEADING|TRAILING} [SEPARATE CHARACTER]]
    [SYNCHRONIZED [LEFT|RIGHT]]
    [JUSTIFIED RIGHT]
    [BLANK WHEN ZERO]
    [VALUE IS {value-option}].
[record-description-entry]...
[LINKAGE SECTION ..]
[REPORT SECTION ..]
[PROCEDURE DIVISION ..]
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

Move to the {data-name} placeholder and type A-DATA-DEFINITION. Erase the placeholders down to the [PICTURE IS {character-string}] placeholder and expand it. Move to the {character-string} placeholder and type 999. Move to the [USAGE IS {COMP | COMP-1 | COMP-2 | COMP-3 | DISPLAY | INDEX | POINTER}] placeholder and expand it. At this point, your screen should look as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION   <--
    PICTURE IS 999      <--
    USAGE IS {COMP|COMP-1|COMP-2|COMP-3|DISPLAY|INDEX|POINTER}  <--
    [SIGN IS {LEADING|TRAILING} [SEPARATE CHARACTER]]
    [SYNCHRONIZED [LEFT|RIGHT]]
    [JUSTIFIED RIGHT]
    [BLANK WHEN ZERO]
    [VALUE IS {value-option}].
[record-description-entry]...
[LINKAGE SECTION ..]
[REPORT SECTION ..]
[PROCEDURE DIVISION ..]
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

The {COMP | COMP-1 | COMP-2 | COMP-3 | DISPLAY | INDEX | POINTER} placeholder needs to be expanded to display a menu and select the DISPLAY option. Erase the placeholders down to the [VALUE IS {value-option}] placeholder and expand it. Type 3 over the {value-option} placeholder. Erase the placeholders down to the [PROCEDURE DIVISION ..] placeholder. At this point, the record definition is complete and your screen should look as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY  <--
    VALUE IS 3.       <--
[PROCEDURE DIVISION ..]
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

E.1.5.2 IF Statement

This section demonstrates how to use LSE to create a simple VAX COBOL statement. An IF statement is developed in the example that follows.

Move your cursor to the [PROCEDURE DIVISION ..] placeholder shown in the previous example and expand it. Erase the placeholders down to the [section] placeholder and expand it. Type SECTION-ONE. Erase the placeholders down t the {paragraph} placeholder and expand it. Type PARAGRAPH-1. At this point your screen should look as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY
    VALUE IS 3.

PROCEDURE DIVISION.   <--
SECTION-ONE SECTION.  <--
PARAGRAPH-1.          <--
    [sentence]...
[paragraph]...
[section]...
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

Move to the [sentence]... placeholder, type IF, and then issue an EXPAND command. Your screen should look as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY
    VALUE IS 3.
```

```
PROCEDURE DIVISION.
SECTION-ONE SECTION.
PARAGRAPH-1.
    IF {conditional-expression}  <--
    THEN
        {then-clause}
    [ELSE {else-part}]
    [END-IF]
    [sentence]...
[paragraph]...
[section]...
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

Move to the {conditional-expression} placeholder and expand it. Choose {arithmetic-expression} IS [NOT] {POSITIVE | NEGATIVE | ZERO} from the menu that appears and expand it.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY
    VALUE IS 3.

PROCEDURE DIVISION.
SECTION-ONE SECTION.
PARAGRAPH-1.
    IF {arithmetic-expression} IS [NOT] {POSITIVE|NEGATIVE|ZERO} <--
    THEN
        {then-clause}
    [ELSE {else-part}]
    [END-IF]
    [sentence]...
[paragraph]...
[section]...
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

Type A-DATA-DEFINITION over the {arithmetic-expression} placeholder. Move to the [NOT] placeholder and delete it. Move the cursor to the {POSITIVE | NEGATIVE | ZERO} placeholder and expand it. This brings up a menu. Choose POSITIVE. Your screen should then look as follows, with the cursor on the {then-clause} placeholder:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY
    VALUE IS 3.
```

```
PROCEDURE DIVISION.
SECTION-ONE SECTION.
PARAGRAPH-1.
    IF A-DATA-DEFINITION IS POSITIVE   <--
    THEN
        {then-clause}
    [ELSE {else-part}]
    [END-IF]
    [sentence]...
[paragraph]...
[section]...
[contained-program]...
END PROGRAM EXAMPLE.
[program]...
```

Expand the {then-clause} placeholder; this brings up a menu. Choose the {statement} placeholder, which also brings up a menu. Type "DISPLAY A-DATA-DEFINITION.". Erase through the [program]... placeholder.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATA-DEFINITION
    PICTURE IS 999
    USAGE IS DISPLAY
    VALUE IS 3.

PROCEDURE DIVISION.
SECTION-ONE SECTION.
PARAGRAPH-1.
    IF A-DATA-DEFINITION IS POSITIVE
    THEN                             <--
        DISPLAY A-DATA-DEFINITION.   <--
END PROGRAM EXAMPLE.                 <--
```

At this point the program is complete. You can then issue the LSE COMPILE command. The LSE COMPILE command allows you to compile the contents of a buffer without leaving the LSE session. When you exit the LSE editing session, you can then link and run the program.

E.2 VAX CDD/Plus

VAX COBOL supports CDD/Plus. CDD/Plus is a new version of the VAX Common Data Dictionary (CDD) that is completely compatible with previous versions while providing new features such as the ability to create field definitions and the ability to track the use of dictionary entities.

This section provides information on using CDD/Plus with VAX COBOL, including the following:

- An overview of data dictionaries
- An introduction to CDD/Plus concepts
- Information on creating, accessing, and referencing data definitions
- Information on creating compiled module entities and viewing CDD/Plus relationships

E.2.1 Overview of Data Dictionaries

A data dictionary system provides you with the ability to create, analyze, and administer metadata. Metadata describes data and describes how that data is used. Metadata keeps track of the location, type, format, and size of the actual data.

Data dictionaries usually provide the following features:

- Ensure the integrity of shared metadata and the procedures used to analyze, maintain, manage, and design business metadata
- Provide a centralized repository for information management shops
- Offer a dynamic aid to software application development

The dictionary represents metadata in the form of dictionary definitions. A CDD/Plus dictionary definition can contain various attributes and can be related to another CDD/Plus dictionary definition. Some commonly used dictionary definitions are fields, records, and databases.

You can store and maintain the actual data values outside the data dictionary in several ways; for example, with a database management system like Rdb/VMS, in RMS files, in CMS libraries, or even off line.

Using a data dictionary enables many products to share the metadata and data. The more you enforce dictionary usage, the more accurate and consistent your data will be. Since many products can share the metadata stored in a data dictionary, using a data dictionary also reduces data redundancy.

E.2.2 CDD/Plus Features

CDD/Plus provides the following features:

- Compatibility with previous versions of the software
- Single user interface
- Distributed dictionary access
- Field-level data descriptions
- Relationships between dictionary definitions
- Pieces tracking
- Data security and integrity
- Call interface

E.2.3 CDD/Plus Concepts

This section provides information on CDD/Plus concepts.

E.2.3.1 CDD/Plus Dictionary Formats

CDD/Plus supports metadata stored in two formats: metadata that you manipulate with the Common Dictionary Operator (CDO) and metadata that you manipulate with the Dictionary Management Utility (DMU).

- CDO format

Dictionaries created in CDO format can store not only definitions, but also information about how the definitions are related. When you install CDD/Plus you create a CDO dictionary on your system. You create and manipulate definitions in CDO dictionaries through the CDO utility.

- DMU Dictionaries

  CDD/Plus supports DMU dictionaries and the utilities that manipulate DMU dictionary definitions—DMU, CDDL and CDDV. If a DMU dictionary does not already exist on your system, CDD/Plus creates one during the installation procedure. You create DMU dictionary definitions through the DMU utility; however, you can read DMU dictionary definitions through either DMU or CDO. Versions of CDD/Plus prior to Version 4.0 support dictionaries in DMU format only.

VAX COBOL supports both formats.

For more information on data dictionary formats, refer to the CDD/Plus documentation.

E.2.3.2 Dictionary Path Names

VAX COBOL allows two types of valid path name parameters when referring to dictionary definitions. They differ in the method of specifying the dictionary origin or root.

- Dictionary anchor path name

  An anchor path name begins with a VMS directory specification as the dictionary origin; it specifies the VMS directory that contains the dictionary. This type of path name is valid for CDO-format dictionary definitions only. For example:

  ```
  MYNODE::DISK$2:[MYDIRECTORY]PERSONNEL.EMPLOYEES_REC
  ```

  The node, device, and directory components are optional. For example, if the dictionary is located at your current default directory, you can specify:

  ```
  PERSONNEL.EMPLOYEES
  ```

- CDD$TOP path name

  You use this to refer to either DMU- or CDO-format dictionary definitions. The path origin is always CDD$TOP. This is known as the DMU naming convention. For example:

  ```
  CDD$TOP.PERSONNEL.EMPLOYEES_REC
  ```

E.2.3.3 Dependency Recording

When dependency recording is in effect, the CDO dictionary is updated at compilation time to show what data entities and relationships defined therein are used by the compiled module (in other words, the data dependencies created by the compilation).

To take advantage of dependency recording, you must specify the /DEPENDENCY_DATA qualifier in the COBOL command when you compile the module. Additionally, the current CDD$DEFAULT must refer to definitions in CDO format.

For more information on dependency recording, see Sections Section E.2.5 and Section E.2.6.

E.2.3.4 Compiled Module Entities

When you compile a program with the /DEPENDENCY_DATA qualifier, the compiler creates a construct known as a compiled module entity in the CDO-format dictionary. A compiled module entity is created for each separately compiled program. The name of the entity is the PROGRAM-ID name with hyphens translated to underscores. Compiled module entities are put in the dictionary associated with the current CDD$DEFAULT directory.

In addition, the compiler creates a temporary file entity for each .OBJ file generated by the compilation. Each compiled module entity contains a pointer to a file entity, and several compiled module entities can point to the same file entity. At the end of the compilation, the file entity does not actually exist in the dictionary. However, information correlating the compiled module entity and the object file entity does exist in the dictionary.

E.2.3.5 Entities

An entity is a piece of information represented in the dictionary. A CDO entity definition can contain various attributes, or characteristics, and can be related to another CDO definition. You can define, store, and access many types of entity definitions in CDO dictionaries. Some commonly used entity definitions include fields, records, and databases.

Field Definitions

A field definition is the smallest unit of metadata that can be created and accessed in the dictionary. Because each piece of metadata is a separately addressable entity, VAX CDD/Plus is known as a field-level dictionary. Field definitions typically include information about the data type and size, and other optional attributes.

Field definitions can be simple data structures or complex subscripted structures. They can be combined to form various record definitions and can be accessed individually from several of the VAX layered products. You need store only one copy of a particular definition that is used by various sources.

VAX CDD/Plus keeps track of dictionary definition usage at the field level. Therefore, you can easily show which dictionary entities (such as records) make use of a particular field definition. When a field definition is changed, you can identify which entities may be affected by the change and which entities need to be redefined in order to access the changed field. This ability to track entities is known as pieces tracking. (See Section E.2.3.7.)

Record Definitions

A record definition is a dictionary entity that typically consists of a grouping of field definitions. You can combine field and record definitions into complex record structures.

Dictionary Directories

You organize your dictionary definitions by creating a dictionary directory structure. Directories map each definition name to a certain location.

A directory is not a dictionary definition, but contains dictionary definitions and other directories. Field and record definitions are grouped in named directories. Dictionary directories are similar in concept to VMS directories; they allow you to hierarchically organize and group the definitions in your dictionary. You can use search lists and wildcards to manipulate data in directories.

E.2.3.6 Relationships

CDO creates relationships when you connect two CDO data definitions in some way.

For example, you can base the definition of a new field on a field definition that already exists in a CDO dictionary. Similarly, you can relate a group of field definitions to a record definition by including the field names in the record definition. You do not need to define these relationships; CDO automatically creates them for you when you create your field and record definitions in CDO.

You can establish a relationship between two CDO definitions in different CDO dictionaries that are distributed on different devices on a single node, on different nodes in a VAXcluster environment, or on nodes connected by a local or wide area network. For example, you can create a record definition in one CDO dictionary that includes field definitions contained in another.

VAX COBOL enables you to create relationships by using the /DEPENDENCY_DATA qualifier when you compile your program. Creation and use of these relationships is discussed in Sections Section E.2.5 and Section E.2.6.

E.2.3.7 Pieces Tracking

Because CDD/Plus keeps track of all CDO dictionary usage, you can easily find out which other dictionary entities make use of a particular field definition. When you want to change a field definition, you can confirm which definitions the change may affect and which entities you must redefine to access the changed field definition.

For example, if you use a particular field definition in several different record definitions, and the record definitions are accessed in turn by other records and by an Rdb/VMS database, CDD/Plus can locate all the uses of the single CDO field definition. You find out about these interrelationships with the SHOW commands. The SHOW USES, SHOW USED_BY, and SHOW WHAT_IF commands help you to keep track of dependent and interrelated definitions and to assess the impact of changes.

You can control changes to your definitions in two ways: you can change the original definition to take effect immediately, or you can create a new version and allow users to incorporate the change over time. When you change or make a new version of a definition, dependent definitions that do not automatically include the change (such as Rdb/VMS databases) are flagged with an informational message about the change. Messages allow you to warn users when a new version of a dictionary definition exists or when inconsistencies may exist between the dictionary and external copies.

E.2.3.8 Distributed Dictionary Access

Through the CDO, you can access metadata in CDO dictionaries and directories that are located on different devices on a single node, on different nodes in a VAXcluster, or on nodes connected by a local or wide area network

You can access metadata in all these places as a single logical dictionary, provided that you have the appropriate access rights. In addition, you can access your DMU dictionaries from CDO. This versatility affords you greater security for sensitive parts of a dictionary and greater flexibility for storing large dictionary files.

E.2.3.9 Data Security and Integrity

To protect dictionary files from unauthorized users, CDD/Plus provides the database administrator with the tools to grant or deny dictionary definition access rights. The CDD/Plus protection provisions for CDO definitions are consistent with Rdb/VMS and VMS protection schemes.

Integrity, the completeness, accuracy, and consistency of definitions, is a critical factor in the success of any dictionary operation. For this reason, CDD/Plus provides journaling capabilities that automatically protect your dictionary sessions from system failures.

E.2.3.10 CDD/Plus Call Interface

You can make direct calls to the CDD/Plus entry points from VAX COBOL programs. Using the call interface allows you to directly access CDO dictionaries without using the CDO utility. For more information on the call interface, refer to the CDD/Plus documentation.

E.2.4 Creating Data Definitions

In CDD/Plus, you can create data definitions in both DMU and CDO format. Definitions that are stored in CDO format enable you to create the definitions on the field level. Additionally, you can create relationships between fields and records and the programs that access the definitions.

In the following example, the CDD/Plus directory is set to SALES and NAME, STREET, CITY, STATE, and ZIP are created as fields using CDO:

```
CDO> SET DEFAULT SALES
CDO> DEFINE FIELD NAME DATATYPE IS TEXT SIZE IS 25 CHARACTERS.
CDO> DEFINE FIELD STREET DATATYPE IS TEXT SIZE IS 20 CHARACTERS.
CDO> DEFINE FIELD CITY DATATYPE IS TEXT SIZE IS 20 CHARACTERS.
CDO> DEFINE FIELD STATE DATATYPE IS TEXT SIZE IS 2 CHARACTERS.
CDO> DEFINE FIELD ZIP DATATYPE IS TEXT SIZE IS 5 CHARACTERS.
```

The fields can then be related to any record, by creating the record and including the required fields.

The following example creates two records in CDO format, CUSTOMER_ADDRESS_RECORD and EMPLOYEE_ADDRESS_RECORD. Both of these record definitions are located in the CDD/Plus directory SALES and share the same field definitions of NAME, STREET, CITY, STATE, and ZIP.

```
CDO> DEFINE RECORD CUSTOMER_ADDRESS_RECORD.
cont> NAME.
cont> STREET
cont> STATE.
cont> ZIP.
cont> END RECORD.
CDO> DEFINE RECORD EMPLOYEE_ADDRESS_RECORD.
cont> NAME.
cont> STREET
cont> STATE.
cont> ZIP.
cont> END RECORD.
```

You can then access the record definitions from your VAX COBOL program using the COPY FROM DICTIONARY statement.

For more information on creating data definitions, refer to the CDD/Plus documentation.

E.2.5 Accessing Data Definitions

You access record definitions stored in CDD/Plus from your VAX COBOL program using the the COPY FROM DICTIONARY statement. Additionally, you can reference data definitions in CDO dictionaries using the RECORD DEPENDENCY statement.

E.2.5.1 Using the COPY FROM DICTIONARY Statement

The COPY FROM DICTIONARY statement enables you to copy data definitions from CDD/Plus. When you use the COPY FROM DICTIONARY statement, the data definitions are included in your program. For example, the following VAX COBOL statements access the data definitions in Section E.2.4:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MASTER-FILE.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY "DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD" FROM DICTIONARY.
COPY "DEVICE:[VMS_DIRECTORY]SALES.EMPLOYEE_ADDRESS_RECORD" FROM DICTIONARY.
    .
    .
    .
```

In the previous example, the device and VMS directory specified refer to the device and VMS directory where CDD/Plus is located.

If you compiled this program with the /LIST and /COPY_LIST command line qualifiers, the source listing would appear as follows:

```
  1        IDENTIFICATION DIVISION.
  2        PROGRAM-ID. MASTER-FILE.
  3        DATA DIVISION.
  4        WORKING-STORAGE SECTION.
  5        COPY "DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD" FROM DICTIONARY.
 6L       *
 7L       *_CDD$TOP.SALES.CUSTOMER_ADDRESS_RECORD
 8L       *
 9L        01 CUSTOMER_ADDRESS_RECORD.
10L           02 NAME                PIC X(25).
11L           02 STREET              PIC X(20).
12L           02 CITY                PIC X(20).
13L           02 STATE               PIC X(2).
14L           02 ZIP                 PIC X(5).
15         COPY "NODE::DEVICE:[VMS_DIRECTORY]SALES.EMPLOYEE_ADDRESS_RECORD" FROM DICTIONARY.
16L       *
17L       *_CDD$TOP.SALES.EMPLOYEE_ADDRESS_RECORD
18L       *
19L        01 EMPLOYEE_ADDRESS_RECORD.
20L           02 NAME                PIC X(25).
21L           02 STREET              PIC X(20).
22L           02 CITY                PIC X(20).
23L           02 STATE               PIC X(2).
24L           02 ZIP                 PIC X(5).
   .
   .
   .
```

E.2.5.2 Using the RECORD DEPENDENCY Statement

The RECORD DEPENDENCY statement allows you to explicity create a CDD/Plus relationship between a compiled module entity and a dictionary entity in CDO format. When you use this qualifier, the entity is not copied into your program and does not appear in the source listing.

The RECORD DEPENDENCY statement can only appear in the PROCEDURE DIVISION and the statement is ignored unless you compile your program with the /DEPENDENCY_DATA qualifier.

The following is an example of how to use the RECORD DEPENDENCY statement:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MASTER-FILE.
    .
    .
    .
PROCEDURE DIVISION.
A0100.
    .
    .
    .
RECORD DEPENDENCY "DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD"
       TYPE IS "CDD$COMPILED_DEPENDS_ON" IN DICTIONARY.
    .
    .
    .
```

When this statement is executed, a CDD/Plus relationship is made between the compiled module entity MASTER_FILE and the CUSTOMER_ADDRESS_RECORD. The relationship type is CDD$COMPILED_DEPENDS_ON which indicates that the program MASTER-FILE depends on CUSTOMER_ADDRESS_RECORD. This information is stored in the dictionary.

For more information on the RECORD DEPENDENCY statement, refer to the *VAX COBOL Reference Manual.*

E.2.6 Using the /DEPENDENCY_DATA Qualifier

The /DEPENDENCY_DATA qualifier enables you to create compiled module entities in CDO-format dictionaries. To create compiled module entities you must:

- Enable dependency recording by compiling your program with the /DEPENDENCY_DATA qualifier
- Direct the COBOL compiler to a CDO-format dictionary or a CDD/Plus compatibility dictionary

If you use the /DEPENDENCY_DATA qualifier when compiling programs that contain COPY FROM DICTIONARY statements, the following entities are created:

- A compiled module entity in CDD/Plus.
- A temporary object file entity (this entity is not stored in the directory). The name of this entity is the same as the object file name.
- A relationship between the compiled module entity and object file entity. The type of this relationship is CDD$IN_FILE.
- A relationship between the compiled module entity and the records copied. The relationship type would be CDD$COMPILED_DEPENDS_ON.

If you use the /DEPENDENCY_DATA qualifier when compiling programs that contain RECORD DEPENDENCY statements, a relationship between the compiled module entity and the entity specified in the RECORD DEPENDENCY statement is created and stored in CDD/Plus. You can explicity state the relationship type as either CDD$COMPILED_DEPENDS_ON or CDD$COMPILED_DERIVED_FROM.

E.2.7 Viewing CDD/Plus Relationships

You can view CDD/Plus relationships from CDO by using the following variations of the CDO SHOW command:

- USED_BY
- USES
- WHAT_IF

SHOW USED_BY

The SHOW USED_BY command displays the CDO objects that are used by the specified object. Using this command is helpful when you are planning to modify data definitions. You can use this command to show the relationships between fields and records. For example, if you use the SHOW USED_BY command for the CUSTOMER_ADDRESS_RECORD created in Section E.2.4, the following information would be displayed:

```
CDO> SHOW USED_BY CUSTOMER_ADDRESS_RECORD
Members of DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD;1
|    DEVICE:[VMS_DIRECTORY]NAME;4       (Type : FIELD)
|    |    via CDD$DATA_AGGREGATE_CONTAINS
|    DEVICE:[VMS_DIRECTORY]STREET;1     (Type : FIELD)
|    |    via CDD$DATA_AGGREGATE_CONTAINS
|    DEVICE:[VMS_DIRECTORY]CITY;1       (Type : FIELD)
|    |    via CDD$DATA_AGGREGATE_CONTAINS
|    DEVICE:[VMS_DIRECTORY]STATE;1      (Type : FIELD)
|    |    via CDD$DATA_AGGREGATE_CONTAINS
|    DEVICE:[VMS_DIRECTORY]NAME;4       (Type : FIELD)
|    |    via CDD$DATA_AGGREGATE_CONTAINS
```

This example shows that the record CUSTOMER_ADDRESS_RECORD uses the fields NAME, STREET, CITY, STATE, and ZIP.

For more information on the SHOW USED_BY command, refer to the CDD/Plus documentation.

SHOW USES

The SHOW USES command displays the CDO objects that use the specified object. Using this command is helpful when you are planning to modify data definitions. You can use this command to show the relationships between dictionary entities and the compiled modules that access them. For example, if you used the SHOW USES command for the CUSTOMER_ADDRESS_RECORD created in Section E.2.4, and compiled the sample code in Section E.2.5.1 with the /DEPENDENCY_DATA qualifier, the following information would be displayed:

```
CDO> SHOW USES CUSTOMER_ADDRESS_RECORD
Owners of DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD;1
|    DEVICE:[VMS_DIRECTORY]MASTER_FILE;1       (Type : CDD$COMPILED_MODULE)
|    |    via CDD$COMPILED_DEPENDS_ON
```

This example shows that the record CUSTOMER_ADDRESS_RECORD is used by the program MASTER-FILE.

For more information on the SHOW USES command, refer to the CDD/Plus documentation.

SHOW WHAT_IF

The SHOW WHAT_IF command displays the ancestor objects that might be affected if the specified object is modified. Using this command is helpful when you are planning to modify data definitions. You can use this command to show the relationships between dictionary entities. For example, if you used the SHOW WHAT_IF command for the CUSTOMER_ADDRESS_RECORD created in Section E.2.4 and compiled the sample code in Section E.2.5.1, with the /DEPENDENCY_DATA qualifier, the following information would be displayed:

```
CDO> SHOW WHAT_IF CUSTOMER_ADDRESS_RECORD
Signaled owners of DEVICE:[VMS_DIRECTORY]SALES.CUSTOMER_ADDRESS_RECORD;1
|    DEVICE:[VMS_DIRECTORY]MASTER_FILE;1      (Type : CDD$COMPILED_MODULE)
|    |    via CDD$COMPILED_DEPENDS_ON
```

This example shows that a change to CUSTOMER_ADDRESS_RECORD might effect the program MASTER-FILE.

For more information on the SHOW WHAT_IF command, refer to the CDD/Plus documentation.

E.2.8 VAX COBOL Support for CDD/Plus Data Types

Table E–1 lists the data types supported by CDD/Plus and VAX COBOL.

Table E–1: CDD/Plus Data Types

Data Type	CDD/Plus	COBOL
UNSPECIFIED	S	U
SIGNED BYTE	S	W
UNSIGNED BYTE	S	W
SIGNED WORD	S	S
UNSIGNED WORD	S	W
SIGNED LONGWORD	S	S
UNSIGNED LONGWORD	S	S
SIGNED QUADWORD	S	S
UNSIGNED QUADWORD	S	W
SIGNED OCTWORD	S	W
UNSIGNED OCTWORD	S	W
F_FLOATING	S	S
F_FLOATING COMPLEX	S	W
D_FLOATING	S	S
D_FLOATING COMPLEX	S	W
G_FLOATING	S	W
G_FLOATING COMPLEX	S	W
H_FLOATING	S	W

S—The facility fully supports the data type.
W—The facility translates the data type into one it supports and issues diagnostics.
U—The data type is unsupported and the facility issues a fatal diagnostic.

(continued on next page)

Table E–1 (Cont.): CDD/Plus Data Types

Data Type	CDD/Plus	COBOL
H_FLOATING COMPLEX	S	W
UNSIGNED NUMERIC	S	S
LEFT OVERPUNCHED NUMERIC	S	S
LEFT SEPARATE NUMERIC	S	S
RIGHT OVERPUNCHED NUMERIC	S	S
RIGHT SEPARATE NUMERIC	S	S
PACKED DECIMAL	S	S
ZONED NUMERIC	S	W
BIT	S	W
DATE	S	W
TEXT	S	S
VARYING STRING	S	W
POINTER	S	S
VIRTUAL FIELD	S	W
SEGMENTED STRING	S	W

S—The facility fully supports the data type.
W—The facility translates the data type into one it supports and issues diagnostics.
U—The data type is unsupported and the facility issues a fatal diagnostic.

For more information on data types, refer to the CDD/Plus documentation.

E.3 VAX COBOL GENERATOR

The VAX COBOL GENERATOR is a screen-oriented program generator that produces VAX COBOL source programs. You can use the VAX COBOL GENERATOR as a productivity tool for the creation and maintenance of data processing applications.

The VAX COBOL GENERATOR uses a graphical interface that allows you to create or modify a program by choosing icons that represent the components making up the program (such as menus and screens). From this input, the VAX COBOL GENERATOR produces a VAX COBOL source program that can be used like any other source program.

The VAX COBOL GENERATOR can also be used for rapid prototyping to produce a program that can later be expanded and refined to become a production application. This makes the VAX COBOL GENERATOR an effective and efficient programming tool for producing and maintaining VAX COBOL programs.

The VAX COBOL GENERATOR can produce programs that call subprograms written in other VMS languages, as well as many Run-Time Library routines and system services. Similarly, programs produced by the VAX COBOL GENERATOR can be called by other VMS products adhering to the VAX Calling Standard.

The default screen interactions utilize the VAX COBOL extensions to the ACCEPT and DISPLAY statements. However, the VAX COBOL GENERATOR can also produce screen applications that use other screen packages, such as the VAX Forms Management System (FMS).

To define an application in the VAX COBOL GENERATOR environment, you select and place icons (representing various parts of the application) in the screen work area for expansion. You define data and procedural flow by connecting these parts together. The VAX COBOL source code generated is then compiled by the VAX COBOL language processor and linked by the VMS Linker. These applications can then be executed on any valid VMS operating system.

E.3.1 VAX COBOL GENERATOR Features

The following list briefly describes some of the VAX COBOL GENERATOR's major features:

- Integrated COBOL programming environment

  From the VAX COBOL GENERATOR you can access other software (such as VAX Rdb/VMS and VAX CDD/Plus), and create forms and reports without leaving the GENERATOR's development environment.

- Sophisticated graphic interface

  The VAX COBOL GENERATOR implements a sophisticated graphic interface. This feature allows you to define and manipulate program parts and relationships by using icons to create nodes. You then connect the nodes using lines showing data and procedural flow and give them names.

- Top-down program design

  The GENERATOR enables you to create programs starting at the highest level. This approach promotes logical and orderly development. The VAX COBOL GENERATOR supports this by allowing you to define structural nodes that represent a complex function. You can then decompose this representation into parts.

  It is not necessary to expand all of the structural nodes for the GENERATOR to produce COBOL programs. This feature allows you to evaluate partially completed programs while work continues on the development of unexpanded structural nodes.

- Single points of control for data used

  The VAX COBOL GENERATOR provides you with a data dictionary to store common data definitions. This provides a single point of control so that data elements used in single, or multiple, applications need only be changed once. These applications can then be regenerated incorporating the changes.

  Optionally, you can use CDD/Plus record definitions created by the GENERATOR and stored in CDD/Plus.

  Libraries are another single point of control for changes. Form, file, local storage, report, and procedure node definitions, as well as a data dictionary, can be stored in a library and referenced from multiple programs.

- Comprehensive on-line assistance

  Help information is available at the DCL level, as well as from within the GENERATOR at all times.

- Automatic documentation of the application design

  The MAP feature provides you with an overall map of the program. Information contained in the map includes node locations, node types, and errors that occurred during the last generation. You can print out a copy of the map using the PRINT MAP command, and use it as a reference.

The GENERATE DOCUMENT command automatically creates a text file of all the information about the nodes in the program or the library. This file can also be printed out and used as a reference.

- Ability to document the source code of the generated program

  VAX COBOL source code generated by the VAX COBOL GENERATOR is automatically documented in the GENERATOR's environment, which also allows you to add your own comments to the code. The GENERATOR prompts you for comments during the development cycle.

For more information on the VAX COBOL GENERATOR, refer to the VAX COBOL GENERATOR documentation.

E.4 VAX Data Base Management System (VAX DBMS)

VAX DBMS is a multiuser general-purpose CODASYL-compliant database management system. VAX DBMS is used for accessing and administrating databases ranging in complexity from simple hierarchies to complex networks with multilevel relationships. VAX DBMS supports full concurrent access in a multiuser environment without compromising the integrity and security of your database.

VAX DBMS features include the following:

- Full concurrent access in a multiuser environment
- Record locking and journaling
- Automatic transaction and verb rollback
- Multiple-database support (one or more databases for processing)
- Online backup
- Integration with CDD/Plus
- Schema, subschema, storage schema, and security schema data definition languages (DDLs)

For more information, refer to the VAX DBMS documentation.

E.5 VAX DEC/Test Manager

VAX DEC/Test Manager helps test software during development and maintenance. This tool automates the organization, execution and review of tests and allows several developers to use one set of tests at the same time.

With DEC/Test Manager you can describe your tests, organize them by assigning them to groups, and choose combinations of tests to run by test name or by group. DEC/Test Manager executes the tests selected and then compares the results with the expected results.

For more information, refer to the VAX DEC/Test Manager documentation.

E.6 VAX DEC/Code Management System (CMS)

The VAX DEC/Code Management System (CMS) is a program librarian for software development and evolution. It is comprised of a set of commands that enable you to manage files of an ongoing project.

CMS enables you to do the following tasks:

- Store ASCII text files in a project library
- Retrieve previous generations of files stored in CMS
- Obtain a report of file modifications, including when, why, and by whom the modifications were made
- Determine the origin of each line of a file, either as an annotated listing or as comments in the file
- Manage concurrent modifications and merge separately developed modifications
- Store related files together as a single element
- Relate the generation of one element to the corresponding generations of other elements

For more information, refer to the VAX DEC/Code Management System documentation.

Index

A

Abort function, 3–4
ACCEPT statement
  EDITING phrase, 17–31
Access
  database key identifier, 15–3
  record search, 15–3
  set owner, 15–3
Access mode
  BATCH, 15–39
  changing, 8–13
  CONCURRENT, 15–39
  default, 8–13
  dynamic, 8–12
  EXCLUSIVE, 15–39
  lock, 15–41
  PROTECTED, 15–39
  random, 8–12
  sequential, 8–12
  UPDATE, 15–40
Address expression, 3–1
/ANALYSIS qualifier
  in compile command, 2–8
/ANALYSIS_DATA qualifier
  use with PDF, E–4
ANSI format, 2–2
  See Developing programs
ANSI-to-terminal
  command string, 19–2
  format conversion, 19–1
/ANSI_FORMAT qualifier
  in compile command, 2–2, 2–8
ANY clause, 15–59
APPLY clause, 8–2, 20–6
Argument-passing mechanisms
  BY CONTENT, 18–8
  BY DESCRIPTOR, 18–9
  BY OMITTED, 18–9
  BY REFERENCE, 18–8
  BY VALUE, 18–8
Arguments
  optional, 18–28, 18–33
Arithmetic expressions
  processing, 4–12
Arithmetic operations
  results of rounding off, 4–9
Arithmetic statements, 4–8, 4–12
  advantages over COMPUTE, 20–4
  common errors, 4–11
  intermediate results, 4–9
  with GIVING phrase, 4–11
Arithmetic statements (Cont.)
  with SIZE ERROR phrase, 4–10
ASSIGN DCL command, 2–44
At end condition
  planning for, 12–1
AT END phrase, 15–56, 15–59
Attributes
  COBOL program, 18–4
/AUDIT qualifier
  in compile command, 2–9
Automatic ROLLBACK, 15–42
AUTOMATIC set membership class, 15–46
Auxiliary keypad keys, 17–21

B

BATCH
  access mode, 15–39
BATCH RETRIEVAL
  allow mode, 15–40
  usage mode, 15–40
BATCH UPDATE
  allow mode, 15–40
  usage mode, 15–40
Binary search, 6–18 to 6–25
  function and results, 6–20
  requirements for, 6–19
  with AT END statement, 6–20
  with keys, 6–20
BIND command, 15–59
Bottom margin, 16–15
/BRIEF qualifier
  in LINK command, 2–31
Byte boundaries
  effects on storage allocation, 6–8

C

COBOL–81 Subset Flagger, C–1 to C–11
  alignment of COMP data items, C–9, C–11
  flagging procedures, C–2
  producing compatible COBOL–81 source programs, C–1
  /STANDARD=PDP11 qualifier, C–1, C–2
  unsupported language elements, C–3, C–8
  VAX COBOL/COBOL–81 differences, C–2, C–8
Calling
  procedure, 18–3
  sequence examples, 18–39
Calling subprograms
  See Interprogram communication

- CALL statement
  - nesting, 18–6
  - use of, 18–5
- Cascade delete, 15–49
- CDD
  - See CDD/Plus
- CDD$DEFAULT, 15–4
- CDD$TOP, 15–2
- CDD/Plus, 15–4, 15–8, E–12 to E–22
  - accessing data definitions
  - creating compile module entities, E–19
  - distributed dictionary access, E–16
  - entities, E–15
  - field definitions, E–15
  - integrity, E–17
  - path names, E–14
  - pieces tracking, E–16
  - protection, E–17
  - record definitions, E–15
  - relationships, E–16
  - using the /DEPENDENCY_DATA qualifier, E–19
- Cell
  - contents, 10–1
  - location in the file, 10–1
  - numbering, 10–1
  - relative record number, 10–1
  - size, 10–1
- Character attributes for terminal screen, 17–8
- /CHECK qualifier
  - in compile command, 2–9
- Class, 5–5
- CLASS-NAME, 5–5
- Class tests, 5–4
  - numeric, 4–4
- CMS
  - integration with LSE, E–1
- COB$SWITCHES, 2–43
- COBOL, 1–1
  - run-time errors, 2–43
- COBOL command, 2–4, 2–6 to 2–15
  - qualifiers, 2–7 to 2–15
- COBOL data manipulation language
  - See Data manipulation language
- COBOL GENERATOR, E–22
- Code Management System, E–25
- Command line
  - common VAX COBOL errors, 2–16
- Command qualifiers
  - with the COBOL command, 2–4, 2–6
  - with the LINK command, 2–4
- COMMIT RETAINING statement, 15–41
- COMMIT statement, 15–3
- Communication
  - See Interprogram communication
  - contained programs, 18–12
- Comparing operands, 5–4
- Compilation summary, 2–25
- Compile
  - listing, 2–17
- Compile command qualifiers
  - /ANALYSIS, 2–8
  - /ANSI_FORMAT, 2–2, 2–8
  - /AUDIT, 2–9
  - /CHECK, 2–9
  - /CONDITIONALS, 2–9
  - /COPY_LIST, 2–10
- Compile command qualifiers (Cont.)
  - /CROSS_REFERENCE, 2–10
  - /DEBUG, 2–10
  - /DEPENDENCY_DATA, 2–11
  - /DESIGN, 2–11
  - /DIAGNOSTICS, 2–11
  - /FIPS[=74], 2–11
  - /FLAGGER, 2–12, D–1
  - /INSTRUCTION_SET, 2–13, D–3
    - See also /INSTRUCTION_SET qualifier
  - /LIST, 2–13
  - /MACHINE_CODE, 2–13
  - /MAP, 2–13
  - /OBJECT, 2–14
  - /SEQUENCE_CHECK, 2–14
  - /STANDARD, 2–14
  - /TRUNCATE, 2–15
  - /WARNINGS, 2–15
- Compiler error messages, 2–16, 2–17
- Compiler implementation limitations, A–1
- Compiling
  - /DEBUG, 3–3
  - VAX COBOL DML program, 15–4
- Compiling programs, 2–4, 2–6
  - conditional compilation lines, 2–16
- Concatenating items
  - nonnumeric data, 7–1
- CONCURRENT
  - access mode, 15–39
  - allow mode, 15–40
- CONCURRENT RETRIEVAL
  - usage mode, 15–40
- CONCURRENT UPDATE
  - usage mode, 15–40
- Conditional compilation lines, 2–16
- Conditional expression
  - database, 15–4
- /CONDITIONALS qualifier
  - in compile command, 2–9
- Condition values
  - returned, 18–34
  - signaled, 18–34
- CONNECT statement, 15–3
  - usage, 15–46
- Contained program
  - COMMON clause, 18–13
  - communication with, 18–12
  - reading a listing, 2–28
- Contained subprogram, 18–1
- CONTENT
  - argument-passing mechanism, 18–8
- CONTINUE statement, 2–45
- Control footing, 16–3
- Control heading, 16–3
- CONTROL KEY IN phrase, 17–21
- Controlling index, 6–19, 6–20
- Conventional Report
  - logical page, 16–7
  - page advancing, 16–8
- CONVERSION clause, 17–10, 17–13
- COPY FROM DICTIONARY statement, E–18
- COPY statement, 2–5
- /COPY_LIST qualifier, 2–17
  - in compile command, 2–10
- Creating an executable image
- /CROSS_REFERENCE qualifier, 2–17, 2–20

/CROSS_REFERENCE qualifier (Cont.)
- in compile command, 2–10
- in LINK command, 2–31

CTRL/C, 3–4
CTRL/Y, 3–4
- interrupting debugger, 3–4

CTRL/Z
- exiting debugger, 3–4
- use with LSE, E–3

Currency indicator, 15–58
- affected by, 15–30
- as pointer, 15–34
- changing the value of a, 15–30
- current of realm, 15–30
- current of record type, 15–32
- current of set type, 15–31
- freeing, 15–50
- navigation aid, 15–33
- null, 15–30
- place markers, 15–30
- realm, 15–30
- record type, 15–32
- run unit, 15–32
- saving, 15–34
- set type, 15–31

Currency indicators, 15–30, 15–38
Current of set type, 15–31

D

Database
- See also Keeplist and DML statements
- administrator (DBA), 15–4, 15–9
- areas, 15–29
- CDD$DEFAULT, 15–4
- CDD$TOP, 15–4
- CDD/Plus, 15–8
- COMMIT RETAINING statement, 15–41
- COMMIT statement, 15–41
- compiling a VAX COBOL DML program, 15–4
  - using the /MAP qualifier, 15–5
- conditional expression, 15–4
- CONNECT statement, 15–46
- copying database records, 15–5
- creating, 15–6
- creating a VAX COBOL DML program, 15–4
- creating new record relationships, 15–96
- creating records, 15–3
- currency indicator
  - freeing, 15–50
    - example, 15–51
  - null, 15–30
  - record type, 15–32
  - run unit, 15–32
  - Run unit, 15–30
  - saving, 15–34
- currency indicators, 15–30, 15–33
  - example, 15–33
  - realm, 15–30
  - See also Currency indicators, 15–30
  - set type, 15–31
- Database Control System, 15–5
- Data Definition Language, 15–4
- data item, 15–8
- data manipulation language, 15–2
  - See also DML
- DBQ utility, 15–59, 15–60

Database (Cont.)
- DB statement, 15–5
- debugging DML programs, 15–58
  - sample session, 15–60
- definition of, 15–6
- degree of record locking, 15–8
- deleting records, 15–3, 15–48
- demonstration package, 15–1
  - creating, 15–2
  - loading, 15–2
- designing, 15–6
- DISCONNECT statement, 15–46
- entry points into, 15–15
- ERASE ALL statement, 15–48
- ERASE statement, 15–49
- erasing records for, 15–3
- error-handling routines, 15–56
- establishing a known currency condition, 15–50
- exception conditions, 15–56
  - AT END phrase, 15–56
  - ON ERROR phrase, 15–56
  - translating DB-CONDITION values, 15–58
  - USE statement
    - example, 15–57
- FETCH NEXT, 15–53
- FETCH statement
  - qualifying, 15–55
- FIND ALL statement, 15–53
- finding records, 15–3
- FIND NEXT, 15–53
- FIND statement
  - qualifying, 15–55
- forked sets, 15–16
- IF EMPTY clause, 15–44
- keeplists, 15–36
- key, 15–8
- key identifier access, 15–3
- linking a VAX COBOL DML program, 15–5
- locking out other users, 15–40
- logical division, 15–29
- maintaining, 15–6
- many-to-many relationships
  - records of the same type, 15–20
  - two types of records, 15–18
- MEMBER test condition, 15–43
- minimizing lock out, 15–40
- modifying members of sorted sets, 15–44
- modifying records, 15–3
- multiset relationships, 15–18
  - example, 15–92, 15–96
- NEW.ROO, 15–2
- one-to-many relationsships
  - records of the same type, 15–25
- OWNER test condition, 15–43
- physical division, 15–29
- place markers, 15–3
- processing control, 15–2
- program examples, 15–75
  - accessing database information, 15–89, 15–91
  - backing up a database, 15–83
  - creating relationships, 15–92, 15–96
  - displaying database information, 15–89, 15–91
  - populating a database, 15–75
- programming techniques, 15–39
  - READY usage modes, 15–39

- Database
  - programming techniques
    - READY usage modes (Cont.)
      - See also READY statement, Access mode, and Usage mode
  - protected access, 15–40
  - quiet points, 15–38
    - figure, 15–38
  - reading a subschema map listing, 15–68
    - example, 15–69, 15–73
  - read-only access, 15–40
  - realms, 15–29
  - RECONNECT statement, 15–47
  - record, 15–8
  - record deleting, 15–3
  - record locking, 15–29, 15–40
    - See also Record locking
    - individual records, 15–41
    - READY statement, 15–41
    - realm level, 15–41
    - record level, 15–41
    - releasing locks, 15–52
  - record retrieval, 15–3
  - record transfers, 15–47
  - record types, 15–9
  - RETAINING clause, 15–34
    - example, 15–34
  - ROLLBACK statement, 15–41
  - running a VAX COBOL DML program, 15–5
  - run unit, 15–29
    - concurrent, 15–40
  - schema, 15–6
  - set occurences, 15–12
  - set relationships, 15–13, 15–14
  - sets, 15–12
  - set types, 15–9
  - simple sets, 15–14, 15–15
  - simultaneous access to, 15–40
  - storage address of the record, 15–8
  - storage schema, 15–6
  - stream, 15–7
    - multiple streams, 15–7
  - subschema, 15–6
  - SUB-SCHEMA section, 15–5
  - system-owned sets, 15–15
  - testing DML programs, 15–58
    - sample session, 15–60
  - transaction, 15–38
  - transactions
    - figure, 15–38
  - unrestricted access, 15–40
  - updating records, 15–3
  - user work area, 15–5
- Database Control System (DBCS), 15–2, 15–32
  - major functions, 15–5
- Database Query utility (DBQ), 15–59
- Data definition, 15–2
- Data Definition Language, 15–4
  - DDL utility, 15–4
- Data Definition Language, 15–2
- Data dictionary
  - See also CDD/Plus
  - metadata, E–13
  - overview, E–13
- Data Division
  - accessing another program's, 18–8
- Data items
  - index, 6–17
- Data manipulation language (DML), 15–2
  - See also DML statements
- Data movement, 5–5, 5–11
- Data-name
  - subscript with, 6–16
- Data names map, 2–21
- Data organization, 5–1
- Data Storage Description Language (DSDL), 15–6
- Data testing, 5–3, 5–5
- Data types
  - scaling and mixing, 20–4
- DB-CONDITION, 15–4
  - translating values, 15–56
- DBCS
  - See Database Control System
- DB-CURRENT-RECORD-ID, 15–4
- DB-CURRENT-RECORD-NAME, 15–4
- DBG$PROCESS, 3–3
- Dbkey, 15–8
  - value, 15–37
- DBM$SIGNAL, 15–58
- DBMDEMO, 15–2
- DBMS, E–24
- DB statement, 15–4, 15–5, 15–8, 15–59
- DCL commands
  - ASSIGN, 2–44
  - CONTINUE, 2–45
  - DEASSIGN, 2–45
  - DEFINE, 2–44
  - for program development, 2–2 to 2–29
- DDL utility
  - See Data Definition Language
- DEASSIGN DCL command, 2–45
- Debugger, 3–1 to 3–15, B–5
  - command summary, 3–5
  - compiling and linking with the Debugger, 3–3
  - conepts, 3–1
  - features, 3–2
  - Getting started, 3–2
  - Help, 3–2
  - initialization, 3–3
  - interrupt, 3–4
  - issuing commands, 3–4
  - prompt (DBG>), 3–3
  - sample debugging session, 3–10
  - starting and ending a session, 3–3
  - VAX COBOL support, 3–9
- Debugging, 3–1 to 3–15
  - LINKING with the Debugger, 2–32
  - sample session, 3–11 to 3–15, 15–60
  - use of desk-checking, B–5
  - use of faulty data, B–5
  - using conditional compilation lines, 2–16
  - VAX COBOL DML programs, 15–58
- Debugging configuration
  - default, 3–3
- /DEBUG qualifier, 3–3
  - in compile command, 2–10
  - in LINK command, 2–32
- DEC/CMS
  - See Code Management System
- Decimal truncation
  - reasons for avoiding, 20–4
  - /TRUNCATE compiler qualifier, 20–4

Declarative procedures
EXTEND, 12–8
file name, 12–8
INPUT, 12–8
INPUT-OUTPUT, 12–8
OUTPUT, 12–8
samples, 12–9
using, 12–7
DEC/Test Manager, E–24
DECwindows Compiler Interface, 2–6
DEFAULT phrase, 17–18
Default user object module libraries, 2–36
DEFINE DCL command, 2–44
Defining tables, 6–1 to 6–11
/DEPENDENCY_DATA qualifier
in compile command, 2–11
using, E–19
DESCRIPTOR
argument-passing mechanism, 18–9
/DESIGN qualifier
in compile command, 2–11
use with PDF, E–4
Desk-checking, B–5
Detail lines, 16–3
Developing programs, 2–1, 2–45
VAX COBOL GENERATOR
choosing a reference format, 2–1, 2–2
ANSI format, 2–2
terminal reference format, 2–1, 2–2
converting reference format, 2–2
creating a program, 2–5
editors, 2–5
reference format, 2–2
/DIAGNOSTICS qualifier
in compile command, 2–11
DISCONNECT statement, 15–3, 15–46
DIVIDE statement, D–6
DML
See Data manipulation language
See DML statements
DML statements, 15–2
COMMIT, 15–3, 15–41
CONNECT, 15–3, 15–46
DISCONNECT, 15–3, 15–46
ERASE, 15–3, 15–48
FETCH, 15–3, 15–52
FIND, 15–3, 15–52
FREE, 15–3, 15–36, 15–50
GET, 15–3
KEEP, 15–3, 15–36
MODIFY, 15–3
READY, 15–29
RECONNECT, 15–3, 15–47
ROLLBACK, 15–3, 15–42
STORE, 15–3
DMU Utility, 15–1
DSDL
See Data Storage Description Language
DUPLICATE clause, 15–59
DWCI
See DECwindows Compiler Interface

E

Edited moves
nonnumeric data, 5–8
Editing
numeric symbols, 4–7
EDITING phrase, 17–31
Editors, 2–3
See also Developing programs
LSE, E–1 to E–12
Elementary data items
nonnumeric, 5–2
Elementary moves
nonnumeric, 5–7
numeric edited, 4–7
Empty member record, 15–18
END PROGRAM, 2–28
Entry
SUB-SCHEMA SECTION, 15–4
Entry points
into the database, 15–15
ERASE statement, 15–3
usage, 15–48
Erasing
a line on the terminal screen, 17–3
database records, 15–3
entire terminal screen, 17–3
to end of line on terminal screen, 17–3
to end of terminal screen, 17–3
Error messages, B–1 to B–17
compiler messages, B–12
REFORMAT Utility, 19–4
Errors
common VAX COBOL command line, 2–16
I/O, B–7
in arithmetic statements, 4–11
in MOVE statements, 4–8
in size, 4–10
inspecting data, 7–33
nonnumeric MOVE statements, 5–10
run-time, B–1
sample error, B–1
run-time input/output, B–7
STRING statements, 7–5
unstringing data, 7–16
EXCLUSIVE
access mode, 15–39
allow mode, 15–40
EXCLUSIVE RETRIEVAL
usage mode, 15–40
EXCLUSIVE UPDATE
usage mode, 15–40
/EXECUTABLE qualifier
in LINK command, 2–32
Execution
interrupting with CTRL/C, 3–4
Execution control
transferring, 18–5
EXIT debugger command, 3–4
EXIT PROGRAM statement, D–10
use of, 18–5
Expression processing
arithmetic, 4–12
EXTERNAL clause
use of, 18–4

F

- FETCH NEXT statement loop, 15–53
- FETCH statement, 15–3
  - AT END phrase, 15–56
  - ON ERROR phrase, 15–56
  - usage, 15–52
- File
  - See also, Indexed file, Relative file, and Sequential file
  - See also Access mode
  - See also Record
  - ACCEPT and DISPLAY statements, 8–16
  - ACCEPT statement, 8–18
  - backing up files, 8–16
  - choosing file organization, 8–11
  - choosing record mode access, 8–11
  - connector, 8–8
  - data record formatting differences, 8–16
  - data type differences, 8–15
  - defining a disk, 8–8
  - defining a magnetic tape, 8–9
  - description entry, 8–8
  - design considerations, 8–7
  - device independence, 8–9
    - using logical names, 8–10
  - DISPLAY statement, 8–18
  - fixed-length records, 8–4
  - handling, 8–1 to 8–18
    - identifying a file, 8–8
  - logical names, 8–17
  - low-volume I/O, 8–16
    - logical names, 8–16
  - mnemonic names, 8–16
  - multiple openings in same program, 8–14
  - opening and closing a, 8–14
  - optimization, 20–6
  - print-controlled, 8–7
  - processing, 8–1
  - record access modes, 8–12
  - record management, 8–1
  - record size, 8–3
  - sharing
    - See File sharing
  - SPECIAL-NAMES paragraph, 8–16
  - variable-length records, 8–5
- File attributes
  - defining, 8–2
- File description, 8–2, 8–4, 8–8, 12–3 to 12–4
  - indexed files, 11–1
  - relative files, 10–4
  - sequential files, 9–3
- File names map, 2–20
- File organization, 8–1
  - See also, Indexed file, Relative file, and Sequential file
  - advantages and disadvantages, 8–12
  - default, 8–12
  - indexed, 8–11
  - relative, 8–11
  - sequential, 8–11
  - specifying, 8–2
- File sharing, 13–1 to 13–18
  - See also Protecting records and Record locking
  - checking file operations, 13–9
  - concepts, 13–1

File sharing (Cont.)

  - describing types of access streams, 13–5
  - determining the access mode, 13–4
  - ensuring successful file sharing, 13–3
  - indicating the access allowed to other streams, 13–5
  - specifying OPEN EXTEND, 13–11
    - with a shared indexed file, 13–12
    - with a shared relative file, 13–11
    - with a shared sequential file, 13–11
  - summarizing related file-sharing criteria, 13–6
  - using file status codes, 13–9
  - using RMS special registers, 13–9
  - using VMS file protection, 13–3
- File specification
  - how RMS builds a COBOL, 8–14
  - keeping as a variable, 8–9
  - overriding at run-time, 8–9
  - variable, 8–9
- File Status values
  - for RMS, 12–5
  - for VAX COBOL, 12–3
- FIND ALL option, 15–53
- FIND NEXT statement loop, 15–53
- FIND statement, 15–3
  - AT END phrase, 15–56
  - ON ERROR phrase, 15–56
  - usage, 15–52
- /FIPS[=74] qualifier
  - in compile command, 2–11
- FIXED set membership class, 15–47
- /FLAGGER qualifier, 2–12, D–1 to D–3
  - description, D–1
  - FIPS levels, D–2
  - options, D–2
    - HIGH_FIPS, D–2
    - INTERMEDIATE_FIPS, D–2
    - MINIMUM_FIPS, D–2
    - OBSOLETE_FIPS, D–2
    - OPTIONAL_FIPS, D–2
    - REPORT_WRITER, D–2
    - SEGMENTATION, D–2
    - SEMENTATION_1, D–2
- Footing area, 16–15
- FOR DB-EXCEPTION statement
  - usage, 15–57
- Forked set, 15–16
- Format
  - ANSI, 2–2
  - conversion of ANSI to terminal, 19–1
  - conversion of terminal to ANSI, 19–2
  - terminal, 2–1
- Form control bytes, 8–7
- Forms
  - See Video forms
- FREE statement, 15–3, 15–36
  - usage, 15–50
- /FULL qualifier
  - in LINK command, 2–32
- Function calls
  - for system routines, 18–33

G

- GET statement, 15–3
- GLOBAL clause
  - definition of, 18–14

GLOBAL clause (Cont.)
  sharing database resources with, 18–15
  sharing data with, 18–14
  sharing files with, 18–15
  sharing other resources, record-name, 18–15
  sharing other resources, report-name, 18–16
  usage, 18–14
GO TO DEPENDING
  advantages of using, 20–4
GROUP INDICATE clause, 16–85
Group items, 5–2
  nonnumeric, 5–2
Group moves
  nonnumeric data, 5–7

H

Help
  online, 3–2
HELP debugger command, 3–2

I

I/O, errors, B–7
I/O exception conditions handling, 12–1 to 12–10
I/O handling
  AT END condition, 12–1
  INVALID KEY condition, 12–2
  using declarative procedures, 12–7
  using file status values, 12–3
IF EMPTY clause, 15–44
IF OWNER clause, 15–44
/INCLUDE qualifier
  in LINK command, 2–33
Index data item, 6–18
  declaration, 6–17
  modifying with SET, 6–17
  where defined, 6–3
Indexed file, 11–1 to 11–14
  See also Optimization
  access modes, 11–3
  alternate key, 11–1
  at end condition, 12–1
  creating, 11–4
  defining, 11–3
  design considerations, 11–2
  File Status values, using, 12–3
  I/O statements, 11–3
  index, 11–2
  invalid key condition, 12–2
  INVALID KEY condition, 12–2
  key, 11–1
    length, 11–1
    location, 11–1
  open modes, 11–3
  optional key, 11–1
  organization, 11–1
    advantages, 8–12
    disadvantages, 8–12
  population, initial, 11–5
  primary key, 11–1
  reading, 11–6
    dynamically, 11–9
    randomly, 11–7
    sequentially, 11–6
  reorganizing, 11–5
  updating, 11–10
Indexed file
  updating (Cont.)
    dynamically, 11–14
    randomly, 11–12
    sequentially, 11–10
Indexes, 6–1
  initializing, 6–17
Indexing
  advantages, 20–4
  efficiency order, 20–4
Indexing compared with subscripting, 20–4
INITIAL clause
  use of, 18–4
Initializing
  tables, 6–11 to 6–14
INSERTION option, 15–3
Inspecting data
  active/inactive arguments, 7–22
  BEFORE/AFTER phrase, 7–18
  finding a match, 7–23
  implicit redefinition, 7–19
  INSPECT operation, 7–21
  interference in tally argument list, 7–26
  replacing phrase, 7–29
  scanner setting, 7–22
  tally argument, 7–24
  tally counter, 7–24
  TALLYING phrase, 7–24
INSPECT statement, 7–17 to 7–33, D–8
  AFTER phrase, 7–18
  BEFORE phrase, 7–18
  common errors, 7–33
  CONVERTING phrase, 7–33
  function, 7–17
  how the INSPECT statement inspects data, 7–21
  implicit redefinition, 7–19
  REPLACING phrase, 7–17
    interference in the replacement argument list, 7–32
    replacement argument, 7–31
    replacement argument list, 7–31
    replacement value, 7–30
    search argument, 7–30
  TALLYING phrase, 7–17, 7–24
    argument list, 7–25
    arguments, 7–24
    counter, 7–24
    interference in the tally arguemnt list, 7–26
/INSTRUCTION_SET qualifier, D–3 to D–5
  determining an, D–4
  GENRIC option, D–5
  how subsetting may affect VAX COBOL programs, D–4
  in compile command, 2–13
  NODECIMAL_STRING option, D–5
  other considerations, D–5
  selecting an option, D–4 to D–5
  subsetting overview, D–4
Interference
  in replacement argument list, 7–32
  in tally argument list, 7–26
Intermediate result item, 4–9
Intermediate results
  for arithmetic statements, 4–9
Interprogram communication, 18–1 to 18–40
  See also calling routines
  accessing another program's Data Division, 18–8

Interprogram communication
  accessing another program's Data Division (Cont.)
    Linkage Section, 18–10
    USING phrase, 18–8
  additional information, 18–40
  calling non-COBOL programs, 18–19
    example, 18–20
  calling routines, 18–27
    defining the argument, 18–28
    determining the type of call, 18–27
    locating the result, 18–36
    making the call, 18–29
  calling system service routines, 18–29
    checking condition values, 18–34
    EXTERNAL phrase, 18–35
    in a procedure call, 18–34
    library return status, 18–36
    VALUE IS clause, 18–35
  calling system service routinescondition value
    symbols, 18–36
  CALL statement
    passing arguments, 18–8
  common language environment, 18–23
    using COBOL in, 18–23
  contained programs, 18–12
    COMMON clause, 18–13
    GLOBAL clause, 18–14
    shaing database resources, 18–15
    sharing data, 18–14
    sharing files, 18–15
    sharing other resources, 18–15
    USE procedures, 18–16
  examples, 18–37
  multiple run-unit
    examples, 18–1
  multiple run-unit concepts, 18–1
    calling procedures, 18–3
    definition, 18–1
  program attributes, 18–4
    EXTERNAL clause, 18–4
    INITIAL clause, 18–4
  sharable images, 18–36
  transferring execution control, 18–5
    CALL statement, 18–5
      nesting, 18–6
    EXIT PROGGAM statement, 18–5
  VAX Procedure Calling and Condition Handling
    Standard, 18–24
    argument list, 18–25
    function value, 18–25
    register usage, 18–24
    stack usage, 18–24
  VMS Run-Time Library routines, 18–25
  VMS System Services routines, 18–26
Interrupt
  debugging session, 3–4
  execution of command, 3–4
  execution of program, 3–4
Invalid key condition
  planning for, 12–2
INVALID KEY phrase, B–12
Invoking
  debugger, 3–3
I-O-CONTROL paragraph, 13–3, 13–13, 20–5 to
  20–9

J

Junction record, 15–18
Justified moves
  nonnumeric data, 5–9

K

Keeplist
  activating, 15–36
  contents, 15–36
  deallocating, 15–36
  emptying, 15–36
  usage, 15–36
Keeplist description entry (LD), 15–2, 15–36, 15–59
KEEP statement, 15–3, 15–36
Key codes, 17–28
Key identifier access
  database, 15–3
Keys
  ascending, 6–6
  descending, 6–6
Keywords
  using with LSE, E–4

L

Language
  data description, 15–6
  data storage description, 15–6
  subschema data description, 15–6
Language-Sensitive Editor
  See LSE
Libraries
  condition value symbols, 18–36
  default user object modules, 2–36
  object modules, 2–35
  return status, 18–36
  system, 2–36, 2–37
  system-supplied object module libraries, 2–35
  VAX DBMS, 15–5
/LIBRARY qualifier
  in LINK command, 2–34
LINAGE clause
  usage, 16–15
LINAGE-COUNTER, 16–16
  special register, 16–16
Linage-file report
  bottom margin, 16–15
  footing area, 16–15
  logical page, 16–14
  makeup, 16–14
  page advancing, 16–16
  page body, 16–15
  page-overflow condition, 16–16
  printing the, 16–20
  top margin, 16–15
LINE-COUNTER, 16–30
Linkage section
  example, 18–10
Linkage Section
  description, 18–10
LINK command, 2–30
  See also LINK command qualifiers
  Positional qualifiers, 2–33
  syntax, 2–30
LINK command qualifiers, 2–31

LINK command qualifiers (Cont.)
/BRIEF, 2–31
/CROSS_REFERENCE, 2–31
/DEBUG, 2–32
/EXECUTABLE, 2–32
/FULL, 2–32
/INCLUDE, 2–33
object module library, 2–34
/LIBRARY, 2–34
/MAP, 2–32
/OPTIONS, 2–34
/SHARE, 2–37, 2–41
/TRACEBACK, 2–33
/USERLIBRARY, 2–33
Linking
/DEBUG, 3–3
Linking programs, 2–4, 2–29 to 2–42
See also LINK command
error messages, 2–41
object module libraries, 2–35
using an object module library, 2–34
using system-supplied object module libraries, 2–35
Listing, 2–17
contained programs, 2–28
object code, 2–25
source program, 2–18
/LIST qualifier
in compile command, 2–13
Literal subscripts
accessing tables, 6–14
definition, 6–14
Locking records, 15–38
See also Protecting records
Logical name, 8–10 to 8–11, 8–16 to 8–18
CDD$DEFAULT, 15–4
CDD$TOP, 15–4
DBM$SIGNAL, 15–58
Logical page
conventional report, 16–7
definition, 16–7
spacing, 16–7
structure, 16–7
Loops
FETCH NEXT, 15–53
FIND NEXT, 15–53
LSE, E–1
compiling source code, E–3
defined, E–1
entering source code, E–3
examples, E–8
exiting, E–3
features of, E–1
invoking, E–3
using placeholders, E–4
using tokens, E–4
VAX COBOL programs (example), E–8 to E–12
VAX COBOL support
tokens and placeholders, E–4

M

/MACHINE_CODE qualifier, 2–25
in compile command, 2–13
MANDATORY set membership class, 15–47
MANUAL, 15–46
Many-to-many relationship, 15–18
Map
data name, 2–21
file name, 2–20
procedure name, 2–21
/MAP qualifier, 2–17, 2–20, 15–5, 15–6
in compile command, 2–13
in LINK command, 2–32
Member record, 15–9
empty member, 15–18
insertion point, 15–31
MEMBER test condition, 15–43
MERGE statement, 14–1, 14–9 to 14–20
example, 14–9
sample programs, 14–9
Messages
REFORMAT Utility error, 19–4
Modifying
database records, 15–3
MODIFY statement, 15–3
Modular Programming Standard, 18–40
MOVE CORRESPONDING statement, 5–10
MOVE statement, 4–5, 5–6
common errors, 4–8
Multiple delimiters
for unstringing data, 7–12
Multiple operands
in ADD and SUBTRACT statements, 4–11
Multiple receiving items
for unstringing data, 7–6
Multiple sending items
for stringing data, 7–1
Multiset relationships, 15–18

N

Nested subprogram, 18–1
Nesting
CALL statements, 18–6
level, 2–28
NEXT, 15–31
Node, 15–1
NO ECHO clause, 17–17
Non-COBOL programs
run unit, 18–19
Nonnumeric data, 5–1 to 5–11
classes, 5–3
common nonnumeric item move errors, 5–10
concatenating items, 7–1
data movement, 5–5
data organization, 5–1
edited moves, 5–8
elementary moves, 5–7
group items, 5–2
group moves, 5–7
justified moves, 5–9
MOVE CORRESPONDING statement, 5–10
MOVE statement, 5–6
multiple receiving items, 5–9
special characters, 5–2
STRING statement, 7–1
subscripted moves, 5–10
testing, 5–3
class tests, 5–4
comparing operands, 5–4
relational operator descriptions, 5–3
using reference modification, 5–11

- Nonnumeric data items
  - elementary, 5–2
  - testing
    - relational tests, 5–3
- Null currency indicator, 15–30
- Numeric class tests, 4–4
- Numeric data, 4–1 to 4–13
  - arithmentic statements
    - multiple operands
      - ADD statement, 4–11
      - SUBTRACT statement, 4–11
  - arithmetic expression processing, 4–12
  - arithmetic statements, 4–8
    - common errors, 4–11
    - GIVING phrase, 4–11
    - intermediate rules, 4–9
  - common numeric move errors, 4–8
  - elementary numeric edited moves, 4–7
  - elementary numeric moves, 4–5
  - evaluating numeric items, 4–2
  - invalid values, 4–2
  - MOVE statement, 4–5
  - NOT ON SIZE ERROR phrase, 4–11
  - /[NO]TRUNCATE qualifier, 4–9
  - numeric edited data rules, 4–7
  - numeric editing symbols, 4–7
  - ON SIZE ERROR phrase, 4–10
  - optimizing, 20–4
  - REMAINDER phrase, 4–10
  - representation of, 20–4
  - ROUNDED phrase, 4–9, 4–10
  - sign conventions, 4–1
  - storing, 4–1
  - testing
    - class test, 4–2, 4–4
    - relational operators description, 4–3
    - relational test, 4–2
    - relational tests, 4–2
    - sign test, 4–2, 4–3
    - success/failure tests, 4–4
- Numeric edited data
  - rules for, 4–7
- Numeric edited data item
  - contents, 4–7
  - description, 4–7
- Numeric edited moves
  - elementary, 4–7
- Numeric editing, 4–7
- Numeric items
  - invalid values in, 4–2
  - testing, 4–2
    - class test, 4–2
    - relation test, 4–2
    - sign test, 4–2
- Numeric moves
  - elementary, 4–5

O

- Object code
  - listing, 2–25
- Object module
  - creating, 2–6
  - default user libraries, 2–36
  - defining the search order for, 2–36
  - using libraries, 2–35
- Object module (Cont.)
  - using system-supplied libraries, 2–35
- /OBJECT qualifier
  - in compile command, 2–14
- Occurrence
  - owner record, 15–12
  - record, 15–12
  - set, 15–9, 15–12
- OCCURS clause
  - indexes, 6–1
  - keys, 6–1
- OMITTED argument-passing mechanism, 18–9
- ON ERROR phrase, 15–56, 15–59
- One-to-many relationship, 15–25
- ON EXCEPTION clause, 17–14
- Open mode
  - EXTEND, 8–14
  - INPUT, 8–14
  - I-O, 8–14
  - OUTPUT, 8–14
- OPEN statement
  - EXTEND, D–12
  - I-O, D–12
- Optimization, 20–1 to 20–20
  - file design, 20–10
  - I/O operations, 20–5
    - APPLY clause, 20–6
      - DEFERRED-WRITE phrase, 20–7
      - EXTENSION phrase, 20–7
      - FILL-SIZE ON phrase, 20–7
      - PREALLOCATION phrase, 20–6
      - WINDOW phrase, 20–7
    - sharing record areas, 20–8
    - using multiple buffers, 20–7
  - indexed files, 20–14
    - I/O, 20–15
      - alternate keys, 20–16
      - bucket size, 20–16
      - caching index roots, 20–19
      - calculating index levels, 20–19
      - file size, 20–17
      - index depth, 20–17
      - records, 20–15
  - numeric data representation, 20–1
    - ADD, 20–2
    - BINARY, 20–1
    - COMP, 20–1
    - COMP-3, 20–1
    - COMPUTE, 20–2
    - DISPLAY, 20–1
    - DIVIDE, 20–2
    - GO TO DEPENDING ON, 20–3
    - IF, 20–3
    - indexing, 20–3
    - mixing data-types, 20–2
    - MULTIPLY, 20–2
    - PACKED-DECIMAL, 20–1
    - PERFORM n TIMES, 20–4
    - PERFORM VARYING, 20–4
    - scaling, 20–2
    - SEARCH, 20–4
    - SEARCH ALL, 20–4
    - subscripting, 20–3
    - SUBTRACT, 20–2
  - relative files, 20–10
    - bucket size, 20–11
    - cell size, 20–11

Optimization
relative files (Cont.)
file size, 20–13
maximum record number, 20–11
sequential files, 20–10
OPTIONAL set membership class, 15–46
/OPTIONS qualifier
in LINK command, 2–34
Owner record, 15–9
occurrence, 15–12
OWNER test condition, 15–43, 15–44

P

Page
logical, 16–7
physical, 16–7
size definition, 16–10
Page body, 16–15
Page footing, 16–3
Page heading, 16–2
PARTS database demonstration package
creating, 15–2
loading, 15–2
PDF
compiling prior to use, E–3
VAX COBOL support, E–5 to E–12
placeholder processing, E–5, E–6
Performance, improving I/O, 8–2
PERFORM statement, D–8
Physical page
definition, 16–7
PIC P digits, D–9
Place markers
database, 15–3
Pointer, 15–30
Position
empty, 15–31
Print-controlled file, 8–4, 8–7
Print file, 9–4
PRIOR, 15–31
Procedure Calling and Condition Handling Standard, 18–40
Procedure calls, 18–34
Procedure names map, 2–21
Processing control, database, 15–2
Productivity tools, E–1
Program
accessing another data division, 18–8
compiling, 2–6
creating, 2–5
VAX COBOL GENERATOR
driver, 18–1
logic errors, B–6
run errors, B–5
faulty data, B–5
Program listing
reading, 2–17 to 2–29
Program switches
See switches
PROTECTED
access mode, 15–39
allow mode, 15–40
PROTECTED phrase, 17–15
PROTECTED RETRIEVAL
lock, 15–36
PROTECTED RETRIEVAL (Cont.)
lock keeplist, 15–36
usage mode, 15–40
PROTECTED UPDATE
usage mode, 15–40
Protecting records, 13–1 to 13–18
automatic record locking, 13–13
bypassing a record lock, 13–18
concepts, 13–1
error conditions, 13–16
hard locks, 13–16
soft record locks, 13–17
manual record locking, 13–14
using record locking, 13–12

Q

QUIT command (LSE), E–3

R

Read-only
access lock, 15–41
READY statement, 15–3, 15–29, 15–39
access mode, 15–39
allow mode, 15–39
record locking, 15–41
Realm
currency indicator, 15–30
nulling a currency indicator, 15–50
record locking level, 15–41
RECONNECT statement, 15–3
usage, 15–47
Record
attributes, 8–2
blocking, specifying, 8–2
cells, 10–1
current, 15–30
database, 15–8
fixed-length, 8–4
format, 8–3
junction, 15–18
locking
See Protecting records, Record locking, and File sharing
maximum size, 8–3
member, 15–9
nulling a currency indicator, 15–50
occurrence, 15–12
owner, 15–9
processing, 8–1 to 8–18
protection
See Protecting records
record-length field, 8–5
Selection Expression, 15–3
space needs on a physical device, 8–2
subschema definition, 15–8
types, 15–9
variable-length, 8–5
Record-length descriptions, multiple, 8–6
Record level
record locking, 15–41
Record locking
See also Protecting records
database, 15–8, 15–29
individual records, 15–41

Record locking (Cont.)
  minimizing lock out, 15–41
  no-read access locks, 15–41
  read-only locks, 15–41
  READY statement, 15–41
  realm level, 15–41
  record level, 15–41
  resolving locking conflicts, 15–41
  using keeplists, 15–41
Record locks
  releasing, 15–50
Record Management System (RMS), 8–1 to 8–2, 8–3, B–7 to B–11
  See also RMS special registers
  building a COBOL file specification, 8–14
  completion codes, 12–5
Record occurrence, 15–8
Record search access, 15–3
Record type
  currency indicator, 15–32
Record variable-length records
  sorting, 14–8
Redefinition
  implied when inspecting data, 7–19
REFERENCE
  argument-passing mechanism, 18–8
Reference format
  See Developing programs
Reference formats
  ANSI, 2–2
  terminal, 2–1
Reference modification, 5–11, D–6 to D–8
reformat
  executing, 19–2
  running, 19–2, 19–3
REFORMAT
  converting reference format, 2–2
  error messages, 19–4
  purpose of, 2–2
REFORMAT Utility, 19–1
Relationship
  many-to-many, 15–18, 15–19
  multiset, 15–18
  one-to-many, 15–25
  simple set, 15–14
Relation tests
  description, 4–2
  nonnumeric data, 5–3
Relative file, 10–1 to 10–15
  See also Optimization
  access mode, 10–5, 10–6
  access modes, 10–3
  at end condition, 12–1
  capabilities, 10–2
  creating, 10–5
  defining, 10–4
  deleting records
    randomly, 10–14
    sequentially, 10–13
  deleting records in, 10–13
  design considerations, 10–2
  File Status values, using, 12–3
  I/O statements, 10–3
  invalid key condition, 12–2
  INVALID KEY condition, 12–2
  open modes, 10–3
Relative file (Cont.)
  organization, 10–1
    advantages, 8–12
    disadvantages, 8–12
  random access mode, 10–6
  reading, 10–7
    dynamically, 10–9
    randomly, 10–8
    sequentially, 10–7
  record cell
    relative record number, 10–1
  record cells, 10–1
  rewriting records, 10–10
    randomly, 10–10, 10–12
    sequentially, 10–10, 10–11
  sequential access mode, 10–5
  tables, similarity to, 10–2
  updating, 10–10
  usage of, 10–2
  VAX RMS allocation for a cell, 10–2
Relative indexing, 6–17
  system overhead, 6–17
Relative record number, 10–1
Releasing locks
  records, 15–50
Replacement argument, 7–31
  list
    interference in, 7–32
    to inspect data, 7–31
Replacement value, 7–30
Replacing phrase
  to inspect data, 7–29
Report
  allocating a printer, 16–23
  bolding items in, 16–87
  bottom margin, 16–15
  components of, 16–2
  control footing, 16–3
  control heading, 16–3
  conventional, 16–7
    See also Conventional Report
    controlling spacing, 16–8
    example, 16–10
    line counter usage, 16–10
    logical page, 16–7
    page-overflow condition, 16–9
    printing, 16–10
  design, 16–1
  detail lines, 16–3
  footing area, 16–15
  GROUP INDICATE clause, 16–37
  Linage-file report, 16–14
    See also Linage-file
    controlling spacing, 16–16
    example, 16–21
    page advancing, 16–16
    page overflow, 16–16
      example, 16–17
    printing, 16–20
    using the LINAGE-COUNTER, 16–16
  logical page, 16–7, 16–14
  makeup, 16–7
  modes of printing, 16–23
  page body, 16–15
  page footing, 16–3
  page heading, 16–2

Report (Cont.)
  physical page, 16–7
  printing, 16–10
  printing totals before detail lines, 16–86
  problem solving, 16–80
    bolding items, 16–87
    example, 16–81, 16–84
    GROUP INDICATE clause, 16–85
    logical lines, 16–80
    physical lines, 16–80
    physical page, 16–86
    printing, 16–80, 16–86
      bold, 16–87
      details, 16–86
      totals, 16–86
    underlining, 16–87
  report footing, 16–3
  report heading, 16–2
  RESET phrase, 16–36
  See also Report Writer, 16–24
  spooling, 16–24
  streamlining your, 16–86
  top margin, 16–15
  total accumulating, 16–5
    crossfooting, 16–35
    crossfoot totals, 16–5
    defining, 16–34
    incrementing, 16–34
    RESET phrase, 16–36
    rolled forward totals, 16–5
    rolling-frward, 16–35
    subtotals, 16–5, 16–34
    UPON phrase, 16–36
  underlining in, 16–87
  UPON phrase, 16–36
  USE BEFORE REPORTING statement, 16–41
Report Writer, 16–24
  assigning a value in a print line, 16–31
  assigning the source for a print field, 16–31
  COLUMN NUMBER clause, 16–30
  CONTROL clause, 16–33
  detail reporting, 16–42
  examples, 16–43
    detail report, 16–44, 16–49, 16–57, 16–63
    input data, 16–43
    summary report, 16–71
  footings
    controlling, 16–33
    generating, 16–33
  GENERATE statement, 16–39
  headings
    controlling, 16–33
    generating, 16–33
  INITIATE statement, 16–38
  LINE clause, 16–29
  logical page
    defining, 16–25
    horizontal spacing, 16–30
    vertical spacing, 16–29
  PAGE clause, 16–25
  printing
    GROUP INDICATE clause, 16–37
  processing a Report Writer report, 16–38
    detail reporting, 16–42
    GENERATE statement, 16–39
    initiating, 16–38
    suppressing a report group, 16–42

Report Writer
  processing a Report Writer report (Cont.)
    terminating processing, 16–40
    USE BEFORE REPORTING statement, 16–41
  REPORT clause, 16–25
  Report Description entry, 16–25
  Report File
    defining, 16–25
  report groups, 16–26
  Report Section
    defining, 16–25
  SOURCE clause, 16–31
  specifying multiple reports, 16–32
  SUPPRESS statement, 16–42
  TERMINATE statement, 16–40
  using Declarative procedures, 16–41, 16–42
  VALUE clause, 16–31
RETAINING clause, 15–34
RETENTION option, 15–3
RMS special registers, B–7 to B–11
  RMS-CURRENT-FILENAME, B–7
  RMS-CURRENT-STS, 12–5, B–7
  RMS-CURRENT-STV, 12–5, B–7
  RMS-FILENAME, B–7 to B–11
  RMS-STS, 12–5, 13–9 to 13–11, B–7 to B–11
  RMS-STV, 12–5, B–7 to B–11
ROLLBACK
  automatic, 15–42
  DML statement, 15–42
  statement, 15–3
Rounding off arithmetic results, 4–9
RUN command, 2–42, 3–3
  syntax, 2–42
Running VAX COBOL programs, 2–42
Run-time
  errors, B–1
  input/output errors, B–7
Run-time errors, 2–43
Run unit, 15–29
  currency indicator, 15–30, 15–32
  current of, 15–32
  examples of COBOL, 18–1 to 18–3
  multiple COBOL program, 18–1
  non-COBOL program, 18–19
  nulling a currency indicator, 15–50

S

SCA, E–1 to E–12
  integration with LSE, E–1
  invoking, E–3
  preparing an SCA library, E–2 to E–3
Scaling
  and mixing data types, 20–4
  definition, 20–4
Schema, 15–6
Scope terminator, 15–59
Screen positioning
  absolute, 17–5
  relative, 17–5
SEARCH ALL
  advantages, 20–4
  requirements, 20–4
Search argument
  usage in REPLACING phrase, 7–30
Searching tables, 6–18

Search order
  definition for object module libraries, 2–36
SEARCH statement
  Format 1, 6–18
  Format 2, 6–19
Selection
  expression, record, 15–3
/SEQUENCE_CHECK qualifier
  in compile command, 2–14
Sequential file, 9–1 to 9–9
  See also Optimization
  access modes, 9–3
  at end condition, 12–1
  creating, 9–4, 9–5
  defining, 9–3
  design, 9–2
  end-of-file mark, 9–1
  end-of-volume label, 9–1
  extending, 9–4, 9–8
  File Status values, using, 12–3
  I/O statements, 9–3
  multiple volumes, 9–1
  open modes, 9–3
  organization, 9–1
  organization of
    advantages, 8–12
    disadvantages, 8–12
  print file, 9–4
  reading, 9–6
  rewriting records in, 9–7
  storage file, 9–4
  updating a, 9–7
Sequential search, 6–18 to 6–19
  function of AT END statement, 6–19
  requirements for, 6–18
Sequential search, see SEARCH statement
Set, 15–12
  forked, 15–16
  nulling a currency indicator, 15–50
  simple, 15–15
  single relationship, 15–14
  system-owned, 15–15
SET ABORT_KEY command, 3–4
Set membership class
  AUTOMATIC, 15–46
  FIXED, 15–47
  MANDATORY, 15–47
  MANUAL, 15–46
  OPTIONAL, 15–46
Set occurrence, 15–9, 15–12
Set owner
  access, 15–3
SET statement, 2–44
  Format 1, 6–18
  Format 2, 6–18
  indexing function, 6–17
Set type, 15–9
  currency indicator, 15–31
  current, 15–31
  ordering criteria, 15–31
Shareable images, 2–37, 2–41
  calling, 18–36
  creating, 2–37
  transfer vectors, 2–40
/SHARE qualifier, 2–37 to 2–40
Sharing
  USE procedures, 18–16
Sharing source text, 2–5
SHOW PLACEHOLDER command (LSE), E–5
SHOW TOKEN command (LSE), E–5
Sign
  conventions, 4–1
  tests
    description of, 4–3
Significant digits, 20–4
Simple set, 15–14, 15–15
Singular set
  See System-owned set
Size
  fixed-length tables, 6–1
  variable-length tables, 6–6
SIZE phrase, 17–16
SORT statement, 14–1 to 14–9
  ASCENDING KEY phrase, 14–1
  COLLATING SEQUENCE phrase, 14–6
    overriding, 14–6
  DESCENDING KEY phrase, 14–1
  file description, 14–1
  file organization, 14–6
  GIVING phrase, 14–2
    with the INPUT PROCEDURE phrase, 14–4
  INPUT PROCEDURE phrase, 14–3
    example, 14–3
    with the GIVING phrase, 14–4
  multiple sorts, 14–6
    example, 14–6
  OUTPUT PROCEDURE phrase, 14–3
    example, 14–3
    with the USING phrase, 14–4
  preventing I/O aborts, 14–8
    USE AFTER STANDARD declarative, 14–8
  sample programs, 14–9
  sorting variable-length records, 14–8
  USING phrase, 14–2
    with the OUTPUT PROCEDURE phrase, 14–4
  WITH DUPLICATES IN ORDER phrase, 14–5
Source Code Analyzer
  See SCA
Source file specification, 2–18
Source program
  listing, 2–18 to 2–20
Source programs
  compiling, 2–4
  creating, 2–3
  linking, 2–4
Source text, 2–18
Special characters
  nonnumeric data, 5–2
SPECIAL-NAMES paragraph, 5–5
Special register
  DB-CONDITION, 15–4, 15–58
  DB-CURRENT-RECORD-ID, 15–4
  DB-CURRENT-RECORD-NAME, 15–4
Stack, 15–36
/STANDARD qualifier, D–6 to D–12
  DIVIDE statement, D–6
  EXIT PROGRAM statement, D–10
  in compile command, 2–14
  INSPECT statement, D–8
  OPEN EXTEND, D–12

/STANDARD qualifier (Cont.)
- OPEN I-O, D–12
- overview, D–6
- PERFORM statement, D–8
- PIC P digits, D–9
- status codes, D–11
- UNSTRING statement, D–7
- variable-length tables, D–10

Status
- library return, 18–36

Status codes, D–11

Storage allocation
- byte boundaries, 6–8
- effect of fill bytes, 6–8
- table data, 6–7
- tables containing COMP or COMP SYNC items, 6–8

Storage file, 9–4

Storage map, 2–20

Storage schema, 15–6

STORE statement, 15–3

Storing numeric data, 4–1

Stream, 15–7

STRING statement, 7–1 to 7–6, D–7
- commaon STRING statement errors, 7–5
- DELIMITED BY phrase, 7–2
- multiple sending items, 7–1
- NOT ON OVERFLOW statement, 7–4
- OVERFLOW statement, 7–4
- POINTER phrase, 7–4

Subprogram
- contained, 18–1
- nested, 18–1

Subschema, 15–6
- Data Storage Description Language, 15–6
- entry, 15–2
- map listing, 15–5
- Record Definition, 15–8

SUB-SCHEMA SECTION, 15–5, 15–59
- entry, 15–4

Subscript
- definition, 6–14

Subscripted moves
- nonnumeric data, 5–10

Subscript evaluation, D–6 to D–8

Subscripting
- with data-names, 6–16

Success/failure tests, 4–4

Switches
- checking, 2–44
- controlling, 2–44
- example using, 2–45
- order of evaluation, 2–44
- program, 2–43 to 2–45
- SET statement, 2–44
- setting internally, 2–44
- setting switches for a process, 2–44
- using, 2–43

Symbol
- record, 3–3

Symbols
- library condition values, 18–36
- numeric editing, 4–7

System
- libraries, 2–37
- routines, 18–33
  - calling subroutine, 18–34

System
- routines (Cont.)
  - function calls, 18–33
  - function results, 18–36

System-owned set, 15–15

System spooler, 16–10

T

Table elements
- initializing, 6–11

Table handling, 6–1 to 6–25
- See also Tables

Tables
- See also Storage allocation
- See also Subscript
- accessing
  - with indexes, 6–14, 6–16
  - with literal subscripts, 6–14
  - with SEARCH statement, 6–18
  - with subscripts, 6–14
- accessing table elements, 6–14
- access methods, 6–14
- assigning values using the SET statement, 6–17
  - integer value, 6–18
- binary search, 6–19
  - with keys, 6–20
  - with multiple keys, 6–20
- defining fixed-length
  - multidimensional, 6–4
  - one-dimensional, 6–1
- defining variable-length tables, 6–6
  - ascending and descending keys, 6–6
  - size, 6–6
- definition, 6–1
- fixed-length
  - specifying size, 6–1
- incrementing an index value using the SET statement, 6–18
- index data item
  - where defined, 6–3
- index data items, 6–17
- initializing, 6–11
  - effect of fill bytes, 6–14
  - INITIALIZE statement, 6–14
  - redefining group level, 6–12
  - with OCCURS clause, 6–11
  - with VALUE clause, 6–11
- initializing indexes, 6–17
- multidimensional
  - accessing with subscripts, 6–15
  - defining, 6–4
  - OCCURS clause, 6–1
    - indexes, 6–1
    - keys, 6–1
- relative indexing, 6–17
- See also Index data item, 6–17
- sequential search, 6–18
  - AT END statement, 6–18
- storage allocation, 6–7
  - using the SYNCHRONIZED clause, 6–8
- subscripting with data names, 6–16

Tally argument
- to inspect data, 7–24

Tally counter
- to inspect data, 7–24

Terminal format, 2–1
  See Developing programs
  advantages of, 2–1
Terminal-to-ANSI
  command string, 19–3
  format conversion, 19–2
Test condition
  member, 15–43
  owner, 15–43, 15–44
Testing
  nonnumeric data items, 5–3
  numeric items, 4–2
    relational tests, 4–2
  sample session, 15–60
  VAX COBOL DML programs, 15–58
Top margin, 16–15
TRACEBACK facility, B–1
/TRACEBACK qualifier
  in LINK command, 2–33
Transaction
  database, 15–38
/TRUNCATE qualifier
  in compile command, 2–15

U

Unstringing data, 7–6
UNSTRING statement, 7–6 to 7–17, D–7
  common errors, 7–16
  COUNT phrase, 7–12, 7–14
  DELIMITED BY phrase, 7–8
  DELIMITER phrase, 7–13
  delimiters, 7–10, 7–11
  function, 7–6
  multiple delimiters, 7–12
  multiple receiving items, 7–6, 7–9
  OVERFLOW statement, 7–16
  POINTER phrase, 7–14
  TALLYING phrase, 7–15
UPDATE access mode, 15–40
Updating database records, 15–3
Usage mode, 15–40
  BATCH RETRIEVAL, 15–40
  BATCH UPDATE, 15–40
  CONCURRENT RETRIEVAL, 15–40
  CONCURRENT UPDATE, 15–40
  EXCLUSIVE RETRIEVAL, 15–40
  EXCLUSIVE UPDATE, 15–40
  PROTECTED RETRIEVAL, 15–40
  PROTECTED UPDATE, 15–40
USE procedure
  use of, B–11
User-defined names
  cross reference list of, 2–22
/USERLIBRARY qualifier
  in LINK command, 2–33
User work area (UWA), 15–2, 15–5
USE statement, 15–59
  FOR DB-EXCEPTION, 15–57
  purpose of, 18–16
  Rule 2 example, 18–18
USING phrase
  usage of, 18–8
Utility
  Database Query (DBQ), 15–59
  DMU, 15–1
Utility (Cont.)
  REFORMAT, 19–1
UWA
  See user work area

V

VALUE
  argument-passing mechanism, 18–8
VALUE OF ID clause
  using the, 8–9
Variable-length records
  creation of, 8–5 to 8–7
  record-length field, 8–5
Variable-length tables, D–10
VARYING phrase, D–8
VAX COBOL
  Debugger support, 3–9
  features, 1–1 to 1–2
  overview, 1–1 to 1–2
VAX COBOL compiler
  function, 2–6
VAX COBOL data manipulation language
  See Data manipulation language
VAX COBOL GENERATOR, E–23
  See COBOL GENERATOR, E–22
VAX DBMS Data Definition Language (DDL), 15–4
VAX DEC/Code Management System
  See CMS
VAX Procedure Calling and Condition Handling
    Standard, 18–40
VFU printing
  customizing vertical spacing, 8–18 to 8–19
  rapid vertical line positioning, 8–18 to 8–19
Video forms, 17–1 to 17–34
  accepting data, 17–13, 17–17, 17–18
    ON EXCEPTION with CONVERSION, 17–1
  ACCEPT statement, 17–2
    CONTROL KEY IN phrase, 17–21
      characters returned, 17–22
      example, 17–28
    DEFAULT phrase, 17–18
      example, 17–19
    EDITING phrase, 17–31
    NO ECHO clause, 17–17
    options, 17–13
  assigning character attributes, 17–8
  assigning default values to data fields, 17–18
  clearing the screen, 17–3
  COLUMN NUMBER phrase, 17–5
    PLUS option, 17–7
  CONVERSION clause, 17–10, 17–13
    example, 17–10
  defining keys, 17–21
  designing, 17–2
  displaying data, 17–10
  DISPLAY statement, 17–2
  editing, 17–31
  ERASE phrase, 17–3
    options, 17–3
  LINE NUMBER phrase, 17–5
    PLUS option, 17–7
  positioning the cursor, 17–5
    absolute, 17–5
    relative, 17–5
  protecting your screen, 17–15

Video forms (Cont.)
- PROTECT phrase, 17–15

VMS Debugger
- see Debugger

VMS Modular Programming Standard, 18–40

VMS Usages, 18–29
- VAX COBOL equivalents, 18–29

W

/WARNINGS qualifier
- in compile command, 2–15

WHERE clause, 15–55, 15–59

WRITE command (LSE), E–5

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX COBOL User Manaul
AA–H632E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ____________________

____

____

What I like best about this manual is ____________________

____

____

What I like least about this manual is ____________________

____

____

I found the following errors in this manual:

Page Description

______ ____________________

______ ____________________

______ ____________________

______ ____________________

______ ____________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ____________________ Dept. __________
Company ____________________ Date __________
Mailing Address ____________________
____________________ Phone __________

— Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

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

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VAX COBOL User Manual
AA–H632E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ____________________

____

____

What I like best about this manual is ____________________

____

____

What I like least about this manual is ____________________

____

____

I found the following errors in this manual:

Page Description

_____ ____________________

_____ ____________________

_____ ____________________

_____ ____________________

_____ ____________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** _____ of the software this manual describes.

Name/Title ____________________ Dept. __________

Company ____________________ Date __________

Mailing Address ____________________

____________________ Phone __________

-- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

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

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here